AN EFFICIENT WINDOW SYSTEM BASED ON CONSTRAINTS

Danny Epstein and Wilf R. LaLonde

SCS-TR-131, MARCH 1988

School of Computer Science, Carleton University
Ottawa, Canada, KIS 5B6

An Efficient Window System Based On Constraints

Danny Epstein and Wilf R. LaLonde

School of Computer Science
Carleton University
Ottawa, Ontario, Canada K1S 5B6

Abstract We describe the design of an efficient constraint-based window system for Smalltalk. This window system permits the specification of windows by constraining the subwindows to satisfy relationships from a very small number of primitive prioritized constraints. In fact, only three classes of constraints are supported, one of which is implicit and unavailable for general use. The system extends the current Smalltalk system by eliminating some of its existing inadequacies and by permitting the integration of both fixed-sized and scalable subwindows in the same window. A goal of the design is to produce a system with real-time response that is fast enough to be substituted for the existing design. A prototype with response times of less than 1/4 a second has been implemented to demonstrate the notions; alternative constraint satisfaction solution techniques are being experimented with to speed it up even further.

1 Introduction

The window system in Smalltalk is weak, difficult to use, and counter-intuitive. One way to extend it is to base it on the notion of constraints as in Borning and Duisberg [3], Borning et al [2], and Sussman and Steele [8]. In this paradigm, a user describes a window configuration in terms of **prioritized constraints**; i.e., boolean relationships that must be satisfied according to some priority scheme. Constraints provide an intuitive means of describing the windows and the relationships between them. Priorities permit less important constraints to be violated when more important higher priority constraints need to be satisfied. They also permit default values to be represented as low priority constraints.

The existing window system does not support windows which are fixed-size or windows whose coordinate systems are fixed-scale. When a window is resized in Smalltalk, subwindows are always resized and rescaled. The new window system permits both the size and the scale of a window to be constrained. Fixed-size windows would be useful for buttons that consist of forms, for example. Currently, forms cannot be used because resizing a form usually results in an unacceptable picture. Fixed-scaled windows would be useful for paragraph editors since fonts should not change size when the window is resized. This requirement is currently handled by paragraph editors with ad hoc techniques. Constraints can also be used to align windows in various ways. Initial constraints can be used to specify the original layout of a window but a user can move or resize it even if it happens to be a subwindow. This provides him with the capability to reorganize the layout of windows as he wishes. For example, a designer could implement a browser which initially had equal-size subwindows. The user could then resize one subwindow and the others would change accordingly.

Unlike more general-purpose systems like ThingLab [1] and its extension [2], our constraint window system (CWS) is designed only for use as part of an upgraded window system. Consequently, it is integrated with the system. It is also considerably simpler. In particular, two user-level constraints (anchors and equality constraints) are sufficient to provide the functionality mentioned above; a third implicit constraint actually provides the backbone of the system. Like the Borning [1] system, our CWS prototype compiles and caches a constraint solution. However, the current prototype has a relatively simplistic solver. A more efficient solver will be the goal of the second prototype. Nevertheless, resizing typical windows in the current system generally takes under 1/4 second on a Mac II running ParcPlace Systems Smalltalk. Our goal is to develop a system that is fast enough to be accepted by typical users.

In our system, constraints are relationships between **constraint variables**. There are only four kinds of constraint variables: numbers, points, rectangles, and transformations. Points, rectangles, and transformations are **container variables** since they contain other variables. The parts of container variables can be independently constrained. Additionally, there are only three types of **constraints**: anchors, equality constraints, and window constraints. **Anchors** force a variable to keep its current value or to have a specific constant value. **Equality constraints** force two variables to have the same value. **Window constraints** interrelate various aspects of a window, such as the window's viewport, local transformation, display box, and display transformation. Only the first two classes of constraints are used for specifying constraint relationships. The latter is associated with windows and used implicitly by the system. Note that we could easily add new classes of constraints like **add** and **multiply** if we wanted to. On the other hand, the anchor and equality constraints were sufficient to exceed the existing capabilities of the system.

2 The Smalltalk Window System and Some of its Inadequacies

The existing window system in Smalltalk is based on the model, view, controller (MVC) paradigm. Each window has a **model** which is the object being viewed, a **view** which is concerned with displaying it, and a **controller** which handles the user interaction with it. There can be several views onto the same model. Views can be constructed in a hierarchy. Top views can be collapsed, moved, framed, or labelled via the blue button or by clicking the label box (depending on the implementation). Subviews have no such operations.

Each view has its own coordinate system and a border of arbitrary thickness. The **viewport rectangle** for a view is the rectangle surrounding the outside of the border in the coordinate system of the superview. The **window rectangle** for a view is the same rectangle in the coordinate system of the view itself. The term window here is used in the technical sense as opposed to the more colloquial sense used, for example, in 'window system'. These two rectangles together specify the view's **local transformation** which is usually abbreviated **transformation**. The **display rectangle** is the viewport rectangle in the coordinate system of the screen. The **inset display rectangle** is the display rectangle without the border. The **display transformation** is the transformation that maps the window rectangle into the display rectangle; it is obtained by composing the successive local transformations from the subview to the top view in the hierarchy.

One difficulty with these definitions is that transformations are in terms of window, viewport, and display rectangles rather than corresponding inset window, inset viewport, and inset display rectangles. The border width is not transformed. If the user designs his application so that the visual data fits in the inset window, there are potential problems. If the top window is a little too small, some of the data is clipped by the border (since the border didn't shrink). Conversely, if the top window is a little too large, extra unused space is evident (since the border didn't expand). Compensating for this is quite difficult. We'll refer to this problem as **window roundoff**. An example is shown below.

Example

A standard system view with a non-editable display text subview could be created and scheduled as follows. Such a display text subview would typically be used with other views that have more active controllers.

```
| topView aDisplayTextView |
aDisplayTextView ← DisplayTextView new
        model: 'Seek the highest mountain\and you shall be peaked!' withCRs;
        controller: NoController new;
        borderWidth: 1;
        insideColor: Form white;
        centered. "omit this if centering is not wanted"
```

```
topView ← StandardSystemView new label: 'Non-editable Text Window'; borderWidth: 1.
topView addSubView: aDisplayTextView. topView controller open
```

If the standard system view is opened as a small rectangle (as opposed to one that is much larger than required to contain the textual data), the first window (see Figure 1) would be displayed. The second results if a large window is used.

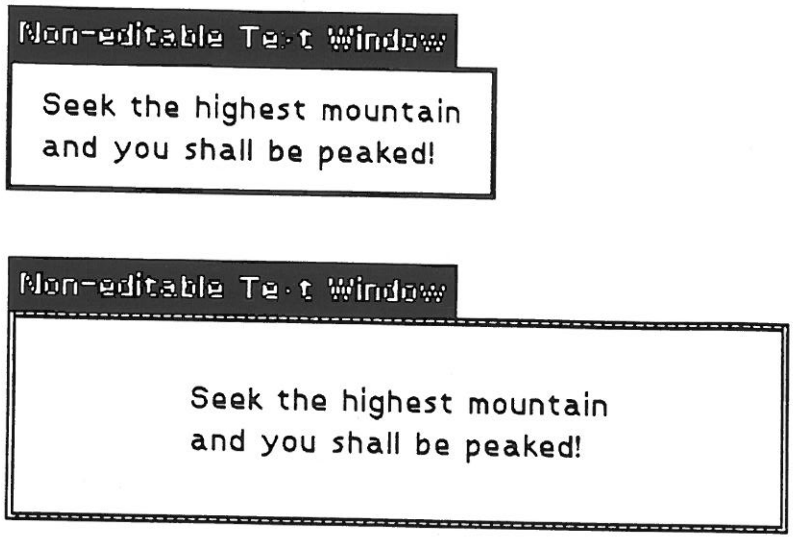


Figure 1: A DiplayText Subwindow

As you can see, if the subview is too large for the text, there is a tendency for the subview border to detach itself from the superview border; i.e., one can see a ring between the two borders. The color of the ring is the inside color of the top view. This can be eliminated in more polished applications by removing the display text view border entirely and enclosing it within a traditional view with the original's border size and inside color. The display text view border is still inset but it is no longer visible.

Top views in Smalltalk can overlap but support for overlapping subviews is inadequate. Views need not be completely visible within their superview but the border of a view is always forced to be visible. The display rectangle and inset display rectangle are actually defined to refer only to the visible parts of the view. There is support for scrolling of views but the scroll amount must be specified in local coordinates rather than display coordinates. During scrolling, no attempt is made to "shift" the existing bitmap and compute only the newly visible data. Scrolling a window with something over it doesn't take into account the covering object.

In Smalltalk, designers normally fix the size of the window rectangle (if they don't, a specific default is used). One might also wish to fix the size of the display rectangle or the value of the scaling factor for the display transformation. We'll refer to these kinds of restrictions as **fixed-window**, **fixed-size**, and **fixed-scale** restrictions respectively. All views in Smalltalk are fixed-window views. The contents of a fixed-window view are transformed by a different amount when the window is resized. Hence the same data fills the view before and after the resize operation (subject to window roundoff); i.e., the view's window rectangle is fixed but the view's display rectangle is varying size. There is no support for fixed-size or fixed-scale views. As mentioned in the introduction, fixed-size views are useful for buttons that display a fixed picture and/or piece of text. Fixed-scale views are also useful for displaying a piece of a large map since the amount of stretch or shrink is pre-specified. Resizing a fixed-scale view, however, does permit a larger (or smaller) amount to be viewed.

We also differentiate between two kinds of fixed-scale views: **unpinned fixed-scale** views and **pinned fixed-scale** views. The contents of an unpinned fixed-scale view do not stretch or move with respect to the screen. When such a view is moved, however, it becomes a view onto a different area of its contents. The contents of a pinned fixed-scale view do not stretch or move with respect to the pinned point. If the pinned point is the top left corner, for example, the same part of the contents will be visible at the top left corner of the view no matter what the new size of the view becomes. In this case, however, the bottom right corner can be anywhere (in the middle or even not visible). The situation could be reversed if the bottom right corner were chosen to be the pinned point instead.

There is no support for displaying objects on views. Most view classes access the inset display box (rectangle) of the view to determine where to display on the screen and what clipping to perform and they often ignore transformation details. This doesn't work very well when a view is scrolled; it is necessary to use the display transformation of the view. Additionally, since a view is an object, it is reasonable to expect to be able to display some other object on it; one would think it would be a display medium of some sort but it isn't. As is, views keep track of their display box but there is no form containing the displayed information. Displaying is done directly to the screen.

There have been several local attempts at providing a better window system in Smalltalk. Rod Nickel [7] merged views and controllers into a single object. Doug Macklem [6] changed the way views resize to allow fixed-scale views and something like fixed-size views to be implemented. Peter Epstein [4] attempted to fix scrolling so that borders could be hidden and so that updates could be done more efficiently. Experiments by LaLonde [5] on pinned fixed-size windows were also performed.

3 The Constraint Window System

The Constraint Window System (CWS) is a hierarchical window system that combines the notions of views and controllers into a single object called a **window**. This new term is more intuitive to novices and also reduces the total number of classes in the system. As in the more familiar system, windows are either top-windows or they have a super-window which contains them as a subwindow. Logically, top-windows have the screen as their super-window. In our first prototype, only top-windows are allowed to overlap and subwindows must be fully visible within their super-windows. Each top-window maintains a system of constraints, to be called a **constraint package**. The constraint package is used to determine the effects of a resize or move operation on a window.

There are some other important differences with the standard system: (1) ***all window components (color excluded) are constraint variables*** and (2) ***a window is a display medium that automatically handles its own clipping***. The first feature is quite important: all protocol that would normally have returned points, rectangles, or transformations under the standard window system now return constraint variables. For example, the display box for a window is a variable; so is the viewport rectangle, the display transformation, etc. A more traditional object can be obtained by asking the variable for its **value**. The intent is for all user interactions to be in terms of variables; this makes sense when we take into account the fact that users should only need to instantiate and configure existing classes of windows by constraining the parts. Only implementers of new classes of windows should need to explicitly ask for the variables' values. The second feature adds to the power and flexibility of the traditional system. For this to be feasible, class Object is extended by adding the following method.

```
displayOn: aDisplayMedium
        ↑self printString asDisplayText displayOn: aDisplayMedium
```

Consequently, all models understand the **displayOn:** message; messages such as 'model **displayOn:** self' are generally sent by the window. For most of the models in use, this default behavior is already specialized; e.g., consider paragraphs, forms, and display text. Recall that each window has its own coordinate system which is used when displaying objects on it.

The definition of the window rectangle is changed to refer to the inside of the border. The viewport and display rectangles, however, are left unchanged; i.e., they refer to the outside of the border. This simplifies matters considerably for the user. It also eliminates the window roundoff problem mentioned above.

As suggested above, there are four classes of constraint variables: NumberVariable, PointVariable, RectangleVariable, and TransformationVariable. An instance of NumberVariable contains a number. It also has a priority associated with it; this priority is set to **unknown** to indicate that the value is unspecified. The other three types of variables are **containers** for other variables. They have no values or priorities of their own although their components do. The types of containers and the names and types of their components are as follows:

Container	**Components**	**Type**
PointVariable	x, y	NumberVariable
RectangleVariable	top, bottom, left, right	NumberVariable
	horizontalMiddle, verticalMiddle	NumberVariable
	height, width	NumberVariable
	topLeft, topRight, bottomLeft, bottomRight topMiddle, bottomMiddle,	PointVariable PointVariable
	middleLeft, middleRight, middle, size	PointVariable
TransformationVariable	scale, offset	PointVariable

Table 1: Container Variables

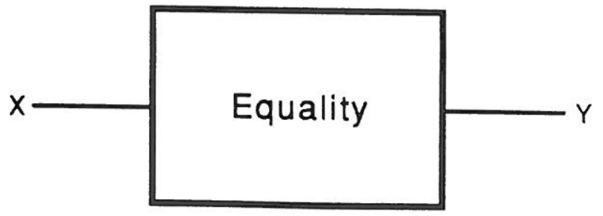
Note that some components of a RectangleVariable contain other components of the same rectangle; e.g., aRectangle.TopLeft.Y == aRectangle.Top. There is only one actual variable. Also note that rectangle variables implicitly constrain their components as follows:

height = bottom - top
width = right - left
verticalMiddle = (top + bottom) / 2
horizontalMiddle = (left + right) / 2

Anchor constraints, equality constraints, and implicit window constraints satisfy the following restrictions:

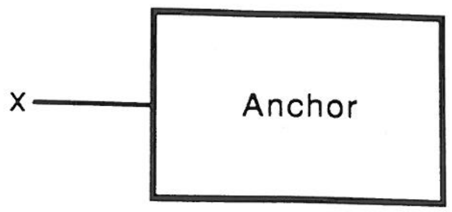
Equality (X, Y):

Forces X and Y to have the same value at a given priority. X and Y may be any kind of variable but they must be of the same type. If they are containers, this is equivalent to several equality constraints on the corresponding components of the containers.



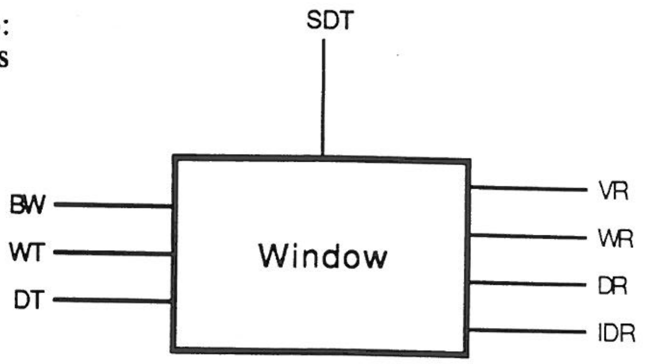
Anchor (X):

Forces X to a given value at a given priority. X may be any kind of variable. If it is a container, this is equivalent to several anchors on the components. If desired, an anchor may force its variable to its current value rather than a specific value. This kind of anchor forces the variable not to change at a given priority.



Window (BW, DR, IDR, WR, VR, WT, DT, SDT):

Forces the given variables to be interrelated as discussed below.



The abbreviations above denote the following:

Instance Variable	Abbreviation	Type
borderWidth	BW	RectangleVariable
displayRectangle	DR	RectangleVariable
insetDisplayRectangle	IDR	RectangleVariable
windowRectangle	WR	RectangleVariable
viewportRectangle	VR	RectangleVariable
windowTransformation	WT	TransformationVariable
displayTransformation	DT	TransformationVariable

Table 2: Window Instance Variables

The border width is a rectangle so that the various edges of the border can have different thicknesses.

Additional Information	Abbreviation	Type
superDisplayTransformation	SDT	TransformationVariable

Table 3: Implicit Window Instance Variables

The super display transformation is either the display transformation of the super-window or the identity transformation if the window is a top-window.

Each window has a window constraint that enforces the following relationships on its variables:

insetDisplayRectangle	= displayRectangle **insetBy:** borderWidth.
insetDisplayRectangle	= displayTransformation **applyTo:** windowRectangle.
displayRectangle	= superDisplayTransformation **applyTo:** viewportRectangle.
displayTransformation	= superDisplayTransformation **compose:** windowTransformation.

Note that the user never explicitly specifies the kind of window restrictions he wants. Instead, he describes fixed-window, fixed-scale, and fixed-size windows using simple anchor constraints as follows:

Fixed-Window:
Anchor the window rectangle.

Fixed-Scale and Unpinned:
Anchor the display transformation.

Fixed-Scale and Pinned:
Anchor the display transformation scale and any point on the window rectangle (top middle, for example).

Fixed-Size:
Anchor the display rectangle size (or the inset display rectangle size).

Subwindows can be aligned in various ways as follows:

Align tops:
Equate the tops of the viewport rectangles (or the display rectangles).

Align bottoms:
Equate the bottoms of the viewport rectangles (or the display rectangles).

Align left sides:
Equate the lefts of the viewport rectangles (or the display rectangles).

Align right sides:
Equate the rights of the viewport rectangles (or the display rectangles).

Align top/bottom centers:
Equate the vertical middles of the viewport rectangles (or display rectangles).

Align left/right centers:
Equate the horizontal middles of the viewport rectangles (or display rectangles).

To create a window, only two methods are provided. As in the standard system, a model is associated with a window in the usual way.

To create a top-window:	Window **new**.
To create a subwindow:	Window **newIn:** aWindow.
To associate a model:	aWindow **model:** anObject

A window displays its model by sending it the "**displayOn:** self" message. Windows are display mediums so clipping need not be handled by the model. Class Window ignores mouse and keyboard events but does display the model. Subclasses of Window provide the more complex mouse and keyboard protocol for specialized windows.

Once a window hierarchy is created, the windows in it can be constrained using the following methods. **Initial** constraints apply when the window is opened the first time; the other **permanent** constraints apply both initially and after changes, like moving or resizing, are made. Anchor constraints force a variable to remain at its current value or to have a particular value. Equality constraints force two variables to have the same value. Permanent anchor constraints need not specify a value if it can be determined from the initial constraints. Each constraint is specified at a given priority. Higher priority constraints override lower priority ones. Low priority constraints can be used to specify default values.

Permanent anchor to a specific value:
aVariable **anchorTo:** aValue **withPriority:** aPriority.

Permanent anchor to current value:
aVariable **anchorToCurrentValueWithPriority:** aPriority.

Permanent anchor with value computed using initial constraints:
aVariable **anchorWithPriority:** aPriority.

Permanent equality:
aVariable **equalTo:** anotherVariable **withPriority:** aPriority.

Initial anchor to a value:
aVariable **initialAnchorTo:** aValue **withPriority:** aPriority.

Initial equality:
aVariable **initialEqualTo:** anotherVariable **withPriority:** aPriority.

The variables referenced when these methods are used are typically components of windows. They can be retrieved with the following messages. Note that they are slight modifications

aWindow **borderWidth.**	returns a rectangle variable
aWindow **windowRectangle.**	returns a rectangle variable
aWindow **viewportRectangle.**	returns a rectangle variable
aWindow **displayRectangle.**	returns a rectangle variable
aWindow **insetDisplayRectangle.**	returns a rectangle variable
aWindow **windowTransformation.**	returns a transformation variable
aWindow **displayTransformation.**	returns a transformation variable

As mentioned above, the border width is a rectangle so that the various edges of the border can have different thicknesses. The components of constraint variables can be accessed using the following obvious methods:

aPointVariable **x.**	{and **y**}
aRectangleVariable **top.**	{and **bottom, left, right**}
aRectangleVariable **height.**	{and **width**}
aRectangleVariable **verticalMiddle.**	{and **horizontalMiddle**}
aRectangleVariable **topLeft.**	{and **topRight, bottomLeft, bottomRight**}
aRectangleVariable **topMiddle**	{and **bottomMiddle, middleLeft, middleRight**}
aRectangleVariable **middle.**	
aRectangleVariable **size.**	
aTransformationVariable **scale.**	{and **offset**}

Priorities are integers between 0 and 12; 0 is the initial priority for all variables (referenced as **UnknownPriority**); 11 is the priority used by the system for anchoring the display rectangle to the resize rectangle when a user resizes the window (referenced as **UserPriority**); 12 represent the **highest** priority - where constraints must be satisfied without exception (referenced as **HighestPriority**).

4 Examples To Illustrate The System

The user describes the desired window configuration by defining a window hierarchy and introducing constraints on the windows in the hierarchy. These constraints may be either initial or permanent constraints.

Example 1: A Simple Example

Consider creating a window like the following with a top-window **a** (inside color white) and two subwindows **b** (gray) and **c** (light gray). Each has a border width of 1 all around. Consequently, adjacent borders have a total width of 2.

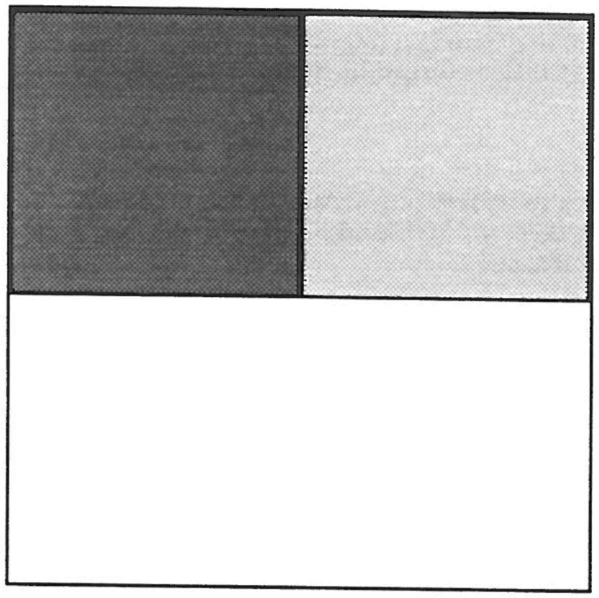


Figure 2: An Example Window

We constrain **a** to be of the fixed-window variety, **b** and **c** to respectively occupy the top left and right quarter of **a** initially. Additionally, the bottoms of **b** and **c** are permanently aligned and the top left corner of **b** is permanently anchored to the top left corner of **a**.

```
"A Standard Example"

(a ← Window new) borderColor: Form black; insideColor: Form white.
(b ← Window newIn: a) borderColor: Form black; insideColor: Form gray.
(c ← Window newIn: a) borderColor: Form black; insideColor: Form lightGray.

border ← Rectangle left: 1 right: 1 top: 1 bottom: 1.
a borderWidth anchorTo: border withPriority: HighestPriority.
b borderWidth anchorTo: border withPriority: HighestPriority.
c borderWidth anchorTo: border withPriority: HighestPriority.
```

```
"The window rectangles can be anything we want."
basicRectangle ← 0@0 extent: 100@100.
a windowRectangle anchorTo: basicRectangle withPriority: HighestPriority.
b windowRectangle anchorTo: basicRectangle withPriority: HighestPriority.
c windowRectangle anchorTo: basicRectangle withPriority: HighestPriority.

"Set up initial constraints: a's display rectangle is some arbitrary value; b and c are 1/4 the size of a;
e.g., b's viewport size is 50@50 compared to a's window size of 100@100)."

a displayRectangle initialAnchorTo: (100@100 extent: 200@200) withPriority: 5.
b viewportRectangle size initialAnchorTo: 50@50 withPriority: 5.
c viewportRectangle size initialAnchorTo: 50@50 withPriority: 5.

"Finally, set up the permanent constraints: b's top left corner is a's window coordinate 0@0; the b
and c bottoms line up."

b viewportRectangle topLeft anchorTo: 0@0 withPriority: HighestPriority.
c viewportRectangle bottom equalTo: b viewportRectangle bottom withPriority: 3.

a open.
```

Note that the borders do not overlap because the window rectangle of **a** refers to the inside of its border, whereas the viewport rectangles of **b** and **c** refer to the outside of their borders. Because **a** is of the fixed-window variety, its contents stretch as follows when it is resized:

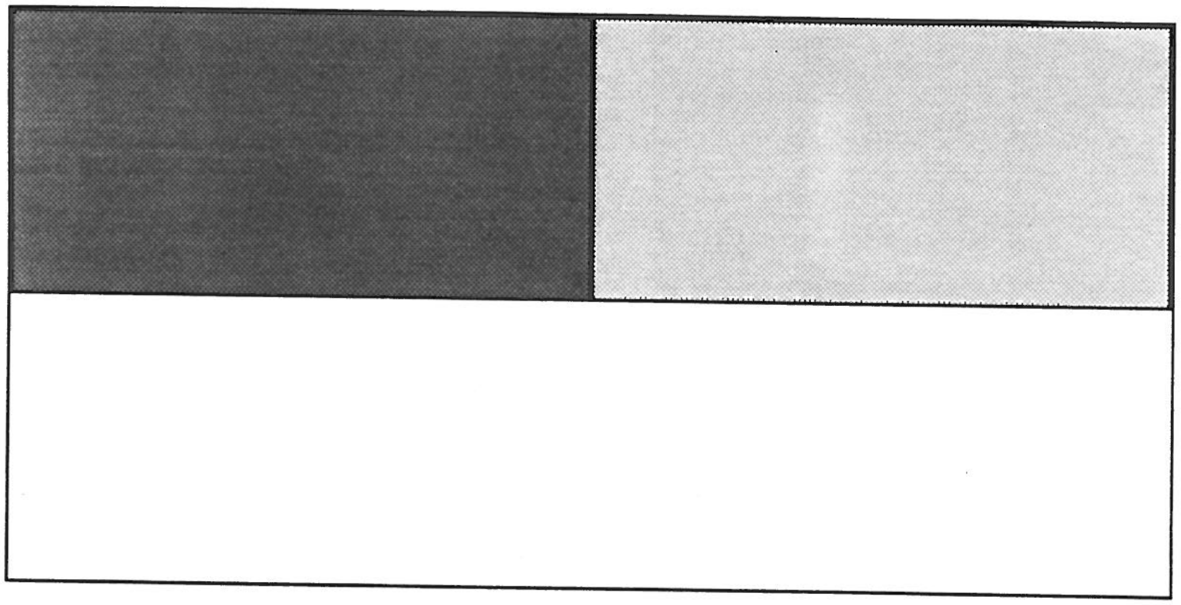


Figure 3: Expanding The Example Window

We permit the user to resize both subwindows, but **b** will always remain in the top left corner of **a** and the bottoms of **b** and **c** will always be aligned:

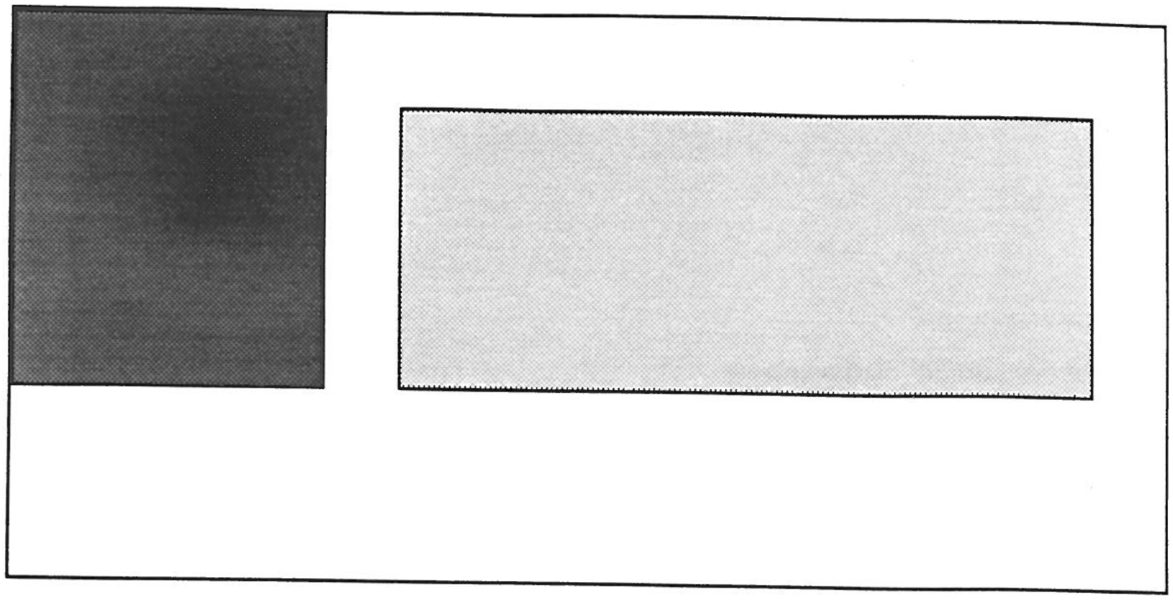


Figure 4: Adjusting The Light Gray Subwindow

Example 2: A Fixed-Scale Pinned Window

This example demonstrates the workings of a fixed-scale window **a** with the center pinned. A quadrangle is drawn in it to illustrate that (1) the quadrangle does not change size (it would have in a standard window) and (2) it remains in the center of **a**. The effect is achieved by anchoring the display transformation's scale and the window center. Since the window rectangle display-transforms into the display rectangle, changing the display rectangle size must correspondingly change the window rectangle size. Since the window center is pinned, the transformation's translation takes up the remaining degree-of-freedom.

```
"A Fixed-Scale Pinned Window Example"
(a ← Window new) borderColor: Form black; insideColor: Form white.
b ← Quadrangle
   region: (-10@-10 corner: 10@10) borderWidth: 1
   borderColor: Form black insideColor: Form gray.
a model: b.

borderWidth ← Rectangle left: 1 right: 1 top: 1 bottom: 1.
a borderWidth anchorTo: borderWidth withPriority: HighestPriority.

"For demonstration, make the window center be 0@0."
windowArea ← -100@-100 corner: 100@100.
a windowRectangle anchorTo: windowArea withPriority: HighestPriority.

"Set up the initial display rectangle as a constraint."
displayArea ← 100@100 extent: 300@300.
a displayRectangle initialAnchorTo: displayArea withPriority: UserPriority.

"Set up the interesting permanent constraints."
a displayTransformation scale anchorTo: 1@1 withPriority: HighestPriority.
a windowRectangle middle anchorTo: 0@0 withPriority: HighestPriority.

a open.
```

This results in the following arrangement of windows:

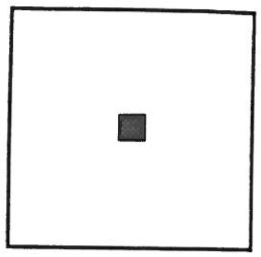


Figure 5: Initial Subwindow

When the window is resized, the model stands still with respect to the center and remains the same size.

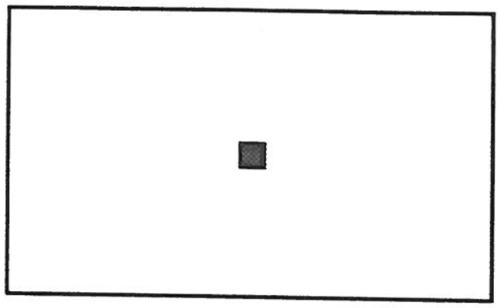


Figure 6: A Fixed-Scale Pinned Subwindow

This is useful for displaying objects that shouldn't be stretched such as text and forms.

Example 3: A Fixed-Scale Unpinned Window

In this example,window **a** is fixed-scale and unpinned. This is achieved by anchoring the entire display transformation (not just the scale as we did above). Consequently, the window is entirely determined when the display rectangle is chosen by the user in a resizing operation. But note that any coordinate in the window must transform to the same place both before and after the resize because the display transformation is frozen.

```
"A Fixed-Scale Unpinned Window Example"

(a ← Window new) borderColor: Form black; insideColor: Form white.
b ← Quadrangle
   region: (-50@-50 corner: 50@50) borderWidth: 1
   borderColor: Form black insideColor: Form gray.
a model: b.

borderWidth ← Rectangle left: 1 right: 1 top: 1 bottom: 1.
a borderWidth anchorTo: borderWidth withPriority: HighestPriority.

"For demonstration, make the window center be 0@0."
windowArea ← -100@-100 corner: 100@100.
a windowRectangle anchorTo: windowArea withPriority: HighestPriority.
```

```
"Set up the initial display rectangle as a constraint."
displayArea ← 100@100 extent: 300@300.
a displayRectangle initialAnchorTo: displayArea withPriority: UserPriority.

"Set up the interesting permanent constraints."
a displayTransformation scale anchorTo: 1@1 withPriority: HighestPriority.

a open.
```

This results in the following arrangement :

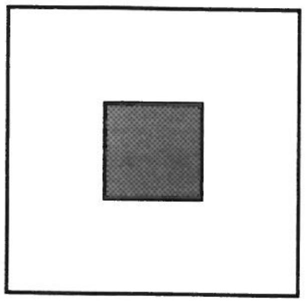


Figure 7: Initial Window

When the window is resized, the model stands still with respect to the screen:

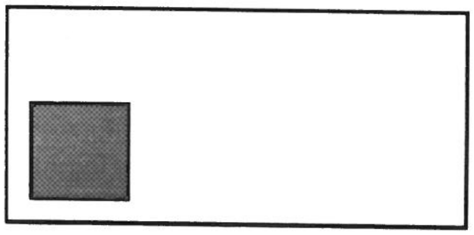


Figure 8: Window After Resizing

It is important to understand the notion of unpinned fixed-scaled not so much because you might wish to use it but because you might accidently constrain the windows in this way. Understanding why it happens is important. Generally we expect to make use of pinned fixed-scale windows rather than the unpinned variety.

Example 4: A Fixed-Size Window

This example demonstrates how fixed-size windows work. The outer window **a** is of the fixed-window variety. The subwindow **b** is a fixed-size window that is forced to the top left corner of **a**.

```
"A Fixed-Size Window Example"

(a ← Window new) borderColor: Form black; insideColor: Form white.
(b ← Window newIn: a) borderColor: Form black; insideColor: Form gray.

border ← Rectangle left: 1 right: 1 top: 1 bottom: 1.
a borderWidth anchorTo: border withPriority: HighestPriority.
b borderWidth anchorTo: border withPriority: HighestPriority.
```

```
"The window rectangles can be anything we want."
basicRectangle ← 0@0 extent: 100@100.
a windowRectangle anchorTo: basicRectangle withPriority: HighestPriority.
b windowRectangle anchorTo: basicRectangle withPriority: HighestPriority.

"Set up initial constraints: a's display rectangle is some arbitrary value."
displayArea ← 100@100 extent: 300@300.
a displayRectangle initialAnchorTo: displayArea withPriority: UserPriority.

"Set up the interesting permanent constraints."
b displayRectangle size anchorTo: 100@100 withPriority: HighestPriority.
b displayRectangle topLeft
   equalTo: a insetDisplayRectangle topLeft withPriority: HighestPriority.

a open.
```

This results in the following arrangement of windows:

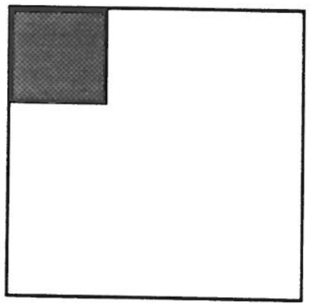


Figure 9: Initial Window

When the outer window is resized, the inner window does not change size and stays in the top left corner:

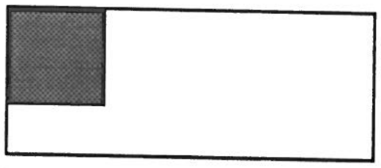


Figure 10: Window After Resizing

This is useful for displaying objects which shouldn't change; e.g., buttons and switches.

Example 5: Arbitrary Sharing of Area Among Subwindows

This example demonstrates how arbitrary sharing of area among subwindows can be specified. The outer window **a** is of the fixed-window variety. The subwindows **b** and **c** are constrained to exactly fill **a** with **b** on the left and **c** on the right. The border between **b** and **c** can be moved as desired.

```
"An Example Demonstrating Area Sharing"

(a ← Window new) borderColor: Form black; insideColor: Form white.
(b ← Window newIn: a) borderColor: Form black; insideColor: Form gray.
(c ← Window newIn: a) borderColor: Form black; insideColor: Form lightGray.
```

```
border ← Rectangle left: 1 right: 1 top: 1 bottom: 1.
a borderWidth anchorTo: border withPriority: HighestPriority.
b borderWidth anchorTo: border withPriority: HighestPriority.
c borderWidth anchorTo: border withPriority: HighestPriority.

"The window rectangles can be anything we want."
basicRectangle ← 0@0 extent: 100@100.
a windowRectangle anchorTo: basicRectangle withPriority: HighestPriority.
b windowRectangle anchorTo: basicRectangle withPriority: HighestPriority.
c windowRectangle anchorTo: basicRectangle withPriority: HighestPriority.

"Set up initial constraints: a's display rectangle is some arbitrary value; b's display rectangle width is
1/3 the width of a's."

displayArea ← 100@100 extent: 300@300.
a displayRectangle initialAnchorTo: displayArea withPriority: UserPriority.
b displayRectangle width initialAnchorTo: 100 withPriority: UserPriority.

"Finally, set up the permanent constraints: b to touch a's three sides pointing left and c to touch a's
three sides pointing right; b's right side also connects with c's left side."

high ← HighestPriority. "Just to make the code below fit on one line."
b displayRectangle left equalTo: a insetDisplayRectangle left withPriority: high.
b displayRectangle top equalTo: a insetDisplayRectangle top withPriority: high.
b displayRectangle bottom equalTo: a insetDisplayRectangle bottom withPriority: high.

c displayRectangle right equalTo: a insetDisplayRectangle right withPriority: high.
c displayRectangle top equalTo: a insetDisplayRectangle top withPriority: high.
c displayRectangle bottom equalTo: a insetDisplayRectangle bottom withPriority: high.

b displayRectangle right equalTo: c displayRectangle left withPriority: high.

a open.
```

This results in the following arrangement of windows:

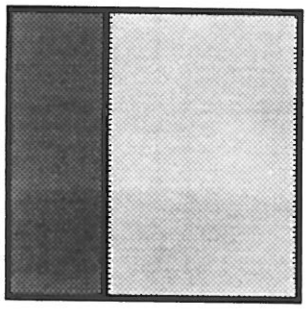


Figure 11: Initial Window

When one of the inner windows is resized, the other one resizes to take up the remaining area of the outer window.

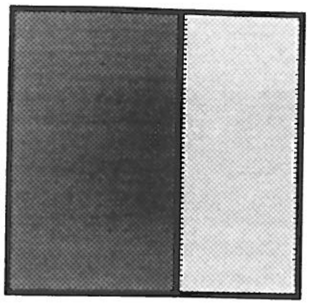


Figure 12: Window After Resizing

This is useful for dividing the area of a window into distinct sections that are used for displaying different aspects of a complex model; e.g., it could be used for dividing the browser's subwindows.

5 Implementation Details

A constraint package (system of constraints) contains a set of variables and a set of constraints (among other things) as shown in Figure 13. Although constraints contain references to the variables they constrain, variables do not refer to the constraints. A constraint package is stored in the top-window for each window hierarchy. Hence it contains the set of variables and constraints for all windows in the hierarchy (see Figure 14). It must also keep track of the fact that different constraints apply when a window is resized versus when it is moved; when a subwindow **w** is adjusted, only windows below the super-window of **w** is affected (its neighbors and its own subwindows).

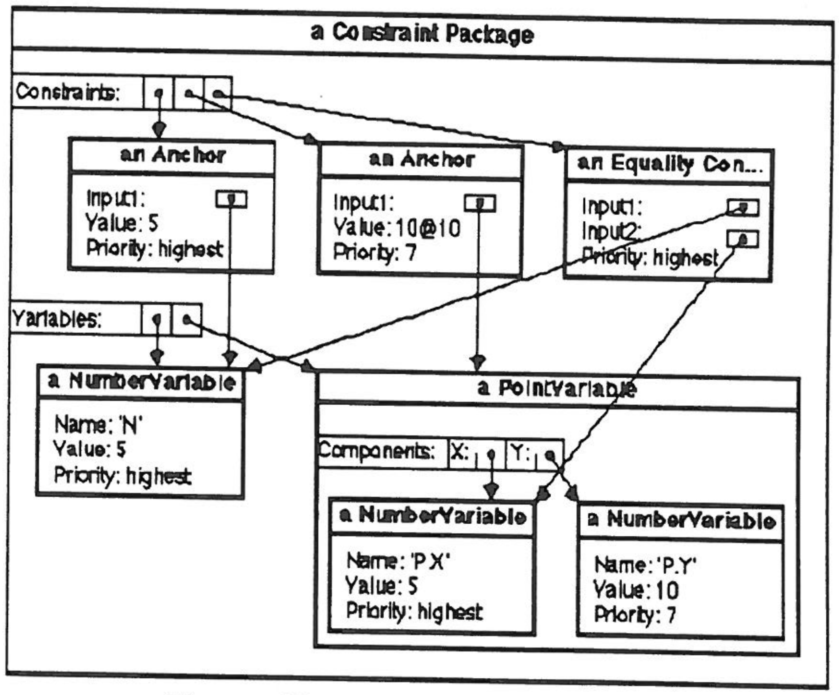


Figure 13: A Constraint Package

A constraint package is solved by asking each constraint in turn to make inferences if it can. In the current system, the order in which the constraints are solved is arbitrary. When all of the constraints can make no more inferences, the package is solved. Solutions then checked for contradictions. A **contradiction** occurs only when the same variable is assigned different values at the same priority. Otherwise, the higher priority value prevails.

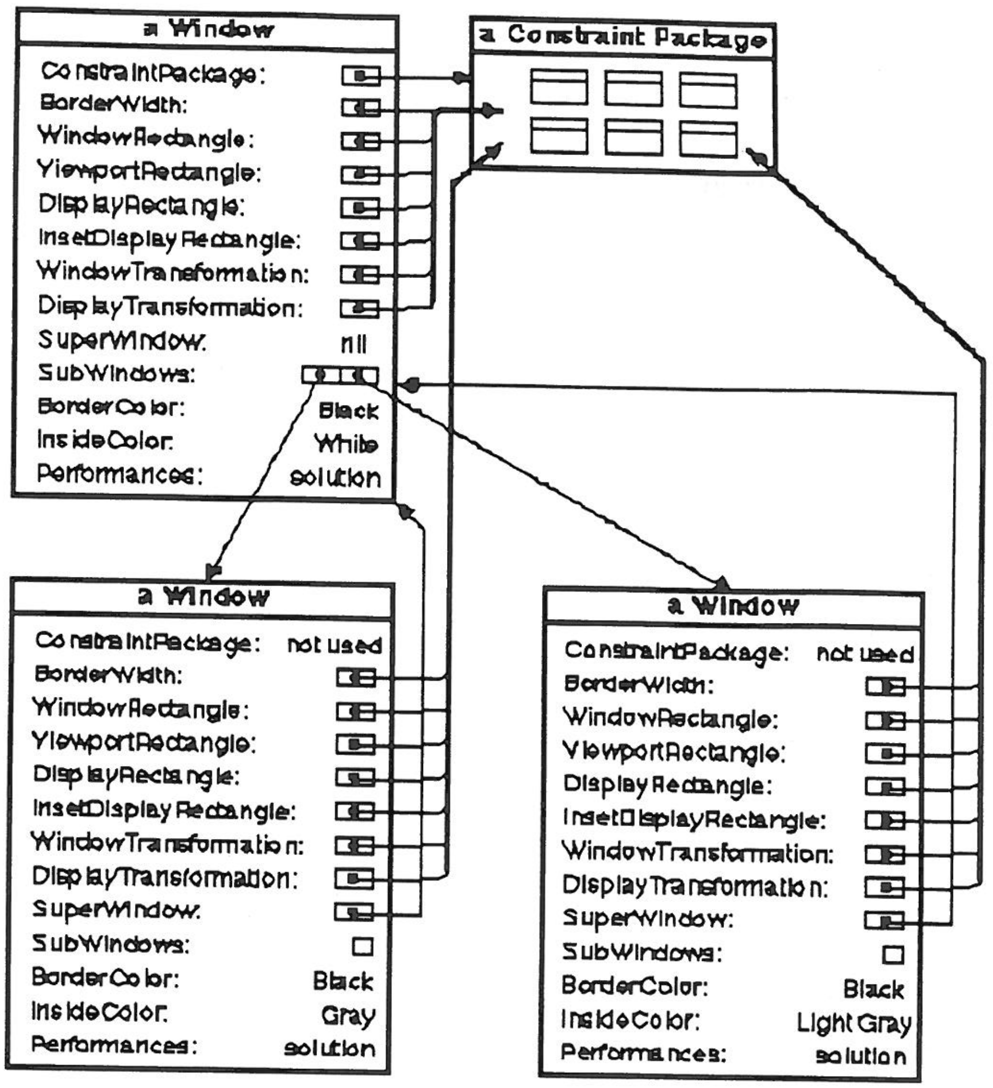


Figure 14: A Hierarchy Of Windows

When a container is anchored to a specific value, each component is also considered to be anchored and when two containers are equated, corresponding components are equated. Also, when two variables are equated at highest priority, these variables can be merged into a single variable. When containers are equated at highest priority, their components can be merged. All of these simplifications could be used to reduce the complexity of the actual solution. A simplified version of a constraint package could be created and the **simplified package** solved. The variables in the simplified package must refer back to the variables in the original package, so that the results can be copied into the original constraint package after solution. The simplified constraint package can be stored so that it need only be created once. An earlier version of our system did this but the speedup was minimal.

Techniques for implementing constraint solvers are known. For example, details can be found in Sussman and Steele [8] and Borning et al [1, 2]. In our case, only a simple system was implemented to test out the notions. We experimented with top-down approaches and bottom-up approaches. There are some obvious inefficiencies. For example, the "compiled" solutions currently consist of a list of method/operands that must be "performed" to get the constraints satisfied. We haven't gone to the bother of physically compiling a method or block containing this list of operations. Our results, reported below, are encouraging precisely because there is much room for improvement.

Although the constraint system does not notice multiple equations in multiple unknowns generally, the window constraint is quite powerful. For instance, it effectively solves 2 equations in 2 unknowns when a window rectangle and its display rectangle is provided without constraining its display transformation. Solving for the display transformation amounts to solving two sets of two equations in two unknowns, the x and y components of the scale. More specifically, the following is solved.

windowRectangle left	*	displayRectangle scale x	+	displayRectangle offset x	=	displayRectangle left
windowRectangle right	*	displayRectangle scale x	+	displayRectangle offset x	=	displayRectangle right
windowRectangle top	*	displayRectangle scale y	+	displayRectangle offset y	=	displayRectangle top
windowRectangle bottom	*	displayRectangle scale y	+	displayRectangle offset y	=	displayRectangle bottom

Table 4: Solving Implicit For ..x And ..y

6 Experimental Results

A constraint package is created when the top-window is created. Currently, it is solved for resizing at the top-window level when it is first opened. More generally, resizing and moving operations are solved for on an as-needed basis when windows are adjusted. Later adjustments of the same window simply execute the cached solution.

Preliminary timing measurements for the first example window constructed in section 4 are provided below. Improvements are still ongoing.

Initial construction of constraint package:	no appreciable time required.
First resize of a window:	approximately 17 seconds.
Successive resizes of a window:	approximately 1/4 second.

Clearly these statistics are preliminary; we expect to have more interesting timing results by the time the paper is due. We are currently working on the second prototype to speed up the first resize operation. Successive resizes can also be speeded up. We've already mentioned that compiling the solution into a method or block should be done. The solutions generated are also sub-optimal. In particular, the current system often computes the same value more than once. Each of these corresponds to satisfying constraints at successively higher priorities. Just removing them will speed up the system.

If a given window configuration is used often, as is the case for browsers, the constraint package with its pre-computed solutions could be stored in a class variable. This would make the first frame operation fast. Alternatively, new windows could be created from prototypes; i.e., by copying sample windows that have been previously opened, fully-precomputed, and then subsequently closed.

8 Conclusions

Constraints provide a natural approach to configuring windows. They give the user considerable power with relatively little increase in complexity. They also permit him to specify windows that could not be specified or created under the current system; namely fixed-size windows and fixed-scale windows. The constraints need not be used for determining a fixed layout for the windows; it can also be used to specify an initial layout. By providing all windows with appropriate blue button menus, this layout can be changed dynamically. Without constraints, such a facility is not likely to work very well because modifications are just as likely to worsen a window design (at least on a temporary basis) as it is to improve it. From the user's point of view, the ability to create these new kinds of constrained windows simplifies the implementation of window-based applications.

Acknowledgements

The research was supported in part by NSERC (Natural Sciences and Engineering Research Council of Canada). Danny in particular would also like to thank Doug Macklem for the use of his Mac II system and for his constant support as well as Peter Epstein for discussions of concepts.

References

1. Borning, A. *The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory*. TOPLAS, 3 (4), October 1981, pp. 353-387.

2. Borning, A., Duisberg, R., Freeman-Benson, B, Kramer, A., and Woolf, M. *Constraint Hierarchies*. OOPSLA '87, Orlando, Florida, October, 1987, pp. 48-60.

3. Borning, A. and Duisberg, R. *Constraint-Based Tools for Building User Interfaces*. ACM Transactions on Graphics, 5 (4), October 1986.

4. Epstein, P. Private communication. 1988.

5. LaLonde, W.R. and Pugh, J.R. *Inside Smalltalk*. Prentice-Hall (forthcoming).

6. Macklem, D. *Graphical Database Design*. Bachelor of Computer Science Honours Project, School of Computer Science, Carleton University, 1987.

7. Nickel, R. *Towards a New Smalltalk User Interface*. Masters of Computer Science Thesis, School of Computer Science, Carleton University, 1986.

8. Sussman, G. and Steele, G. CONSTRAINTS — A Language for Expressing Almost-Hierarchical Descriptions. Artificial Intelligence, 14 (1), January 1980, pp. 1-39.