Constraint Solving for Beautiful User Interfaces: How Solving Strategies Support Layout Aesthetics

Clemens Zeidler
Department of Computer Science
38 Princes Street
Auckland 1010, New Zealand
czei002@aucklanduni.ac.nz

Christof Lutteroth
Department of Computer Science
38 Princes Street
Auckland 1010, New Zealand
lutteroth@cs.auckland.ac.nz

Gerald Weber
Department of Computer Science
38 Princes Street
Auckland 1010, New Zealand
gerald@cs.auckland.ac.nz

ABSTRACT

Layout managers provide an automatic way to place controls in a graphical user interface (GUI). With the wide distribution of fully GUI-enabled smartphones, as well as very large or even multiple personal desktop monitors, the logical size of commonly used GUIs has become highly variable. A layout manager can cope with different size requirements and rearrange controls depending on the new layout size. However, there has been no research on how the distribution of additional or lacking space, to all controls in the layout, effects aesthetics.

Much of the previous research focuses on discrete changes to layout. This includes changing the layout elements [15], or swapping around layout elements [7]. In this paper we focus strictly on the optimization of resizing of GUI components, and in this area we focus on rather subtle changes. This paper describes and compares strategies to distribute available space in a visual appealing way. All strategies are modeled with a constraint-based layout manager, since such a layout manager can be used to describe a wide range of layouts. Some aesthetic problems of constraint based layout managers have been identified and solutions have been provided.

In a user evaluation three solving strategies, equal distribution, weighted distribution and a minimal deviation, have been compared. As a result, the minimal deviation approach seems to be a good strategy for large and small layout sizes. The minimal deviation and the equal distribution strategy is best at large layout sizes while the weighted distribution approach seems to perform better at small layout sizes. Furthermore, the evaluation shows that layouts with a high degree of symmetry are clearly preferred by the users.

Categories and Subject Descriptors

H.5.2 [**User Interfaces**]: Evaluation/methodology

General Terms

Experimentation, Measurement, Performance

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CHINZ '12, July 02 - 03 2012, Dunedin, New Zealand Copyright 2012 ACM 978-1-4503-1474-9/12/07 $15.00.

Keywords

Aesthetics, Constraint Based Layout, GUIs, Layout Manager

1. INTRODUCTION

In early GUI frameworks, controls such as buttons or text views had to be placed manually at a fixed position and with a fixed size, e.g. in Microsoft Foundation Class library (MFC). This can become very tedious as soon as new controls have to be inserted or the layout of existing controls has to be modified. Modern GUI frameworks solve this problem by offering layout managers which allow developers to position the controls in a user interface more abstractly. Rearranging and modifying a GUI can become easier and a re-layout of the GUI at different window sizes can be done automatically by the layout manager. Furthermore, the use of a layout manager often leads to more consistent GUIs since it can make sure that the controls, the layout items of the layout, are well aligned and consistently spaced.

To setup a GUI using a layout manager, the developer has to specify a set of layout specifications. To keep things as simple as possible for the developer, the layout specifications that are required to define a layout with good visual appearance should be small. This means that layout specifications usually do not specify every single detail about a layout, but leave some of the details to the layout manager. Interesting are the cases when the available space in a GUI is insufficient or when there is more space available than needed. Both cases happen regularly when windows are resized, e.g. to adjust a GUI to the size of the available screen space. For example, consider a simple layout containing just two controls in a horizontal row, spanning the complete window size (Figure 3). Depending on the window size, the layout manager has to decide what control width is the best to yield a visually appealing result. An important hint for that decision can be the control's preferred size, which describes the size preferred by the control in the absence of any other constraints.

This paper addresses the following overall research question: how should the space available in a GUI be distributed among the layout items? To answer this question, we need to consider what layouts are perceived as aesthetically pleasing, as well as the solving strategies that determine the layouts. We motivate a minimal deviation strategy to distribute the available space. In a constraint based layout system this can be implemented by using a quadratic solving strategy and we show how this solving strategy is superior to a linear solving strategy. In this paper we focus on a subproblem of this general question. We focus on a very restricted class of layouts, and we are considering a restricted class of GUIs, namely GUIs that

are comprised solely of buttons. This is worthwhile since it allows us to isolate possible cross-influencing factors. Our restricted experiments yield interesting results that can now be tested in other more general classes of layout.

In the field of typesetting and document layout much research has been done about how to create accessible and clear document layouts using automated layout systems [9]. However, no study has been undertaken as to how different solving methods compare to each other with respect to aesthetics. The compared solving strategies are equal distribution, weighted distribution and minimal deviation. All solving strategies were implemented and evaluated using the Auckland Layout Model (ALM) [11] a constraint based layout manager. However, the results presented in this paper are not limited to the ALM and can be applied to other layout managers as well. A constraint based layout system is able to model all interesting layouts and solving strategies for our research. Pitfalls of the solving strategies are analyzed and solutions for them are presented.

We performed a user evaluation where the equal distribution, weighted distribution and minimal deviation strategies were compared to each other. Our study shows that the minimal deviation strategy gives good results for small and large layout sizes, while the other solving strategies only demonstrate their strength either at small or at large layout sizes.

In the next section an introduction to layout managers and the different types of common layout classes is given. Section 3 gives an overview of related work, including how other layout manager distribute available space and how Gestalt principles are used in other fields to get visually appealing results. A detailed description of constraint based layout and how systems of layout constraints can be solved using linear and quadratic objective functions is given in Section 4. Section 5 compares the effect of linear and quadratic objective functions on the visual quality of the layout. In Section 6 a user evaluation is presented that provides answers to the research question. It shows how the layouts produced by different solving approaches are perceived aesthetically by users. The results indicate which solving strategies generally lead to more beautiful layouts.

2. LAYOUT MANAGERS (OVERVIEW)

In early GUI developer toolkits, GUI items had to be placed manually in certain fixed position. This static approach can be tedious and error prone, especially during the design process where GUI items are moved around quite often. In this case, already placed items have to be rearranged when inserting a new element. A layout manager assists the developer in setting up dynamic graphical user interfaces. GUI items managed by a layout manager are placed automatically following certain rules.

Furthermore, the layout manager can adjust the layout, e.g. when the user resizes a window. In this case a layout manager repositions GUI items dynamically to fit into the new size. Another case is a font change or a change of the application language, e.g. from English to German. Generally, in both cases the displayed text will change its size and thus requires the rearrangement of GUI items. Using a static approach makes it frustrating for the developer to handle such cases. However a layout manager handles these use cases with ease without further work from the developer.

Anything that can be placed into a layout is called a *layout item*. The most important layout items are GUI controls, e.g. buttons or text labels. Other important layout items are spacers, which are invisible and can be placed between other layout items. A spacer can occupy a fixed amount of space or can act like a spring to push other layout items aside. In this way, a spacer can be used to refine a layout and give it the desired shape. In order to create nested layouts it is important to have an item that can hold another layout. This can easily be achieved by treating a layout as a special layout item. In the following we assume that layout items are rectangular.

In general a layout item has a *minimal*, a *maximal* and a *preferred size*. The preferred size is the size the layout item should assume if there are no other constraints for the item size. This can be illustrate with a pinched sponge which, after releasing it, expands to its original or preferred size. From the size values for each layout item in the layout, the corresponding size values of the complete layout can be calculated. Similar to a single layout item, the preferred size of a whole layout is the size it should assume if there are no other constraints. Notice that in a layout of minimal size, not all layout items may have their minimal size. This is because other larger layout items may be preventing the layout from shrinking further. In general, a layout or a layout item has a size different from its preferred size. Thus, there is a *size discrepancy* between the actual size and the preferred size.

There are different existing types of layout in various frameworks (see Section 3). Most of these frameworks provide special layout classes for special types of layout problems, e.g. group layout, grid layout or flow layout. Usually these special layouts can be combined by creating nested layouts. These most common layout classes are described briefly in the following sections.

2.1 Group Layout

A group layout is a simple 1-dimensional layout that can hold items side by side in a single row or column. There are two main variants of this layout, a horizontal group layout which can hold a row of items and a vertical group layout which can hold a column of items.

By nesting horizontal and vertical group layouts many useful layout configurations can be created. However, this type of layout is not sufficient for more complex layouts, e.g. a link between layout items in two different group layouts is not possible.

2.2 Grid (Bag) Layout

Some of the shortcomings of a group layout can be avoided by using a grid layout, also known as a table layout. Here, a layout item can be placed in a 2-dimensional table. A layout item can occupy more than one cell in the grid, and thus it is possible to create complex layouts. Furthermore, it is possible to create a link between items not directly adjacent, e.g. by placing them in the same row or column. This makes the grid layout reasonably flexible and powerful.

The grid layout can be tuned by giving the rows and columns special weightings. This is useful in specifying which row and column should use more space compared to the other rows and columns.

2.3 Flow Layout

A flow layout is basically a horizontal group layout that can span over multiple rows if items do not fit into one row. This is comparable with a line of text in a word processor: if the end of the line is reached, the text is continued in the next line. An example of a flow layout is a button bar that becomes a multi-line button bar in case the window becomes smaller than the button bar width.

2.4 Constraint-Based Layout

In a constraint-based layout, the layout specifications are described by constraints using linear equalities and inequalities. An example for a simple GUI constraint for a horizontal two-button layout is

$$button1_{right} = button2_{left}.$$

There are two types of constraints: *hard constraints*, which have to be satisfied, and *soft constraints*, which can be violated if necessary. The way how soft constraints are solved depends on the implementation of the constraint solver and can be used to control the final visual appearance of the layout. Layouts created with a group or grid layout can alternatively be created using a constraint-based layout manager. However, constraint-based layout managers support even more complex layouts, which makes them very powerful. For example, in a grid layout, layout items are always aligned to a outer fix grid while in a constraint-based layout a layout item can be aligned relative to another layout item and so is not bound to the fix grid. Constraint-based layouts are discussed in detail in Section 4.

3. RELATED WORK

3.1 Layout Managers

Some of the most prominent GUI frameworks that provide layout mangers are Qt [1], Java AWT [16], Cocoa [2], Windows Forms [13], GTK+ [3] and wxWidgets [4]. Most of these layout managers distribute available space in a simple way, however, different managers use different methods of distribution. How exactly available space is distributed is neither well documented nor is there any explanation why a particular method of distribution has been chosen.

For example, the Grid Bag Layout from the Java AWT framework distributes the layout size discrepancy using weights which can be assigned to columns and rows (weighted distribution). This means generally that an item grows or shrinks by

$$\Delta size_{item} = discrepancy \cdot weight_{item} \,/ \sum_{i \in items} weight_i.$$

The Qt toolkit follows a different approach and distributes or takes available space available space equally from all items in the layout (equally distribution):

$$\Delta size_{item} = discrepancy \,/ \#items$$

Another approach is implemented in the Haiku OS[5]. Here, for all items in a group layout the sum of the quadratic item discrepancies is minimized (minimal deviation). This is described in more detail in Section 4.3.2.

Similar to the ALM layout manager used in this research (see Section 4), the Java layout class SpringLayout and the layout manager of the Mac OS Cocoa API, Auto Layout, is based on constraints. In Auto Layout, the programmer can specify linear constraints in the form $y = m \cdot x + b$ between two variables x and y. These variables could, for example, be the width or the edge of a layout item. However, this approach is not as powerful as general constraint-based layout managers such as ALM, which allow to specify more complex constraints and make it possible to create layouts in a more abstract way. For example, constraints with multiple variables or variables not connected to any layout items are not possible. A wide overview of different techniques for solving GUI constraints is given in [1].

Besides methods of distributing the discrepancy, other approaches have been tried to adapt a layout to different sizes. SUPPLE is an automated system that can adapt layouts to changes in display size, in particular to different devices. The system supports discrete changes of layout items, i.e. it changes the controls that are used within an input form depending on the available space [15]. Optimizations of more experimental layouts has been studied for the GADGET framework [7]. For example, GADGET targets problems like how a GUI can be automatically generated using certain optimization rules.

3.2 Layout Aesthetics

The scientific field of Gestalt psychology [10] covers principles about the perception of shapes and groups of shapes. For example, the law of equality states that similar shapes are perceived as a group, and the law of proximity states that shapes which are placed close to each other are perceived as a group. These findings can be transferred to user interfaces, where aligned controls are perceived as a group. The law of equality can be applied when items are placed in the same row or column and share the same height or width. When items are aligned close to each other the law of proximity can be applied. This can be used to group related controls [8] to achieve a clear layout appearance. Gestalt psychology is the basis for many fields and is used in most aesthetics related papers.

Gestalt principles can also be applied to conventional typography [9] as well as to web documents [3]. Here they can help to structure a document to make it easier to read and understand. Another application is the pagination problem, which targets the question how to best distribute content over multiple pages, e.g. how to place figures and text in a complex document to produce visually pleasing results [14]. In the field of graph layout, a set of similar layout aesthetics has been used to optimize graphs [4].

The knowledge of Gestalt principles can help to layout UI objects in a more pleasant way [3]. Layout mangers often make it easier to set up good layouts, without having to define the final layout in all its details. However, they do not apply Gestalt principles all by themselves: if Gestalt principles are used depends on how a GUI designer specifies a layout, not on the layout manager.

4. CONSTRAINT-BASED LAYOUTS

In a constraint-based layout manager, user interface layouts are specified mathematically as constraint problems. This makes it possible to create complex and flexible layout specifications, and calculate actual layouts using numerical constraint solving methods [12]. The Auckland Layout Model (ALM) [11] is the constraint-based layout manager used for this research, and a suitable representative of constraint-based layout in general. ALM was chosen because all common layout specifications and solving strategies can be emulated using ALM. The discrepancy distribution methods discussed here can also be used with other constraint-based layout managers.

Each layout item in ALM is connected to a *tab* on each of its four borders; a tab is a horizontal or vertical grid line in the layout. Relations between tabs, and so between

[1] Qt – a cross-platform application and UI framework, 2011 `http://qt.nokia.com/products/`
[2] Cocoa Auto Layout Guide, 2011 `http://developer.apple.com`
[3] The GTK+ project, 2011, `http://www.gtk.org/`
[4] wxWidgets Cross-Platform GUI Library, 2011, `http://www.wxwidgets.org`
[5] The Haiku Operating System, 2011, `http://www.haiku-os.org`

different layout items, can be specified using layout constraints. For example, to place two buttons beside of each other, the right border of the left button shares a tab with the left border of the right button. A rectangular space, surrounded by tabs, that is occupied by a layout item is called an *area*.

The developer is able to add arbitrary constraints to the layout specification. For example, an additional constraint like

$$width_{button1} = 2 \cdot width_{button2}$$

could be used to ensure that the width of *button*1 is two times as big as the with of *button*2. Compared to a grid layout where rows and columns have a fixed order, tabs do not have a strict order which allows more flexible layouts.

There are two kinds of constraints used to specify a layout. First, hard-constraints are needed to set the fixed properties of a layout item, such as its minimum and maximum size. Secondly, soft-constraints are used to give a hint how a layout item should look like, e.g. the item size should be close to the preferred item size. The actual influence of the soft-constraints on the final layout varies depending on the solving strategy used.

4.1 Specifying Constraints

Constraints that have to be satisfied exactly are called hard-constraints. A hard-constraint could be either an equality or an inequality constraint, and can be described by

$$\begin{aligned} A_i \cdot \overline{x} &= \overline{b}_i \qquad i \in equalities \\ A_i \cdot \overline{x} &\geq \overline{b}_i \qquad i \in inequalities \end{aligned}$$

Here A is the constraint matrix, $\overline{x}$ is the tab or variable vector and $\overline{b}$ is a constant vector.

Soft constraints are specified in the same way as hard constraints, but because they can be violated, it is often possible to prioritize them. In case of a conflict between soft constraints, the constraint with the smallest priority will be violated most. Similar to hard constraints, developers may add custom soft-constraints to a layout specification.

The most important use case for soft constraints is the specification of preferred sizes for layout items. Preferred size constraints are a common way to give layout items a reasonable size, i.e. make them as close to their preferred size as possible, while still accommodating size adjustments. For example, the preferred width of a button is the width needed to display the button label plus some extra space for the border. This border is actually not completely needed and labels can be abbreviated, so the button could be narrower than the preferred width. Similarly, it is possible to make the button wider than the preferred width. The solver has to decide which width is the best, considering that the item needs to fit into the overall layout, e.g. that it aligns with its neighbors. Using a preferred width soft constraint, it will choose a size as close to the preferred width as possible. The mathematically description of soft-constraints depends on the objective function used, and is discussed later in Section 4.3.

4.2 Rows and Columns

A layout item is always connected to two horizontal and two vertical tabs. The two horizontal tabs can naturally be regarded as a row, and the two vertical tabs as a column. Multiple layout items sharing the same horizontal or vertical tabs also share the same row or column, respectively. In this way there can be interruptions in a row or a column, e.g. there could be another item between two items in a row that is only connected to one or even none of the horizontal row tabs. This is not the traditional definition of rows and columns but allows a simple grouping of the generally unordered tab system.

4.3 Layout Optimization

A suitable constraint solver for user interface constraints must be able to solve the hard-constraints (Section 4.1) and must also handle soft-constraints. Soft-constraints are described separately from hard-constraints by a scalar objective function. In general, this objective functions is minimized while satisfying the hard-constraints at the same time.

4.3.1 Linear Objective Function

The simplest approach for an objective function is a linear objective function, which can be utilized to describe soft-constraints and optimized using linear programming. Technically, this is done by first adding a hard-constraint for each soft-constraint, of the form

$$\sum_i a_{soft,i} \cdot x_i + s_{shrink} - s_{grow} = b_{soft}.$$

Here, s_{shrink} and s_{grow} are two new positive slack variables which express that the soft-constraint can be violated in both direction. The goal is to keep both slack variables as small as possible to violate the soft-constraint as little as possible. The penalty factors p_{shrink} and p_{grow} can be used to prioritize a soft constraint, with a large penalty factor meaning that growing or shrinking away from the optimal values is suppressed.

Finally, this leads to the linear objective function, which is the weighted sum of all slack variables, and which must now be minimized [11]:

$$\sum_{i \in soft} p_{grow,i} \cdot s_{grow,i} + p_{shrink,i} \cdot s_{shrink,i} \to min. \quad (1)$$

A suitable solver for this purpose is lp_solve [2], which uses the simplex algorithm [5].

One of the problems of a linear objective function is that minimizing (1) generally leads to many valid solutions; the linear approach is non-deterministic. This means that not all soft-constraints are violated in a uniform way, e.g. only a few constraints are violated and its not clear which constraints are violated.

4.3.2 Quadratic Objective Function

A deterministic approach is to minimize the square of the deviation from a desired target value. For simple preferred size constraints this can be written as

$$\sum_{i \in soft} (x_i - pref_i)^2 \to min.$$

More general, the soft-constraints in matrix form

$$A_{soft} \cdot \overline{x} = \overline{b}_{pref}$$

can be used to form the quadratic objective function

$$\frac{1}{2} x^T A_{soft}^T A_{soft} x - b_{pref}^T A_{soft} x \to min.$$

Replacing $A_{soft}^T A_{soft} = G$ and $-b_{pref}^T A_{soft}^T = g^T$ this could be simplified to:

$$\frac{1}{2} \overline{x}^T G \overline{x} + \overline{g}^T \overline{x} \to min$$

This is a known quadratic programming optimization problem and could, for example, be solved using the Active Set method [6]. To use the Active Set method, first a valid

base solution for the hard constraints has to been found. Continuing from that base solution, the Active Set method minimizes the quadratic objective function, while staying in the solution space of the hard-constraints.

Soft constraints can be weighted by a penalty p. For the simple preferred size constraints, this leads to the objective function

$$\sum_{i \in soft} p_i^2 \cdot (x_i - pref_i)^2 \rightarrow min.$$

Constraints with larger penalties p will be violated less than constraints with smaller penalties. Notice that compared to the linear objective function only one penalty factor for growing and shrinking can be applied. Thus the developer can not specify if a layout item is more likely to grow than to shrink. However, by combining two soft inequality constraints, e.g. $x > b$ and $x < b$, a similar growing and shrinking behavior can be achieved.

A soft inequality constraint is an inequality constraint that can be violated if necessary. Soft inequality constraints are not directly supported using a quadratic objective function but can be constructed from a hard inequality constraint and a normal soft constraint. To construct a soft inequality constraint a positive slack variable s has to be added to the basic inequality. For example,

$$\sum_i c_i x_i < r_i \text{ becomes } \sum_i c_i x_i - s < r_i,$$

$$\sum_i c_i x_i > r_i \text{ becomes } \sum_i c_i x_i + s > r_i.$$

This means s can always be chosen to satisfy the inequality. However, only if other constraints require to violate the soft inequality constraint, then s should be greater than zero. This can be achieved by adding the soft constraint $s = 0$ with a sufficient high penalty.

5. AESTHETICS PROBLEMS OF CONSTRAINT-BASED LAYOUT

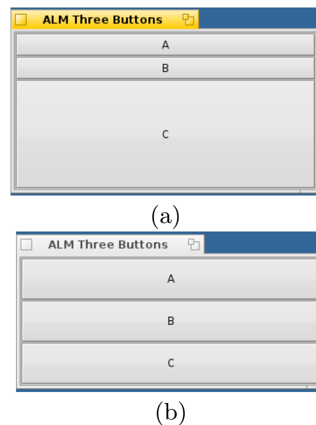
Describing soft-constraints using a linear or a quadratic objective function leads to different behaviors when distributing the size discrepancy to the layout items. In this section, the behavior of preferred size soft-constraints is discussed from an aesthetic point of view.

5.1 Linear Objective Functions Cause Non-determinism

An example for a simple homogeneous layout is a layout containing just three buttons with exactly the same properties. Figure 1 a) shows the resulting layout solved by lp_solve (linear objective function). As expected, all hard-constraints are satisfied. The soft-constraints for the first two buttons are matched exactly, meaning the height is equal to the preferred size of the buttons. The only violated soft-constraint is the preferred height of the third button.

From the aesthetic point of view, this layout configuration looks odd. Because all buttons have the same properties, one would expect that all buttons take the same amount of space. The height ratio between the different buttons is not specified by the layout, but is a result of how the solver solved the constraints. It is theoretically even possible for the ratio to changes nondeterministically during resizing. Following Gestalt principles, the three buttons with identical properties should be perceived as a group, which is not the case here.

One solution to this problem is to manually specify a size relation between the related items, but this is extra



(a)

(b)

Figure 1: Simple three button layout with all buttons having the same properties. The layout is solved using a) a linear and b) a quadratic objective function.

work for the developer. A better solution is to leverage the advantages of a quadratic objective function, which minimizes the deviation to the preferred item size for each item, not just the sum of deviations over all the items. Figure 1 b) illustrate how a quadratic objective function leads to the desired uniform result.

5.1.1 *Proportionality Scale Variables*

The problem of the linear objective function, as described above, could partially be solved by introducing additional free scaling variables s. These scaling variables can be used to make the size of a layout item proportional to their preferred size. To do so, the preferred size soft constraints have to be rewritten to:

$$x - x_{pref} \cdot s = 0.$$

However, the approach can only be used in special cases, i.e. when all layout items can get a size proportional to their preferred size. In case this requirement is not satisfied anymore, the soft-constraints have to be violated again which leads to the same problem of non-determinism as described previously.

5.2 Spring Effects

One disadvantage of minimizing the deviation to the preferred sizes is that this sometimes leads to an unwanted *spring effect*. This is an analogy between preferred size constraint and mechanic springs, pulling or pressing an item to its preferred size.

The problem occurs when multiple “springs” are coupled and thus their strengths combined to a resulting force. Such a coupling could, for example, be observed in a multi-row layout like in Figure 2. In the first row three “springs” and in the second row two “springs” are coupled. Because each button has the same preferred height, this results in the same descriptive spring force F_s for each button. Thus the first row has a “spring” force of $3 \cdot F_s$ and the second row a force of $2 \cdot F_s$. For the solved layout this means both rows have a different height.

This is certainly a limitation of constraint-based layout because since all items have the same properties one would expect, according to the equality Gestalt law, two equal sized rows. In such a case a relation between rows has to be specified explicitly, e.g. by applying a hard-constraint that keeps the height of both row constant. Another, more general solution is to define preferred size constraints on whole rows and columns only.

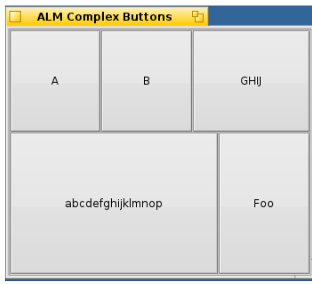


Figure 2: Spring effect: Three buttons in the first row pulling stronger to their preferred size than the two buttons in the second row.

A row or column is defined by two tabs and at least one layout item between these tabs. Rows are bordered by horizontal tabs and columns by vertical tabs. Layout items connected to the same two tabs are associated with the same column or row (Section 4.2).

To solve the spring effect problem, the preferred size constraint is only applied for rows and columns and not for each layout item. Since there could be multiple items in a row or a column, the weighted average of the preferred sizes of the items is used, using penalties as weights. In the upper example, this has the affect that both rows have the same preferred height, and thus get the same height when solving the layout.

6. EVALUATION

There are many ways to place layout items in a layout. Similar to a typesetting system, an important goal is to to create layouts that are aesthetically pleasing for the user [9]. In this section, we evaluate different solving strategies with regard to aesthetic perception.

The difference in the perceived aesthetics of layouts generated by different solving strategies are small and difficult to measure. For some users it may not be obvious what the differences are between the same layout rendered with different solving strategies. Furthermore, the criteria of aesthetics are subjective and vary between users.

An important aspect of this evaluation is the analysis of the resize behavior of a layout. Layouts should look pleasing at different sizes, not just for a particular initial size. Therefore, the evaluation will consider different sizes of the same GUI, a small size close to the layout minimum size and a large size approximately twice as large as the preferred layout size. Here, three solving strategies that place items in a 1-dimensional and 2-dimensional layout are analyzed. All layouts and solving strategies described in the following were implemented using ALM's constraint system.

6.1 Single-Row Layouts

A very simple layout is a layout consisting of just a single row, e.g. a group of buttons arranged besides each other. Three different solving strategies to distribute the size discrepancy to the buttons in the row are evaluated.

First, *equal distribution* gives each item the exact same amount of space in a line. Here the preferred size of an item is not take into account. Note that the theoretical minimum size of a layout can generally not be reached here, i.e. when one of the layout items reached its minimum size then the other ones cannot be made smaller either. In practice this can be solved by violating the equality constraint once an item reaches its minimum size.

Secondly, *weighted distribution* keeps the size ratio between items in a line constant. This means a weight is assigned to each item. The layout item sizes are given by

$$Size_{item} = Size_{layout} \cdot w_{item} / \sum_{i \in items} w_i.$$

For this evaluation, the item weight is chosen as the relative item size at a small initial layout size, where the items are close to their preferred size.

Thirdly, the item sizes are determined by calculating the *minimal deviation* from the preferred size for each layout item. This can be achieved with a solving strategy that uses a quadratic objective function (see Section 4.3.2). For very large layouts, the minimal deviation approach converges to the equal distribution approach because the preferred size becomes small compared to the actual item size. For layout sizes close to the layout's preferred size, the result is close to the weighted distribution because the weights are chosen to match the preferred size.

An example for a simple two-button row at small layout size is shown in Figure 3. For simplicity, no item maximum size is taken into account. Maximum sizes result in more complex layouts and thus make the analysis of the evaluation results more complicated. For example, when the maximum of one layout item in an equally distribution layout is reached, the layout cannot be resized any further.

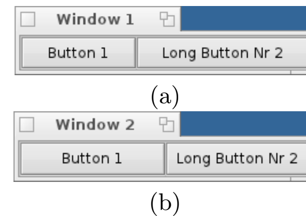


Figure 3: Two different solving strategies for a simple two-button layout: (a) minimal deviation and (b) equal distribution.

6.2 Multi-Row Layouts

Another interesting question is how the three different solving strategies from the previous section perform in a multi-row layout. In this evaluation, a two-row and a three-row layout is evaluated with regard to its perceived aesthetics. The two-row layout has three buttons in the first row and two button in the second row. In the three-row layout, another row with two buttons is appended.

The first case we considered in this study is a minimal deviation approach, where the sizes of the items are chosen as close to their preferred sizes as possible. However, as shown in Figure 4 (a), this leads to an irregular, staggered appearance, which is unusual for multi-row layouts. In multi-row layouts, the items are usually aligned in a grid, which comes naturally when using common grid-based layout managers. Therefore, we also evaluate some layouts where the items are aligned in a grid, with each item taking up as many cells as seems natural for their given preferred size. More specifically, we want to compare two different solving strategies for the grid-aligned layouts: either the space of the items in the first row uses an equal distribution, as in Figure 4 (b), or it uses a weighted distribution, as in Figure 4 (c). The sizes of the items in the second and third row follow from their alignment with the first row. If the minimal deviation approach were used for the first row, with the items in the second and third row being aligned to the first row, the resulting layout would look very similar to the cases (b) and (c), depending on the layout size,

Question	Score
I easily saw the difference between the layouts.	1.3
It was easy to judge the different layouts.	0.6
I have experience with designing UIs.	1.3
I have graphics design experience.	0.3

Table 1: Average questionnaire ratings on a standard 5-point Likert-scale (-2 to $+2$).

therefore we did not examine this case in separation.

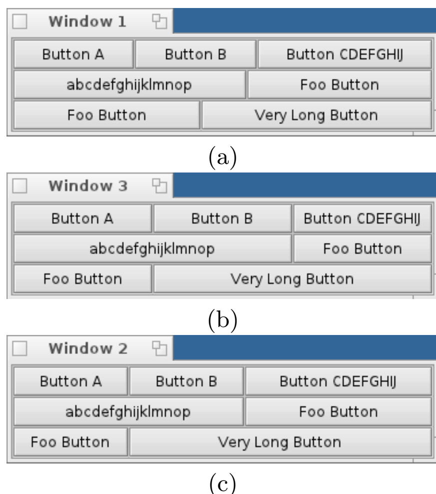


Figure 4: Three row layout at small layout size. (a) Minimal deviation without alignment. (b) Equal distribution with alignment. (c) Weighted distribution with alignment.

6.3 Methodology

Participants were asked to compare various single- and multi-row layouts shown on paper. Each layout was rendered at two different width: a small width close to the minimal width, and a larger width about twice as wide as the small width. The participants were asked to judge the layouts by their visual appearance and rank them accordingly, which is expressed by a score: the worst layout got zero points, the layout in between got one point, and the best layout got two points. The participants were instructed to consider if every button gets a reasonable amount of space for its label. Furthermore, the personal preferences for the button placement and size should be taken into account.

After judging the different layouts, the participants were asked to fill in a questionnaire. The questionnaire used a standard 5-point Likert-scale, followed by open questions asking the participants to describe the criteria for their layout preferences.

6.4 Results

The study had 15 participants. All of them had a Computer Science background, and most of them had experience in designing graphical user interfaces but only casual experience in graphics design. While it was easy for them to see the differences between the given layouts, it was not easy for them to judge them (see Table 1). To determine the significance of the differences in preference, a one-sided Welch t-test is used.

6.4.1 *Significant Preference Differences*

For all the single-row layouts, there is no significant difference between the minimal deviation and the equal distribution layouts. For large layouts, this is expected because minimal deviation and equal distribution layouts look almost identical. For the three-button layout in its large size, the minimal deviation and the equal distribution layouts are significantly ($p < 0.05$) better than the weighted distribution layout.

For the multi-row layouts, the grid-aligned layout is clearly preferred over the unaligned minimal deviation layout. For the two-row layout, the deviation layout gets only 10% for the small size, and 15% for the large size. The scores were even worse for the three-row layout, where only two participant liked the minimal deviation layout.

This is an interesting finding and means a symmetrical layout where the layout items borders are aligned to each other is more pleasant than a layout where each individual item gets the space closest to its preferred space. This can be explained with Gestalt psychology: objects that are aligned to each other are perceived as a group, thus it is easier for us to understand the layout, which makes it preferable [8]. For the usage of constraint-based layout managers like ALM, this indicates it is better to reuse existing tabs to create more alignment in layouts.

6.4.2 *Preference Trends*

Apart form the clear findings of the previous section, some other interesting observations can be made from the taken data. First, a similar tendency as in the multi-row layouts can be seen for the single-row layout in the large size: for large layout sizes, the equal distribution layout is more liked than the weighted distribution layout. When combining the results from the two- and the three-rows layouts, this tendency is significant at the $p < 0.1$ level.

A contrary tendency can be seen at small layout sizes. For the multi-row layouts at small layout sizes, the weighted distribution layout is preferred over the equal distribution layout. When the two- and three-row layout results are combined, this is significant at the $p < 0.1$ level. In the small single-row scenario, where minimal deviation layout and weighted layout look identical, this observation cannot be made. However, at least it could be said that the equal distribution layout does not lead to better results than the weighted distribution layout.

To sum up, there is a tendency that at small layout sizes weighted distribution layouts are more preferred than equal distribution layouts. At large sizes, the equal distribution and minimal deviation layouts are preferred above the weighted distribution approach. This means that the minimal deviation approach, which is equal to the weighted distribution at small layout sizes and very similar to the equal distribution at large layout sizes, is well-suited for all the sizes, small and large. Furthermore, the minimal deviation solution scales smoothly down to small layout sizes where it seems to be important that each item gets a fair amount of the size discrepancy. For large layouts, the minimal deviation approach roughly distributes all layout item equally, which has been found to be the most preferred solution for large layouts.

6.4.3 *Qualitative Responses*

When asked about the criteria for preferring one layout over another, the most frequent answer from the participants was that alignment of the buttons is an important factor (6 participants). For others, enough space for the button labels and the button margins was important (3 participants). Furthermore, three participants stated

that they prefer layouts with equal button size.

These qualitative statements are consistent with the findings from the quantitative layout evaluation. First, aligned multi-row layouts are preferable over unaligned layouts. Secondly, for small layout sizes, there is the trend that minimal deviation layouts, where the size discrepancy is uniformly distributed on the button margin, are more liked. Thirdly, for large layout sizes, each layout item should get the same amount of space, which is the case in equal distribution and minimal deviation layouts.

7. CONCLUSION

Layout managers are a convenient way to arrange items in a layout, independent from the actual layout size. All layout managers need to define a strategy to distribute additional or lacking space, i.e. the discrepancy between the preferred and the actual layout size. Looking at principles such as the Gestalt laws, it is clear that the distribution strategy is likely to affect the aesthetics of a layout. However, our review of existing layout managers shows that there is no agreement on how this is best done.

Using constraints is the most powerful approach for layout management, and all other approaches can be reduced to it. To deal with conflicting constraints such as preferred sizes, constraint solvers have to optimize an objective function. We have identified two issues that affect constraint-based layout managers: a linear objective function can lead to nondeterministic layouts, and spring effects can lead to layout distortions. For the latter one, we have identified a solution that makes sure preferred size constraints are specified for rows and columns rather than for individual items.

In an empirical evaluation, we have investigated the effects of three layout solving strategies – equal distribution, weighted distribution and minimal deviation – on aesthetics. The evaluation shows that while a weighted distribution tends to be preferred at small layout sizes, an equal distribution is preferred at large layout sizes. As a good tradeoff, the minimal deviation approach yields aesthetically pleasing results at small and large layout sizes. Another finding is that users prefer GUI layouts in which the items are aligned over layouts with less alignment – a finding that is consistent with the Gestalt principles.

References

[1] G. J. Badros, A. Borning, and P. J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Comput.-Hum. Interact.*, 8(4):267–306, Dec. 2001.

[2] M. Berkelaar, P. Notebaert, and K. Eikland. lp_solve: (Mixed integer) linear programming problem solver. 2007. http://lpsolve.sourceforge.net.

[3] J. Borchers, O. Deussen, A. Klingert, and C. Knörzer. Layout rules for graphical web documents. *Computers & Graphics*, 20(3):415 – 426, 1996.

[4] M. K. Coleman and D. S. Parker. Aesthetics-based graph layout for human consumption. *Softw. Pract. Exper.*, 26(12):1415–1438, Dec. 1996.

[5] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ, 1963.

[6] R. Fletcher. *Practical methods of optimization; (2nd ed.)*. Wiley-Interscience, 1987.

[7] J. Fogarty and S. E. Hudson. Gadget: a toolkit for optimization-based approaches to interface and display generation. In *UIST*, pages 125–134. ACM, 2003.

[8] S. Heim. *The resonant interface: HCI foundations for interaction design*. Pearson/Addison Wesley, 2007.

[9] D. E. Knuth. *The Computers & Typesetting, Vol. A: The Texbook*. Addison-Wesley, 1986.

[10] W. Köhler. *Gestalt psychology : an introd. to new concepts in modern psychology*. Liveright, 1947.

[11] C. Lutteroth, R. Strandh, and G. Weber. Domain specific High-Level constraints for user interface layout. *Constraints*, 13(3), 2008.

[12] C. Lutteroth and G. Weber. Modular specification of gui layout using constraints. In *Proceedings of the 19th Australian Conference on Software Engineering*, pages 300–309. IEEE Computer Society, 2008.

[13] M. MacDonald. *User interfaces in VB. Net: windows forms and custom control*. Apress Series. Apress, 2002.

[14] M. F. Plass. Optimal pagination techniques for automatic typesetting systems. In *International Symposium on Physical Design*, 1981.

[15] D. S. Weld, C. R. Anderson, P. Domingos, O. Etzioni, K. Gajos, T. A. Lau, and S. A. Wolfman. Automatically personalizing user interfaces. In G. Gottlob and T. Walsh, editors, *IJCAI*, pages 1613–1619. Morgan Kaufmann, 2003.

[16] J. Zukowski. *Java AWT reference*. O'Reilly & Associates, Inc., 1997.