The Layout Widget

A Hierarchical Constraint Based Widget

Keith Packard†

Abstract

The X Toolkit `[Swick90]` geometry management process is extremely flexible and powerful; however the existing composite widget classes make it difficult for the application developer both to simply design an application layout, and even more important, to make the layout work in a wide variety of environments.

The Layout widget class is described which uses a stretch/shrink model similar to TeX `[Knuth84]` to constrain the layout of an application in a manner which allows the geometry of the children to respect the desires of the application designer, while adapting to its environment, both in terms of the changing geometry allocated to the widget, and to the changing needs of the child widgets. The specification of the child layout is entirely contained in a resource which is interpreted at run time.

Introduction

History

Over a year ago, I was rewriting the user interface of a game program (xmille) to use the X Toolkit and Athena widget set instead of the original X10 based user interface. When it came time to construct a widget tree which could represent the layout of the various widgets, I started with the Athena composite widgets (Paned, Box, and Form) and tried to come up with a general mechanism which would automatically choose the correct size for the application, regardless of the fonts used and also allow the application to be resized in a useful manner.

---

†*Keith Packard (**keithp@ncd.com**) has been a member of the Network Computing Devices engineering staff since last June. Prior to that, he was the senior staff member at the X Consortium. He has worked with and participated in the design of much of the X Window System since early days of the X Consortium.*

After a few hours of frustration, it became clear that no combination of the existing widgets would do what I wanted. I gave up the project and went on to other things. In the back of my mind, I had an idea that a new composite widget class which could solve the xmille problem would also be generally useful for a wide variety of layout problems, so I started to design what would become the Layout widget.

What I wanted was a general 2D layout system which would take into consideration both some sort of abstract geometrical description of the layout along with the sizes of the child widgets and 'do the right thing' to produce a reasonable presentation on the screen.

Goals

The Layout widget tries to meet the following goals:

- Flexible.

The widget should be usable in every situation to which it might be applied. This relates both to how the external system constrains the space available to the widget as well as how the child widgets relate to their parent. For example, the user should be able to use a different font resource and have the parent widget adapt automatically and reasonably, giving the child widgets more space as needed. An application should be able to be run in an extremely limited amount of space, and yet still provide useful information to the user. Widgets containing more important information should be given more space than others.

- Easy to use.

Constructing the widget layout should be straightforward and intuitive. The constraint model should be easily understood by all developers. Some flexibility in what layouts were available could be limited in order to make the system substantially easier to use.

- Readily modifiable

When developing the user interface, the layout should be quickly modifiable to allow the programmer to try alternative layouts without extensive work. In fact, by placing the layout description in a resource attached to the widget, even the end user can adjust the layout as necessary to work in different environments. This allows the user to reconstruct the appearance of the application.

- Predictable.

In every situation to which the layout could be subjected, the results on the screen should be easily predicted from the layout description. If the results on the screen surprise the developer, just think what they're going to do to the user. The developer is not going to run the application in all possible environments. Therefore, they should be able to create a layout description which can run with almost any limitations.

Solution

Introduction

The design of the Layout widget was a long process; several iterations were required to transform the idea of a general 2D layout system into something which is actually useful. I will go over the basic requirements of any solution and how those requirements meet the goals stated above. Then I will discuss the constraint system and how a hierarchical approach is useful. Finally I will give some complete working examples.

Requirements of the solution

These follow directly from the goals of the project.

- Constraint based.

This requirement follows from goals of flexibility and predictability. A constraint based system, if correctly specified, can be modeled easily by the programmer and will automatically adapt to changing circumstances. By always constructing the layout based on the constraints, the results on the screen are be predictable.

- Hierarchical

This means that sets of child widgets are grouped together and treated as a composite object (which can be grouped with other objects etc.). This allows the user interface to be constructed in larger pieces, making the interactions the developer has to understand limited to a single level.

- Respect child widget geometry

When a child widget requests a change it its geometry, the request should be honored as far as possible. This allows the geometry to reflect not only the desires of the developer, but also the requirements for each widget.

- Be useful for almost all 2D layouts

Where this doesn't conflict with the goal of usability, the system should be designed to provide as much flexibility in describing the layout as possible.

The Pieces of the System

The Layout widget consists of several cooperating ideas. First, the layout is be described hierarchically. Next constraints are associated with each object. Finally, the constraints are propagated up the hierarchy to produce the desired layout.

Hierarchical construction

The Layout widget composes objects into boxes stacking them either vertically or horizontally. These boxes can then be composed along with other objects into other boxes. At the top level, the Layout widget contains a single box. To specify the layout for a window, the proposed geometry is first broken down into boxes. Figure 1 shows an example of a simple dialog box which fits the hierarchical layout system quite nicely. Dashed lines indicate how the dialog can be broken down into boxes, each box containing objects aligned either vertically or horizontally.

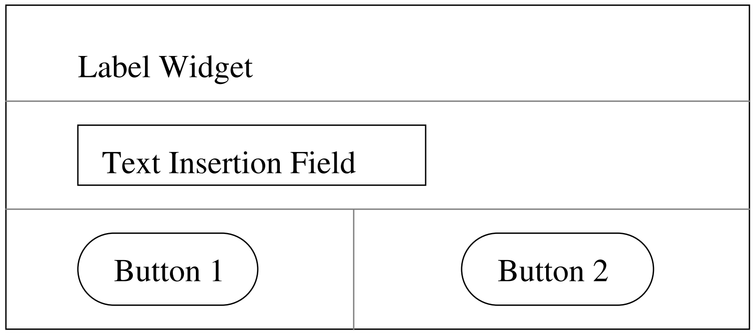

Figure 1: Example Dialog Box

The hierarchy of these boxes is specified as:

Vertical Box containing:
    Label Widget
    Text Insertion Field
    Horizontal Box containing:
        Button 1
        Button 2

By specifying the layout hierarchically, the problem of how to position objects relative to one another has been reduced to a problem of how to arrange objects next to one another in a box.

This system is in contrast to a constraint system using a flat topology, where every child widget is treated at the same level and is placed in the constraint system individually. In such a system the above layout might be described as follows:

Top of the Label is bound to the top of the composite widget
Left edge of the Label is bound to the left side of the composite widget
Right edge of the Label is bound to the right edge of the composite widget
Bottom of the Label is bound to the top of the Text Insertion Field
Left edge of the Text Insertion Field is bound to the left side of the composite widget
Right edge of the Text Insertion Field is bound to the right side of the composite widget
Bottom of the Text Insertion Field is bound to the top of Button 1
Bottom of the Text Insertion Field is bound to the top of Button 2
Bottom of Button 1 is bound to the bottom of the composite widget
Bottom of Button 2 is bound to the bottom of the composite widget
Left edge of Button 1 is bound to the left side of the composite widget
Right edge of Button 1 is bound to the left edge of Button 2
Right edge of Button 2 is bound to the right side of the composite widget

Unless some structure is given to these constraints, the number grows quickly with the complexity of the layout. However, a non-hierarchical systems could also solve many problems which the Layout widget does not address. A layout like:

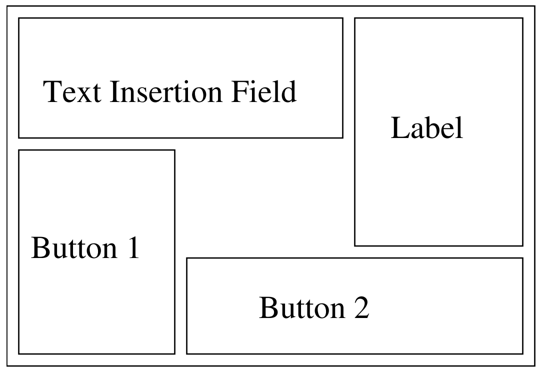

Figure 2: An example of something Layout doesn't do

cannot be described using a hierarchy of horizontal and vertical boxes. As such layouts are not really all that useful, and the ability to combine objects together into composite objects makes the problem of describing the layout significantly easier, I decided to forgo the flat layout in lieu of an easier to use system.

The Layout Constraint System

A Constraint system consists of a collection of statements (constraints) about the relations between the objects to be laid out which should be all made simultaneously true, whenever possible. For example, referring to Figure 1 above, the following constraints could be used to guide the positioning of the objects:

- The Label, the Text Insertion Field and Button 2should be left aligned.
- Those elements should be spaced about the same distance from the left edge as the Label is tall.
- Button 2 should be spaced from the right edge the same amount as Button 1 is spaced from the left edge
- The buttons should be spaced about three times as far apart as they are from each edge.
- The Text Insertion Field shouldn't shrink below a 'reasonable' size; however it should be given as much space as it needs; even if the dialog box must get larger.
- The vertical spacing between all of the various fields should be about the same as the height of the Label.

Given this description of the constraints desired, a system for describing them is needed. There are several options, the one Layout uses is derived from the TeX 'Boxes and Glue' model.

The Layout model

In the Layout constraint system, the objects grouped into a box have three attributes

- Natural size

This is the size which this object would like to be. For widgets, this is the size reported by XtQueryGeometry.

- Stretch

This is a value which indicates the willingness of this object to grow beyond the its natural size.

- Shrink

This is a value which indicates the willingness of this object to become smaller than its natural size.

Both stretch and shrink can be additionally attributed with an order of 'infinity'. This works the same way infinity works with TeX glue—a value of one order of infinity is larger than any value of any smaller order of infinity (e.g. $1\ \infty^2 > 1000\ \infty^1$). Each object has a natural size, stretch and shrink in both dimensions. Shrink values of order zero are a special case; they indicate the maximum amount that an object can be shrunk by. The Layout widget will not reduce the object by more than this amount.

Boxes are composed along either a vertical or horizontal axis. All of the objects in a box are placed along this axis. Once the size of a box is determined, the size and placement of the objects inside it can be computed. Normal to the axis of the box, each object is placed at the edge of the box and made as large as the box. Along the axis of the box, each object is given a piece of the difference between the actual size of the box and the natural size of the box. When $Shrink_{object}$ has a smaller order of infinity than $Shrink_{box}$, the object is left in its natural size. Otherwise, if the box is smaller than its natural size, each object is shrunk in proportion to its shrink value:

$$Size_{object} = Natural_{object} - \frac{Shrink_{object}}{Shrink_{box}} \times (Natural_{box} - Size_{box})$$

Similarly, when the box is larger than its natural size, each object with stretch of the same order of infinity as the stretch value of the box is resized in the ratio:

$$Size_{object} = Natural_{object} + \frac{Stretch_{object}}{Stretch_{object}} \times (Size_{box} - Natural_{box})$$

These equations are intended to be identical to the TeX model.

Inherited Stretch and Shrink

Insanity is hereditary: you get it from your kids.

In a similar fashion, boxes inherit their natural size, stretch and shrink values from the objects contained in them. An alternative method which was tested and discarded had explicit stretch and shrink values for these composite objects. This was not reasonable because it forced the user to generate many more values for the objects, and it meant that the attributes of the primitive objects could be overridden by the attributes of the composite objects above them in the tree. It is much easier to have the Layout widget compute ‘reasonable’ values for the composite boxes and use them instead. Some trouble was taken over the development of the mechanism for inheriting these values.

Along the axis of the box, the simplest model also turns out to work exceedingly well:

- The natural size is the total of the natural sizes of the objects.
- The stretch is the sum of the child objects stretch values.
- The shrink is the sum of the child objects shrink values.

The sum of stretch and shrink values is complicated by the notion of infinity: the sum of two values a and b has order of the maximum of the orders of a and b. If a and b are of the same order then the base value is the sum of the base values of a and b. Otherwise, it is the base value of the one of larger order.

Normal to the axis of the box, things are different and no obvious method jumps out. Because all of the objects in the box are expanded to the size of the box along this dimension, the constraints for the box must take all of the objects into account. The attributes are designed to respect the requests of the most rigid child object; this will automatically satisfy the constraints of all of the other objects.

- The natural size is the natural size of the object with the smallest order of infinity for its shrink attribute. If more than one object qualifies, the largest natural size is used. Either stretch or shrink could have been chosen here, but shrinking a box usually requires more care than stretching.
- The stretch is the minimum of all of the stretch values.
- The shrink value is essentially the minimum of the shrink values. However, if the minimum shrink order is zero, then the shrink value is the natural size of the box minus the largest minimum size of objects of shrink order zero. This seems complicated, but the effect is to honor the special semantics for shrink of order zero by not shrinking any object by more than its shrink value.

The minimum of stretch and shrink values is also complicated by infinite values. The minimum of two values a and b is the value with minimal order, unless they have the same order in which case it is the value with a smaller base value.

Glue

The above discussion leaves out one piece of the puzzle: how to space widgets so that they have blank areas between them. Glue is just another object which can be composed in a box along with other boxes and widgets. It has the three attributes of natural size, stretch and shrink; however it only has these attributes for the axis of the box in which it resides. Along the other axis, glue is simply ignored.

The Layout Language

Instead of using the Xt constraint mechanism, which attaches constraint resources to the child widgets, the Layout widget has a single resource which describes the geometry of the entire set of widgets. This is because describing the hierarchy is most clearly done in a single place, using indentation or some other syntactic sugar to clearly indicate the structure, and placing the constraints separate from the hierarchy makes it difficult to associate the two. This is a BNF representation of the major elements of this language:

Box	:==	**vertical** **{** Objects **}**
	:==	**horizontal** **{** Objects **}**
Object	:==	Box
	:==	Widget
	:==	Glue
	:==	Variable
Widget	:==	*widget-name* **< +** Value **-** Value * **+** Value **-** Value **>**
Glue	:==	Expression **< +** Value **-** Value >
Variable	:==	*variable-name* **=** Expression
Value	:==	Expression **inff***

The constraints on a widget are expressed as < + horizontal-stretch - horizontal-shrink * + vertical-stretch -vertical-shrink >, while those for glue are simply < +stretch -shrink >, the orientation of the glue being inherited from the box it is contained in. When a stretch or shrink value, along with the + or - character is not specified, it is taken as zero. For example, < +inf * -10 > specifies a widget constraint of 1, infinity order 1 horizontal stretch, 0 horizontal shrink, 0 vertical stretch and -10 infinity order 0 shrink.

The order of infinity for a particular value is the number of **f** characters in the **inf** following the base value. For example, `1inff` has order 2. Expressions are simple arithmetic (`+`, `−`, `*`, `/`) along with some special syntax:

Expression	:==	Expression *binary-op (+, -, *, /)* Expression
	:==	**width** *widget-name*
	:==	**height** *widget-name*
	:==	Expression **%** **of** Expression
	:==	Expression **%**
	:==	**$** *variable-name*

Width and **height** return the size of the named widget in the indicated direction. **% of** is a simple shorthand for a / b * 100, while a **%** standing alone returns that part of the size of the object to which the constraint applies.

Variables are a simple short hand for repetitive typing. Assign an expression to a variable, and then use that variable in a Value and the expression will be evaluated in the context of where that value is used (which doesn't really matter, except for the **%** operator).

A complete example

Looking back to Figure 1, the complete layout description for that widget which follows the constraints given above would be:

```
vertical {
    space = height label
    $space < +inf -100% >
    horizontal {
        $space < -100% >
        label
        $space < +inf -100% >
    }
    $space < +inf -100% >
    horizontal {
        $space < -100% >
        textInsertionField
        $space < +inf -100% >
    }
    $space < +inf -100% >
    horizontal {
        $space < -100% >
        button1
        (3 * $space) < +inf -100% >
        button2
        $space < -100% >
    }
    $space < +inf -100% >
}
```

Note how some additional boxes were added to space the textInsertionField and label from the edges of the dialog. This same layout could be described in many other ways.

When this dialog box is resized by the user, the vertical white space between the various fields is resized evenly, and the buttons gravitate to the edges of the dialog. When the textInsertionField requests more space, the dialog box will try to become larger.

Summary

The original problem of how to layout the xmille program has been resolved; xmille now runs on everything from a 1280x1024 NCD terminal to a 640x480 PC; and even when the fonts used internally are made larger (to accommodate a visually impaired colleague), the layout continues to function well.

The original goals of the project have been met, and the resulting Layout system is simple to use and powerful enough to solve a wide variety of user interface problems. Instead of spending several hours trying different combinations of composite widgets; simply starting with the Layout widget and trying different specifications makes the development of new applications extremely rapid.

The Layout widget is not restricted to use with the Athena widgets; it can be integrated with Motif or OLIT applications. It is available via anonymous FTP along with a collection of sample programs which use it from export.lcs.mit.edu as Layout.tar.Z.

References

`[Knuth84]` Donald E. Knuth, *The TEXBook*, Addison Wesley Publishing Company, 1984.

`[Swick90]` Paul Ascente and Ralph Swick, *X Window System Toolkit*, Digital Press, 1990.

`[Scheifler92]` Robert W. Scheifler and James Gettys, *X Window System*, Digital Press, 1992.