HAL
open science

A New Layout Method for Graphical User Interfaces

Adriano Scoditti, Wolfgang Stuerzlinger

To cite this version:

Adriano Scoditti, Wolfgang Stuerzlinger. A New Layout Method for Graphical User Interfaces. IEEE Symposium on Human Factors and Ergonomics 2009, 2010, Toronto, ON, Canada. pp.642-647, 10.1109/TIC-STH.2009.5444422 . hal-00953333

HAL Id: hal-00953333

https://inria.hal.science/hal-00953333v1

Submitted on 28 Feb 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A New Layout Method for Graphical User Interfaces

Adriano Scoditti
Laboratoire d'Informatique de Grenoble
Université Grenoble I / CNRS, France
http://iihm.imag.fr

Wolfgang Stuerzlinger
Dept. of Computer Science and Engineering
York University, Toronto, Canada
http://www.cse.yorku.ca/ wolfgang

***Abstract*—The layout mechanisms for many GUI toolkits are hard to understand, the associated tools and API's often difficult to use. This work investigates new, easy-to-understand layout mechanisms and evaluates its implementation. We will analyze the requirements for the definition of layouts of a graphical user interface. Part of the issue is that several aspects need to be considered simultaneously while laying-out a component: the alignment with other components as well as its own behaviour while resizing its container. Moreover, the used tools should isolate the designer/drawer from the implementation details of the framework.**

We present the details of our new GUI layout system, discuss the choices we made for our new layout algorithm and detail implementation issues. Moreover, we present also the user interface for our new GUI builder system that contains several innovations, such as a preview window to show the effects of layout configuration choices in real-time. We present an evaluation of our new system by attacking the complex GUI layout problem mentioned above.

I. Introduction

Building GUIs involves several technical issues such as the definition of each components' layout, the minimum (and maximum) sizes of each component, the containment hierarchy, the interaction behavior of each component, the integration with the core software, and the portability of the interface itself both among different projects and different systems. As this work focuses only on the layout problem, the interaction between the GUI and the software is beyond the scope of this document. One central problem in GUIs is to define how components are placed within a window and how the components size changes when the window is resized. This is commonly refereed to as defining the layout of a GUI. Various solutions for the definition of layouts have been proposed, [1][2][3][4]. Most commercial toolkits make one or more of these solutions available to programmers. Using the Java/Swing terminology, some simple examples are the fixed-position layout, grid layout, flow layout, row and column layout, but all of these can only deal with restricted problems. A very flexible and powerful mechanism is the Struts and Springs layout [5] that uses constraints to define the positions of components. This is implemented in the SpringLayout included in the latest Java Virtual Machine. The downside of the Struts and Springs method is that computing the layout requires quite a bit of computational resources. All of the above layout methods aim to make the process transparent for the graphic designer. However, as we will see later, none of them have reached a satisfying abstraction level that permits the graphical designer to ignore implementation problems. Furthermore, the problem of defining user interfaces independently of the implementation in a portable way is addressed by a combination of the MVC (Model-View-Controller) programming pattern and storing the layout in external files, typically using some form of structured file format.

We will analyze the above-mentioned problems in depth both from the point of view of a graphical designer and a programmer. The first one wants to concentrate on the design of the GUI without having to worry about the limitations of the underlying GUI toolkit or the used GUI builder. In other words, wants to define a user interface focusing on the quality of the interface itself without having to address implementation issues. The programmer, on the other hand, strives to achieve a modular software architecture, wants to reuse his code, and aims to provide powerful tools to the interface designer.

II. Resizing behavior

While building a GUI, the most important step is to define the space occupied by each component of the interface itself in the right container . This means that, first of all, we must be able to define the position and the size of each component. However, this raises issue when the container is resized and still it is not sufficient to define all desired visual effects - most importantly various forms of alignment. Alignment of components helps with visual grouping and is consequently considered necessary in graphical design. For example, when positioning a text label next to a text field, normally the baseline of the text label is aligned with the baseline of the text inside it (or it's first line). Or the center of the label is aligned with the center of a larger text box.

A. Evaluation of Previous Struts and Springs Implementations

Two different implementations of the Struts and Springs algorithm exist. The first one, the Swing SpringLayout, uses the classical approach of using constraints to define the layout of each component and then solving the whole set of constraints simultaneously. The second alternative, the Cocoa layout system, is based on an interpretation of the Struts and Springs concept, which utilizes the springs only to determine a viable partition of available space when resizing a component.

Both of the presented implementations have their pros and cons. The Cocoa implementation, for example, is extremely fast although it is not able to define and to lay out complex examples, as demonstrated in the previous chapter. On the other hand the Java SpringLayout is powerful, but solving the constraint-equation system is not very fast. Another important difference between the two implementations concerns the technical definition of which entities struts and springs can constrain to. There are two alternatives here, and the struts and springs can:

1) Connect each component of the GUI only to the edges of the window that contains the component itself.
2) Connect each component to other components present in the window.

Although the first option is simple and fast to implement, it is limited as one cannot define inter-component relationships. The second alternative, as implemented in the Swing SprintLayout, is to permit each component to attach to arbitrary points in the window or to the sides of other components. This guarantees that no overlay can happen, as struts can be defined between two components or even among several components. The downside of this is that many constraints may be needed for complex windows.

The goal of our work is to introduce a fast and simple to use layout algorithm. Our intent is to maintain the advantages of all the algorithms presented here, while avoiding their disadvantages as far as possible. For this, we improved Cocoa's algoritm and added a new interface component.

III. Intui Struts and Springs

It is an extension of the method used in the Cocoa framework. First, we introduce the concept of references, discuss the fundamental algorithm, and then introduce a new interface component, a *Spacer*, to increase the power of the approach.

A. References

To avoid overlap, each component in a GUI needs to keep track of other components around it. At a minimum, each component needs to consider at least one component on each side of it. For this, we introduce the so called *references*. Each component has four references, one for each side, that link it to it's neighbors. If there is no neighbor, it can also link a component to the window itself. Component frequently have multiple neighbors on the same side. To resolve this problem, let's analyze the scenario represented in Figure 1.

In Figure 1 on the left, each box represents a graphical user interface component. The bigger frame represents the window. In this figure, we analyze how references for component *A* are defined. As mentioned above, the method searchers for the closest component relative to the center of each side. The bottom reference of *A* points to *D*, as it is the only component below. The top and left references are pointing to the window frame. On the right side of *A*, *C* will be chosen as the reference, as it is closer to the center of *A*. In this example, there is no reference to *B*, but depending on the references for the remainder of the layout it may not be necessary to have one.

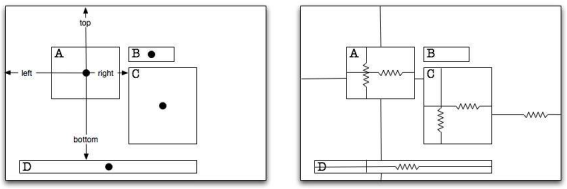

Fig. 1. Each object automatically chooses references by detecting the the nearest component in the four directions from the center of each side. On the right image, the overall configuration of the widget *A* in the main container is highlighted.

IV. Intui Struts and Springs Algorithm

In the Intui Struts and Springs layout method, each component is aware of the components around it due to the references. If a spring is associated with a reference, the algorithm can then better compute proportions relative to other components, which leads to an overall more powerful algorithm. Let's analyze the behavior of the widget *A* in Figure 1, by associating struts and springs to the scenario, as shown in the right part of the Figure. Furthermore, each component has also internal struts and springs, to allow widgets to vary in size depending on their content. While resizing the external frame, a widget *A* can count the number of springs around itself by invoking a recursive method that stops when the container frame is reached. Once the number of struts and springs in a direction is known, the widget can compute accurately how it should resize. One fairly obvious optimization of the algorithm shown above is to count the number of springs in advance for each component, whenever the layout has been finalized, as this removes the recursive search for references. Then the amount of work to be done for each resize is effectively constant, which makes the algorithm a lot faster. However, we use the above version of the algorithm in the real-time preview window of our interface builder (discussed later).

V. Minimum Size

One of the most important attributes for defining a GUI is the definition of the minimum size of the window. To explain minimum size management in our algorithm, we first state our underlying assumption: In a resizable GUI, at least one component will be resizable or have to change its origin when resizing a window. If the GUI is not resizable, this clearly does not apply. Within our framework, we define a default minimum size for each component depending on the component itself and it's normal use. For example, the minimum size of a text label is smaller than the minimum size of a text area. The management of horizontal and vertical resizing is handled independently, so that a component that has reached its vertical minimum size can still change in the horizontal direction. When resizing the window, each component first computes its future size and then compares it with this predefined minimum size to detect a resize that would make it too small. Similarly, each component also checks if the resize would move (even part of it) outside of the window frame and signals that it

cannot resize accordingly in this case. Checking for overlap between components is not done, as this would require too much work on specifying all constraints by the designer. In the next section we will introduce a solution for this problem via spacers.

VI. The Spacer - A Mechanism for Coupling, Alignment, and Minimum Size Control

Figure 2 depicts the common scenario of multiple components inside a window. The figure illustrates multiple issues, namely that the simple idea of having references always use the center of a component can easily fail and the importance of symmetry. In this case there is a vertical symmetry of components *B* and *C* relative to the vertical midpoint of component *A*. The Intui algorithm as described above cannot correctly deal with this example.

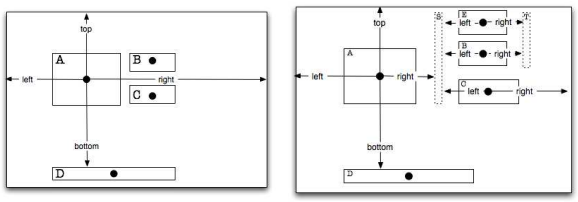

Fig. 2. On the left: Illustration of a failure case for the simple Intui algorithm. Due to it's use of midpoints for references, it cannot identify the two components to it's right and hence incorrectly links component *A* to the window frame. On the right: Spacers can be used to align components. Here the right sides of components *B* and *E* are aligned via spacer *T*. The horizontal extent of the spacer is exaggerated for better illustration.

Our solution to this problem is to introduce a new GUI component, that exists solely for the purposes of layout: a *spacer*. This is basically an invisible component, which has all the normal properties such as a position and a size. This size can either be constant or variable in each dimension. This spacer can assume different roles in the definition of a layout: it couples the layout of components, can be used to align components, or controls the minimum size of a window.

To couple components, a spacer is placed between them. Then the references for the components on either side reference the spacer, which effectively groups their influence and couples their relative layout. Figure 2 illustrates this, by placing a spacer *S* between components *A*, *B*, and *C*. The midpoint reference for *A* now references the spacer *S*, and both *B* and *C* reference it as well. This effectively couples their layout so that the three components never overlap. In the figure the horizontal extent of the spacer has been exaggerated for better illustration, typically this spacer would have a horizontal extent of a single pixel (and a variable vertical extent).

We illustrate in figure 2 how spacers can be used to align components. Here the two spacers *S* and *T*. Spacer *S* aligns the left sides of the three components relative to *A*, wherease *T* aligns the right sides of the two (resizable) components *B* and *E*. This example is an extension of the coupling property explained above.

As presented in section V, our algorithm automatically manages the minimum size of a window, based on the properties of the components. However, sometimes a GUI designer may need to define a minimum size for a window, where this size is larger than the sum of the components at their smallest possible size.

Figure 3 illustrates how spacers can be used to achieve this, namely by adding a spacer *S* between the other two components. The vertical size of *S* is then set to have either a constant size or to never go below a certain minimum vertical extent. If it is constant, the window is effectively non-resizable, if the other two components have fixed size, too. If it has a minimum, this minimum will always guarantee a proportional spatial separation between components *A* and *B*.

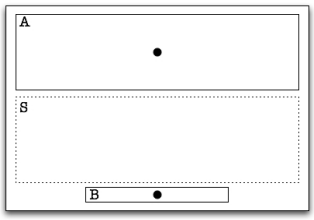

Fig. 3. A spacer can also be used to control the minimum size of a layout, see text.

A. Relation With Previous Work

When configured as a simple non-resizable rectangle, the spacer as introduced here is similar to the concept of a (one-dimensional) *Gap* in the Swing BorderLayout. However, a spacer can also occupy space in both dimensions simultaneously. The concept of *Docking* in the Windows Presentation Framework is also similar. We can simply compare the WPF anchor property to the strut concept. Furthermore, the docking property can be simply seen as a null size strut able to keep constant the distance of an edge of a component from another edge of its container.

The spacer also fulfills the role of an aligner, similar to the concept of alignment in the Windows Presentation Framework, but while WPF resolves all the alignment problem by using the Grid Layout in the appropriate way, the Intui aligner can be seen as a real object added to the interface to align other component. In this way, there is no difference between what the user wants to do and how the algorithm implements it.

In summary, the novel idea of a spacer subsumes a variety of layout concepts that have been introduced in other systems. However, spacers also extend these concepts in several ways.

VII. Hierarchical Intui Layout

As presented above, our new variant of the struts and springs algorithm can cover many common cases. However, in some cases it may be unable to defined complex interdependencies correctly. As an example, consider the vertical resize behavior of the layout shown in figure 4. In this scenario, the intent of the designer is to preserve the "cross" shape in

both dimensions. However, the Intui algorithm simply counts the number of springs along each dimension and distributes space proportionally. Hence, all components will be uniformly resized by a third of the dimension change and the "cross" shape will not be preserved at different window sizes.

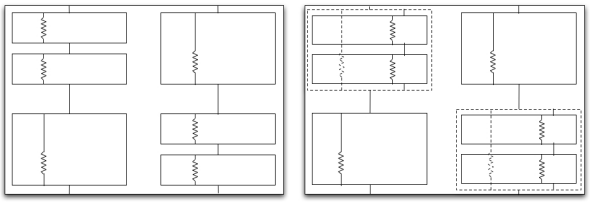

Fig. 4. With layers the algorithm can preserve the distribution of the blocks.

Although modifications of the Intui algorithm may guarantee a more intelligent distribution of the available space among the components, it also means introducing additional parameters, which would make the framework more complex and less intuitive. To address this situation in a simpler manner, we introduce another component, the *container*, which is identical to the hierarchical containers available in many other GUI toolkits. Then one can simply wrap each pair of smaller frames together into a higher level component. Then the bigger frames and the containers will share the same space, which will preserve the alignment of the overall layout. See figure 4 for an illustration. Although containers are often seen as special purpose components, they also serve to define a decomposition of space in a layout. While one can use Spacers for similar purposes in some situations, containers are more powerful. Containers can "encapsulate" several components into a new entity, which effectively allows to synthesize complex resize behaviors in a bottom-up fashion. In other words, the designer can specify different resizing behaviors at different levels of the layout hierarchy to achieve his/her design goal. For simplicity, we implemented hierarchical layout by designating a spacer as a container, i.e. allowing the placement of components inside a spacer. In other words, the spacer introduced above is also our mechanism for creating hierarchical layouts. In terms of efficiency containers can be considered a trade-off. Hierarchical containers offer a way to modularize a layout, but take a bit more time as the traversal of the hierarchy will be slower than the constant-time resize algorithm of the non-hiearchical Intui layout algorithm.

VIII. Real Time Preview

Almost all current GUI builders permit the user test the constructed GUI at the press of a button. Although this feature lets the user identify any problems with the layout in this mode, they can fix these problems only after they have exited this test mode. In other words, the test view is not visible when the construction mode is active.

While this idea has the advantage that it allows the user to focus on the final look exclusively, it hinders the debugging and correction process for any problem found as the user can't simultaneously see the definition of the interface and the "live" version that can be resized interactively. Hence, the user is often obliged to switch repeatedly between the building environment and the testing environment. Expert users may need less iterations here, but could still benefit from an improved solution.

Our answer to this problem is a real time preview that is visible *during* the construction process and can be resized at any time. To illustrate the impact of a simultaneous preview in an interface builder, we refer the reader to figure 5, which depicts a screenshot of our interface builder.

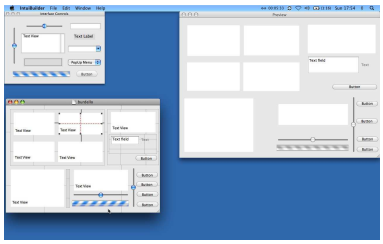

Fig. 5. The Intui Interface builder features a real-time preview of the constructed interface, shown in a different size compared to the construction window.

In the top right is a small panel with all the components available for selection and/or drag and drop. The bottom left shows the construction window and the right side shows the preview. The GUI construction window contains abstract widgets that permit modification of the resize behavior, e.g. see the selected widget in the top-center. The preview window is a fully functional GUI, and the user can enter text, click on buttons, etc.

The two views of the GUI let the user directly compare the positions and the sizes of each component at different window sizes. This enhances the user experience with direct feedback for any change in the construction window [1]. Another way to express this is to say that we provide a "What You See Is What You Get" (WYSIWYG) interface.

IX. Defining the Resize Behavior

The graphical metaphor for the resizing bit-mask is represented by solid and dashed lines in the construction window. To ensure adequate visibility, the lines illustrating the resizing behavior are only shown for the currently focused component, struts are visualized by black solid lines and springs are visualized by red dashed lines, both of them are drawn on a grey background. When a new component is dropped or otherwise instantiated in the construction window, the initial resize mask is calculated by a heuristic algorithm that uses the following information:

1) Its relative position within it's container.
2) Its size,independently width and height.

[1] In an early prototype, we let the user also modify the layout in the preview window. However, we removed this feature because it introduced an ambiguity, as the references for a component might be different at different sizes due to different position of the midpoint. Hence we dropped this feature.

3) The resize masks of all components it references (i.e. the masks of all components around it).

The relative position within the container is used as follows: if the center of the component is within the first 25% or the last 25% of each dimension, struts are placed towards the outside. Otherwise, the component is in the center part of the window and springs are used. The mask to define the intrinsic size of the component is determined to be fixed depending on the type of the component (buttons are fixed, text-areas resize). The behavior of adjacent components influences the initial value for the resize mask as the following example: if a component is placed in the center of the window but it is surrounded by component with only external struts, it will be configured with external struts as well. The same algorithm also automatically determines the references between the components by shooting a ray starting from the midpoint of each side and identifying the closest component along the ray (unless the window frame is hit). To avoid potentially unstable situations, the algorithm checks that two components that reference each other has the same value from both side of the ray. If there is a disagreement, the configuration of the component where the algorithm is running (i.e. the focused component) will change the configuration to create the needed coherence (in this way, the choices of the user are always respected because as we will see later, it is possible that on the other component the user manually modified the configuration). The above algorithm is also used to determine the new resize mask every time the user moves a component (unless the resize mask is manually overridden, see below). The net result is that the construction process is greatly accelerated, as the above algorithm frequently sets the desired resize mask for a component correctly - at least to our experience. To aid the user in understanding the behavior of the algorithm, the background of each container shows a grid of the decision boundaries used for the position mask calculation in medium grey. Although this heuristic algorithm can forecast the intention of the user in most cases for simple user interfaces, it clearly cannot deal with arbitrary layouts. Hence, the user interface of the GUI builder must allow the user to change the default configuration for the resize behavior. The visualization of the struts and springs in the construction window is fully interactive. In other words, the user can modify the resize mask by clicking on solid and dashed lines depicting the struts and springs. Clicking on a strut changes it to a spring and vice versa. Every time the resize mask is modified in the construction window, the preview window updates the layout of all components to immediately visualize the consequences of user actions.

To avoid potentially unstable situations due to conflicting relative resize masks, the GUI builder checks that two components that reference each other set the resize mask to the same value from both sides. An unstable situation can never happen because, as we saw before, the needed coherence is always created by the automatic algorithm while adding the component. This means that while manually modifying an external strut for a component, for example, this modification is automatically propagated towards the other component that this strut references.

X. Evaluation

As an evaluation of the Intui GUI builder, we defined a complex user interface. There are several ways to achieve this, but we choose the alternative that emulates the behavior of an irregular grid. Figure 6 depicts the Intui builder with the final result and a bigger preview that illustrates the correct resize behaviour.

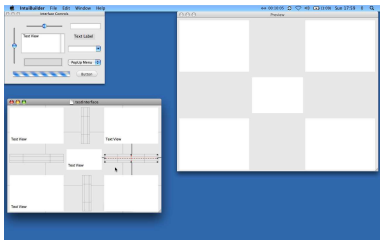

Fig. 6. Screenshot of IntuiBuilder while constructing the evaulation user interface.

The construction process for the test user interface is fairly simple. Once all five text boxes are dropped into the correct positions in the construction window, the following steps need to be carried out:

1) The heuristic resize mask algorithm has already set the behavior for all four corner boxes. Hence, no action is needed on them.
2) The resizing behavior for the central box needs to be changed by adjusting the internal behavior.
3) To avoid potential overlap, four spacers need to be added and their internal resize behavior adjusted by making them resizable along their shorter dimension only, because the longer one is well forecasted.

XI. Hierarchical GUI Construction

In the last chapter we discussed the concept of a container hierarchy for GUI widgets. Here we mention how we allow the user to place components inside spacers to use them as containers.

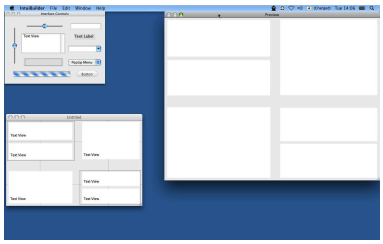

Fig. 7. Screenshot of IntuiBuilder for the hierarchical layout.

An explanatory example is shown in figure 7, where we show how the layout discussed in section X can be constructed with a hierarchy. As visible in the figure, each pair of smaller

text boxes is placed within a container. Then the designer can define the layout of the four top level components to preserve the overall layout with the cross in the the middle. The smaller text boxes resize within each container and are internally resizable, but have fixed distances externally. When resizing, the space will first be distributed among the four high-level components, and then each pair of smaller text boxes split the space within their container evenly.

XII. Conclusion and Future Work

We introduced a new way to define the layout of graphical user interfaces. The new Intui layout algorithm implements a conceptually simple variant of the struts and springs model, that is powerful enough to address many common layout problems. For this, we introduced the novel concept of a *Spacer*, which serves multiple purposes: It couples components, aligns them, can be used to intuitively define the minimum size of a window, and can be used as a container. Furthermore, the computational cost for resizing Intui layouts is far less costly than with a full implementation of a constraint-based solver. We also introduced a novel construction interface for graphical user interface builders, which makes the construction of common user interfaces easier to perform. This is achieved in part by a heuristic algorithm that automates common component placement decisions. Furthermore, we introduced the concept of a real-time preview window, which is sized to immediately visualize how the current layout looks at a different size.

Some of our design choices are arguably non-optimal and may be improved. One such issue is an improved discovery method for references. The current approach automatically finds references based on the midpoint of each side of a widget. This permitted us to ensure coherency and generality in the builder. Our intent was to enable the user to immediately see the dependencies when looking at a widget in the construction window. Furthermore, this design decision kept the file format simple. On the other hand, a user may not immediately understand the way this definition of references works. Furthermore, the references for a widget change whenever it is moved in the construction window, sometimes with results that are not immediately predictable.

Another issue is that the GUI builder does not snap to common positions when a component is moved or resized. Examples are the center of the window or positions aligned to another widget. This functionality may be added in the future. Furthermore, common guidelines, such as a default margin, should also be introduced as snapping points.

References

[1] S. Chatty, S. Sire, J.-L. Vinot, P. Lecoanet, A. Lemort, and C. Mertz, "Revisiting visual interface programming: creating gui tools for designers and programmers," in *UIST '04: Proceedings of the 17th annual ACM symposium on User interface software and technology*. New York, NY, USA: ACM, 2004, pp. 267–276.

[2] S. E. Hudson and S. P. Mohamed, "Interactive specification of flexible user interface displays," *ACM Trans. Inf. Syst.*, vol. 8, no. 3, pp. 269–288, 1990.

[3] K. Miyashita, S. Matsuoka, S. Takahashi, A. Yonezawa, and T. Kamada, "Declarative programming of graphical interfaces by visual examples," in *UIST '92: Proceedings of the 5th annual ACM symposium on User interface software and technology*. New York, NY, USA: ACM, 1992, pp. 107–116.

[4] P. Dragicevic, S. Chatty, D. Thevenin, and J.-L. Vinot, "Artistic resizing: a technique for rich scale-sensitive vector graphics," in *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*. New York, NY, USA: ACM, 2006, p. 6.

[5] L. Cardelli, "Building user interfaces by direct manipulation," in *UIST '88: Proceedings of the 1st annual ACM SIGGRAPH symposium on User Interface Software*. New York, NY, USA: ACM, 1988, pp. 152–166.

[6] S. E. Hudson and K. Tanaka, "Providing visually rich resizable images for user interface components," in *UIST '00: Proceedings of the 13th annual ACM symposium on User interface software and technology*. New York, NY, USA: ACM, 2000, pp. 227–235.

[7] D. Kurlander and S. Feiner, "Inferring constraints from multiple snapshots," *ACM Trans. Graph.*, vol. 12, no. 4, pp. 277–304, 1993.

[8] M. A. Linton, J. M. Vlissides, and P. R. Calder, "Composing user interfaces with interviews," Stanford, CA, USA, Tech. Rep., 1988.

[9] D. E. Knuth, *The TeXbook*. Addison-Wesley Professional, 1986.

[10] J. Dan R. Olsen, *Developping User Interfaces*. Morgan Kaufmann Publishers, 1998.

[11] Microsoft, "Microsoft developer network." World Wide Web electronic publication, 2007. [Online]. Available: http://msdn2.microsoft.com

[12] A. Inc., "Apple developer connection," World Wide Web electronic publication, 2007. [Online]. Available: http://developer.apple.com

[13] S. Microsistem, "Java api specification," World Wide Web electronic publication, 2007. [Online]. Available: http://java.sun.vom/j2se/1.5.0/docs/api/