Layout and Constraints

In chapter 4 we discussed how user interfaces can be assembled from previously built widgets. One of the issues that we deferred was the placement of those widgets on the screen. When a user drags or resizes windows and other objects, many other things must be kept consistent. Widgets must be repositioned, connecting lines between objects must stay connected, alignments must be preserved, etc. Keeping objects geometrically consistent is a fundamental problem. When the objects are widgets it is called the layout problem. When the objects are other things it is called constraints.

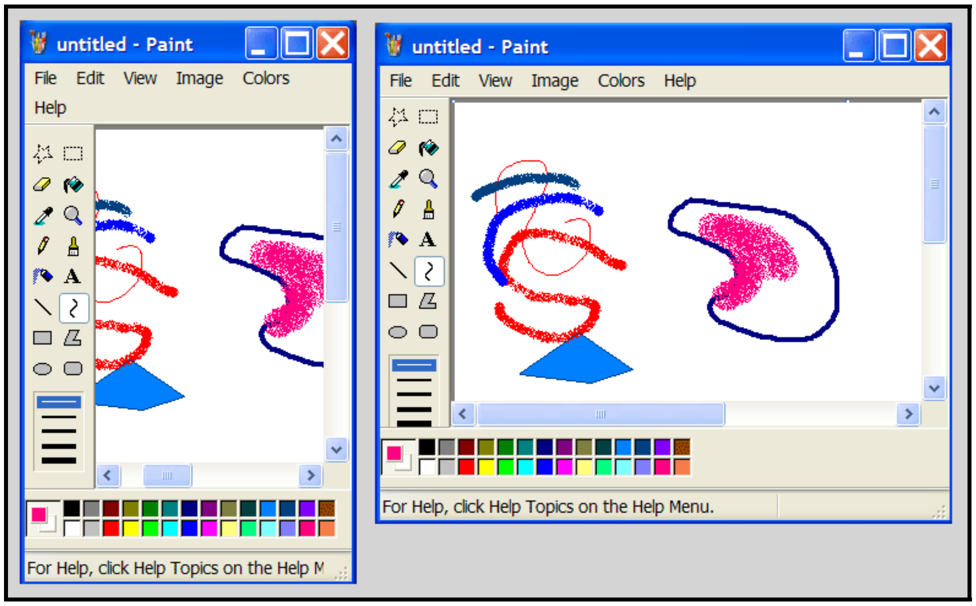

untitled - Paint
File
Edit
View
Image
Colors
Help
For Help, click Help Topics on the Help M
untitled - Paint
File
Edit
View
Image
Colors
Help
For Help, click Help Topics on the Help Menu.

Figure 5.1 – Widgets move when windows are resized

Consider the widgets in figure 5-1. When the window changes size, there are a variety of things that happen. The scroll bars and color palette remain a constant distance from the right and bottom edges. When there is not enough room for the menu bar it changes itself to two lines. When the menu bar occupies two lines, it causes the painting and the tool palette to move down to make room for the second menu line. When there is not enough room for the line width selector, it gets clipped off of the bottom. When the window gets narrower, the paint region in the middle changes size. When the paint window changes size, the scroll-bars change the size of their sliders. All of these work together so that the user interface retains visual consistency. This chapter addresses these kinds of issues. We will first discuss widget layout. Secondly we will look at constraints, which are a more general model for maintaining visual and interactive consistency.

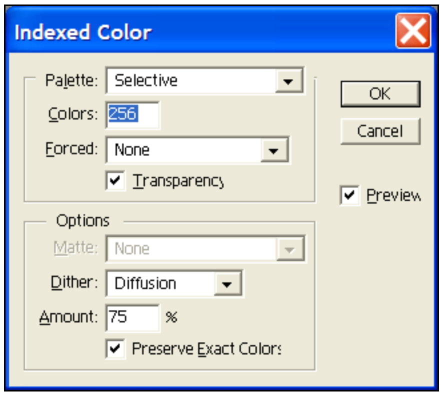
Layout

Figure 5.1 demonstrates most of the problems that must be addressed by a layout system. When we use windowed applications, we assume all widgets move appropriately. However, this movement must be programmed to behave correctly. The layout problem is complicated by the fact that we really want to design layouts by drawing them. Since Bill Buxton's MenuLay[1] system was published in 1983 it has been clear that drawing widgets and their layout is preferable to programming the layout in many cases. Our layout models therefore are constrained by whether we want to program them or draw them. If we want to draw layouts, we must consider the user interface for doing so. The three most popular layout algorithms are fixed position, edge-anchored and variable intrinsic size.

For each style of layout we must deal with three issues: 1) what information must be stored with the widgets from which the layout can be computed, 2) what is the algorithm for computing the layout, and 3) how will the interface designer specify the layout either interactively or programmatically.

Fixed position layout

The simplest layout mechanism is to assign every widget a fixed rectangle whose coordinates are relative to the rectangle of its parent widget. Figure 5.2 shows a dialog box that uses a fixed layout. All of the widgets are of fixed size and there are a fixed number of them. There is no reason to resize this dialog and therefore a fixed layout is entirely appropriate.



Palette: Selective
Colors: 256
Forced: None
✓ Transparency
✓ Preview
Matte: None
Dither: Diffusion
Amount: 75 %
✓ Preserve Exact Colors

Figure 5.2 – Fixed size dialog

The data required is simply a rectangle for each widget and the layout algorithm is very straightforward. Since each child has its desired bounds determined relative to its container widget, the container widget simply sets the child's bounds location (left,top) to be its own location plus the desired child's location. The child's size is unchanged from its desires. This allows a container widget to be moved around and all of its children will stay in position relative to the container. The recursive nature of the algorithm repositions children's children. The algorithm is shown in figure 5.3.

```
public class Widget
{
        other fields and methods
    public Rectangle desiredBounds;
    public void doLayout(Rectangle newBounds)
    {   setBounds(newBounds);
        foreach child widget C
        {   Rectangle newChildBounds=
                new Rectangle(newBounds.left+C.desiredBounds.left,
                    newBounds.Top+C.desiredBounds.top,
                    C.desiredBounds.width, C.desiredBounds.height);
            C.doLayout(newChildBounds);
        }
    }
}
```

Figure 5.3 – Fixed layout algorithm.

The advantages of fixed layout are a simple algorithm, minimal data and a very simple model to specify. When designing programmatically we just need to construct a rectangle with the desired location and size, set desiredBounds and then add the widget to its container. Figure 5.4 shows an interface design being drawn interactively. The designer positions and resizes each widget to create a complete design. Designing an interface with fixed layout is as simple as drawing in a variety of other applications. A final advantage is that the model is trivial to understand and all widgets stay exactly where the designer has placed them. This model was used in Apple's HyperCard[2] and early versions of Visual Basic. The primary disadvantage is that it does not handle resizable windows. If the window gets smaller, widgets are simply clipped or not displayed. If the window gets bigger all of the widgets stay in the upper left-hand corner of the container and waste the additional space. In cases like figure 5.2, this is fine but for the paint program in figure 5.1, fixed layout would be unacceptable. In a toolkit that uses fixed layout, the problems of figure 5.1 would need to be handled by special code in response to window resize events. The edge-anchored and variable intrinsic size layout algorithms attempt to eliminate the programmer's need to write such code.

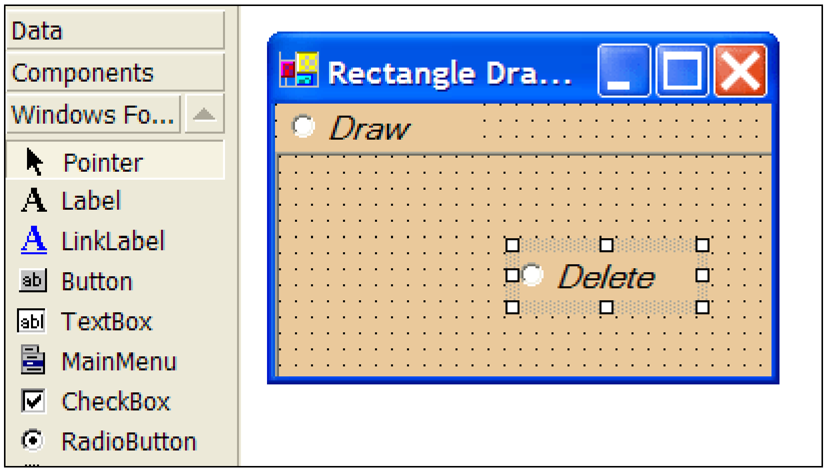


Figure 5.4 – Drawing widget layouts

Edge-anchored layout

We can expand the fixed layout algorithm to accommodate most of the common resizing needs. Note that in figure 5.1 most of the layout issues involve keeping widgets collected around the edge of the pane with the central space occupied by the primary paint area. We can accommodate this behavior by modifying our widgets so that their edges can be anchored to the edges of their container's rectangle. This form of layout was first introduced by Cardelli[3]. One of its more popular current uses is in Visual C# and Visual Basic.

Using the C# form of edge-anchored layout, each widget has coordinates for left, right, top and bottom. The interpretation of these depends upon the boolean values anchorLeft, anchorRight, anchorTop, anchorBottom. In addition the widget has a desiredWidth and desiredHeight. The algorithms for the X and Y axes are independent and identical so we will only describe how the layout in X works.

The core idea is that an edge is either anchored or not. If an edge is anchored, then it is a fixed distance away from the corresponding container edge. If only one edge is anchored, then the desired width is used to determine the location of the other edge. If neither edge is anchored, then the left value is used to compute a proportional distance between the container edges and the desired width is used for the other edge. If both edges are anchored, they follow their respective edges. The algorithm is shown in figure 5.5.

```
Public class Widget
{
        . . . .  other fields and methods . . .
        public Rectangle bounds;
        public int left, right, top, bottom;
        public boolean anchorLeft, anchorRight, anchorTop, anchorBottom;
        public int width, height;
        public const int MAXSIZE=4096;
        public void doLayout( Rectangle newBounds)
        {
                bounds=newBounds;
                foreach child widget C
                {       Rectangle childBounds=new Rectangle();
                        if (C.anchorLeft)
                        {       childBounds.left=newBounds.left+C.left;
                                if (C.anchorRight)
                                {       childBounds.right=newBounds.right-C.right; }
                                else
                                {       childBounds.right=childBounds.left+C.width; }
                        }
                        else if (C.anchorRight) // right is anchored left is not
                        {       childBounds.right=newBounds.right-C.right;
                                childBounds.left=childBounds.right-C.width;
                        }
                        else // neither edge is anchored
                        {       childBounds.left = newBounds.width*C.left/MAXSIZE;
                                childBounds.right=childBounds.left+C.width;
                        }
                        . . . . perform similar computation for Y . . . .
                        C.doLayout(childBounds);
                }
        }
}
```

Figure 5.5 – Edge-anchored layout algorithm

There are a few key ideas to this algorithm. The first is that a widget never does it own layout, nor does it use the anchor or location information. The widget's container does all of that. When a widget receives a doLayout() call it simply accepts the bounds information and then performs layout on its children if it has any. Normally left and right are the number of pixels from their respective edges. If one edge is unanchored then the width/height is used to compute its position from the other anchored edge. If both edges are unanchored then we use left to store a fractional value. Since left is an integer we use left/MAXSIZE as the fractional value. For example if left==MAXSIZE/2 then width*left/MAXSIZE would be width*(MAXSIZE/2)/MAXSIZE or width/2. This would horizontally center the widget. Most layout problems can be handled with this mechanism. Setting left to 0 would place the widget at the far left.

To lay out a widget, a programmer sets the various member fields and adds the widget to its container. The container then will make certain that the widget's bounds are placed in the correct position. Interactively, we need a user interface to specify the values. For the most part the user interface is the same as that shown in figure 5.4. The user draws out the bounding rectangle for the widget and then moves it around to the correct place on the form. Visual Studio provides a special editor for the anchors. Two examples of this are shown in figure 5.6. The anchor editor on the left is the default and shows the widget anchored to the top and left. This is exactly the same behavior as the fixed layout algorithm. The editor on the right is only anchored to the left edge with the vertical position of the widget to be proportional between the top and bottom.

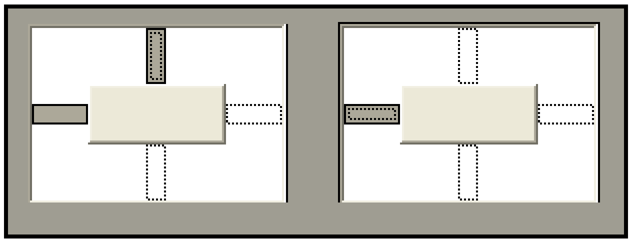


Figure 5.6 – Anchor editors

Figure 5.7 shows an interface design with the form at two different sizes. The button is using the anchors shown on the left in figure 5.6 and the text box is using the anchors shown on the right.

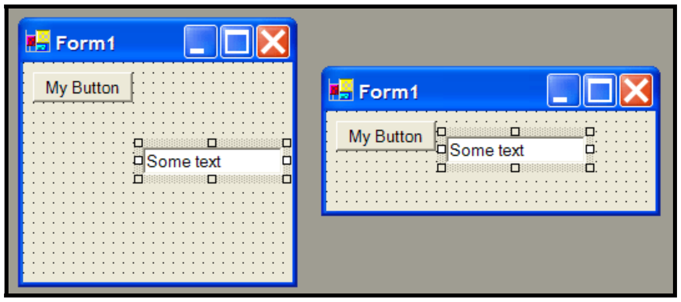


Figure 5.7 – Resizing with anchors

Revisiting our paint application in figure 5.1, we would lay out the pane with the paint area and its scrollbars using anchors to all edges. This would allow room for the menu bar above and the palettes below and to the left. The paint area essentially grows with the window itself. The vertical scroll bar would be anchored top, bottom and right to the paint area pane. Its left edge would be determined by the width of the scroll bar.

Variable intrinsic size layout

The fixed layout and edge-anchored layout presume that the designer wants to manually position the widgets. Many times this is not the case. There are many situations where one wants to create a list or other group of widgets and have them position themselves in an appropriate way. A good example is a menu. We add items to the list and the items all position themselves in the list and the width of the menu is automatically determined. We are not required to manually design the menu, it just works. Another example is when changing an interface from English to German or English to Kanji (Japanese ideographic writing). German words are in general longer than English words. This means that many labels will need to get larger, which will force changes in the layout. Using the two previous mechanisms, the layout will need to be redrawn by the designer. Ideographic languages cause similar layout distortions. Inserting or removing items from a list will also cause the interface layout to be redesigned. In these cases we may prefer a layout model that handles these automatically.

The variable intrinsic size layout is designed to automatically adapt the layout to changes in interface content. The idea is that each widget knows how much screen space it can reasonably use (intrinsic size) and each container has a plan for how to allocate screen space based on the needs of its children. By recursively arranging containers with various layout plans we can produce a wide variety of designs that automatically adjust to their content.

This layout model was first developed Donald Knuth in his T_EX[4] system for laying out mathematical formulae. This layout strategy was adapted to user interfaces by Linton, Vlissides and Calder in their InterViews[5] system. Its most popular current use is in the Java user interface architecture. A restricted form of this algorithm is used in HTML <table> tags.

Basic Layout Algorithm

The variable intrinsic size layout algorithm is based on two recursive passes. The first pass requests each widget to report its desired size. The second pass sets the widget bounds. The general algorithm is shown in figure 5.8.

```
public void doLayout(Rectangle newBounds)
{
    foreach child widget C
    {    ask for desired size of C }
    based on desired sizes and newBounds, decide where each child should go
    foreach child widget C
    {    C.doLayout( new bounds for C); }
}
```

Figure 5.8 – Generic layout algorithm

Most modern intrinsic size layouts ask a widget for its minimum size (the smallest dimensions that it can effectively use), its desired size (the dimensions that would work best for this widget) and its maximum size (the largest dimensions that it can effectively use). These size values are determined by the nature of each widget and, if it is a container, by the sizes of its children. This is done by adding the methods in figure 5.9 to the Widget class. Each class of widget must implement these three methods in its own way.

```
public class Dimension
{    public int width;
     public int height;
}

public class Widget
{
     . . . other methods and fields . . .
     public Dimension getMinSize() { . . . }
     public Dimension getDesiredSize() { . . . }
     public Dimension getMaxSize() { . . . }
}
```

Figure 5.9 – Abstract methods to report desired sizes

Sizes of simple widgets

Figure 5.11 shows a button widget with various properties of the widget identified. The marginWidth and bevelWidth are usually properties set by the designer. The label width depends upon the actual text of the label and its font. There is usually a margin height property that is not shown. The label height is also computed from the label's font. Based on these values we can implement the size methods for a button as in figure 5.10.

```
public class Button
{       . . . other methods and fields . . .
        public Dimension getMinSize()
        {       int minWidth = bevelWidth*2+font.getLength(labelText);
                int minHeight = bevelWidth*2+font.getHeight();
                return new Dimension(minWidth,minHeight);
        }
        public Dimension getDesiredSize()
        {       int desWidth = bevelWidth*2+marginWidth*2+font.getLength(labelText);
                int minHeight=bevelWidth*2+marginHeight*2+font.getHeight();
                return new Dimension(desWidth,desHeight)
        }
        public Dimension getMaxSize()
        {       return getDesiredSize(); }
}
```

Figure 5.10 – Desired size methods for an example button

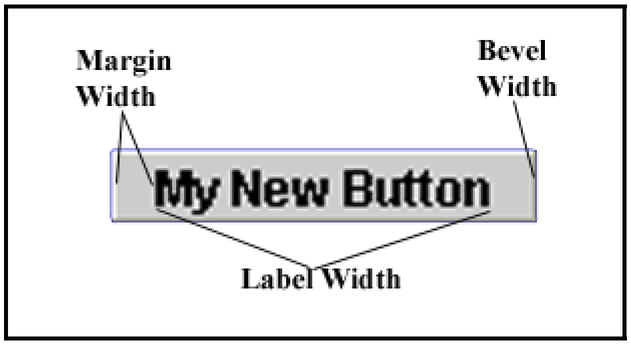


Figure 5.11 – Button size parameters

For the minimum size we dispense with the margins around the text so that the button can be as small as possible. We need the bevel space to show whether the button has been pressed or not. Of course we must have the button label or the user will not know the button's purpose. The desired size adds in the margins so the button will look better. Note that the maximum size is the same as the desired size. There is no need to give the button more screen space.

Figure 5.12 shows a horizontal scroll bar. The scroll bar has different size needs from the button. Its height is fixed so its min, desired and max height will all be the same constant. Its minimum width would be arrowWidth*2+sliderWidth*2. This would be just enough to show all of the controls with a little room to move the slider. Its desired width might be arrowWidth*2+sliderWidth*5, which would provide more room to scroll. Its maximum width would be some constant MAXSIZE because the scroll bar wants as much horizontal space as it can get.

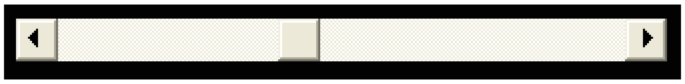


Figure 5.12 – Scroll bar

The paint region in figure 5.1 might specify a modest 20x20 as its minimum to give just enough space to paint a small icon. For its desired size it might report the size of a modest screen at 400x400. For its maximum size it wants as much screen space as it can get so it would report MAXSIZE x MAXSIZE. This would allow the paint region to grow and shrink as the window was resized.

Just because a widget has expressed a minimum or maximum size does not mean that its container will not violate those requests. A widget must be prepared to deal with whatever bounds it is given. For example if a button is too small it may truncate its text or use a smaller font. If the button bounds are larger than the button's maximum size it might draw its bevel around the entire bounds and center its text.

Simple container layouts

So far we have discussed the size requirements of widgets that have no children. We actually build designs by collecting widgets together in containers. The simplest containers are the vertical and horizontal stack. In InterViews and Java these are referred to as Box widgets. The menu in figure 5.13 is an example of a vertical stack and the toolbar in figure 5.14 is an example of a horizontal stack. Figure 5.15 shows a schematic of a horizontal stack. We can use this to illustrate the layout of a stack.

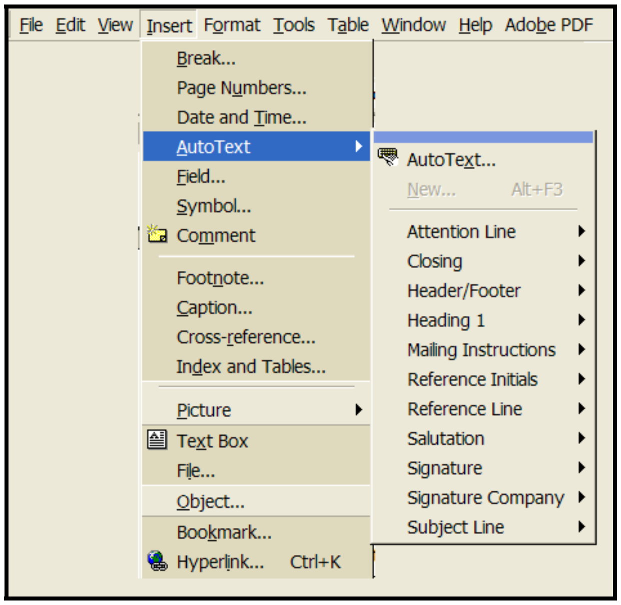


Figure 5.13 – Menu layout

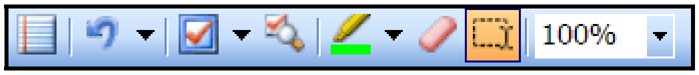


Figure 5.14 – Toolbar layout

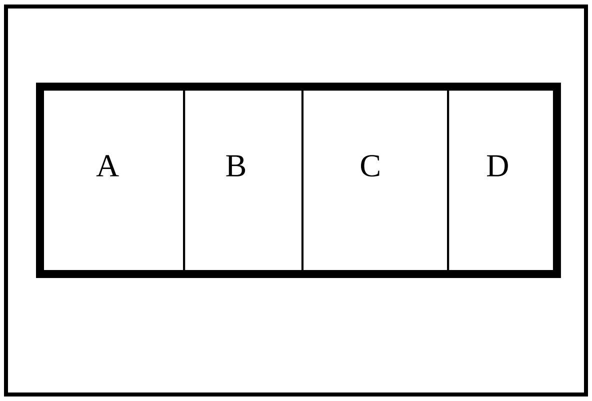


Figure 5.15 – Horizontal stack

The size methods for this stack can be implemented as follows. The width is the sum of the children’s widths and the height is the maximum of their heights.

By exchanging width and height we can implement a vertical stack. The algorithms for the intrinsic sizes of a horizontal stack are in figure 5.16.

```
public class HorizontalStack
{
        public Dimension getMinSize()
        {       int minWidth=0;
                int minHeight=0;
                foreach child widget C
                {       Dimension childSize = C.getMinSize();
                        minWidth += childSize.width;
                        if (minHeight<childSize.height)
                        {       minHeight=childSize.height; }
                }
                return new Dimension(minWidth,minHeight);
        }
        public Dimension getDesiredSize()
        {       similar to getMinSize using C.getDesiredSize() }
        public Dimension getMaxSize()
        {       similar to getMinSize using C.getMaxSize() }
}
```

Figure 5.16 – Size reporting for a horizontal stack

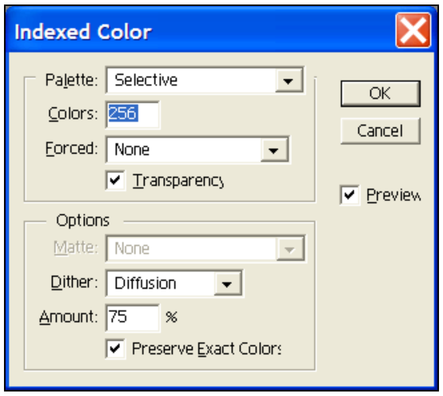
The doLayout() method for a horizontal stack has a number of cases depending upon the bounds the stack receives. It is common when a new window is opened to request the desired size of the root widget and allocate that as the window size. However, there may not be enough screen space for a window that big. The way in which space is allocated among the children depends upon whether the width is less than min, greater than min but less than desired, greater than desired but less than max, or greater than max. The idea of the algorithm is to give each child as much as possible and then divide up the remainder proportionally among the children according to their requests. The layout algorithm for a horizontal stack is shown in figure 5.17.

```
public class HorizontalStack
{
        . . . the other methods and fields . . .
        public void doLayout(Rectangle newBounds)
        {       Dimension min = getMinSize();
                Dimension desired = getDesiredSize();
                Dimension max = getMaxSize();

                If (min.width>=newBounds.width)
                {       // give all children their minimum and let them be clipped
                        int childLeft=newBounds.left;
                        foreach child widget C
                        {       Rectangle childBounds = new Rectangle();
                                childBounds.top=newBounds.top;
                                childBounds.height=newBounds.height;
                                childBounds.left=childLeft;
                                childBounds.width= C.getMinSize().width;
                                childLeft+=childBounds.width;
                                C.doLayout(childBounds);
                        }
                }
                else if (desired.width>=newBounds.width)
                {       // give min to all and proportional on what is available for desired
                        int desiredMargin = desired.width-min.width;
                        float fraction= (float)(newBounds.width-min.width)/desiredMargin;
                        int childLeft=newBounds.left;
                        foreach child widget C
                        {       Rectangle childBounds=new Rectangle();
                                childBounds.top=newBounds.top;
                                childBounds.height=newBounds.height;
                                childBounds.left=childLeft;
                                int minWidth=C.getMinSize().width;
                                int desWidth=C.getDesiredSize().width;
                                childBounds.width=minWidth+(desWidth-minWidth)*fraction;
                                childLeft+=childBounds.width;
                                C.doLayout(childBounds);
                        }
                }
                else
                {       // allocate what remains based on maximum widths
                        int maxMargin = max.width-desired.width;
                        float fraction= (float)(newBounds.width-desired.width)/maxMargin;
                        int childLeft=newBounds.left;
                        foreach child widget C
                        {       . . . Similar code to previous case . . .
                        }
                }
        }
}
```

Figure 5.17 – Layout algorithm for horizontal stack

Using combinations of vertical and horizontal stacks we can produce a variety of layouts. For example figure 5.18 is composed of a horizontal stack that contains a vertical stack of the Palette and Options containers and the vertical stack of OK, Cancel and Preview widgets. The Options container is a vertical stack. The Dither is actually a horizontal stack of the label “Dither” and a combo-box that selects the dither mode.



Palette: Selective
Colors: 256
Forced: None
✓ Transparency
✓ Preview
Matte: None
Dither: Diffusion
Amount: 75 %
✓ Preserve Exact Colors

Figure 5.18 – Layout composed of stacks of stacks

One of the problems that we have glossed over in our layout algorithm is the efficiency of computing the desired sizes. The problem is in the recursive use of the algorithm. In figure 5.18 the window will call the root widget to ask for its desired size. The root widget will recursively call all of its children to compute that size. When the root widget starts to do its layout it will call the Palette group’s desired sizes again. When the Palette group starts its own layout it will call the Colors group’s desired sizes yet again and finally when the Colors group does its layout it will call desired sizes on the text box. If one counts carefully, the desired size methods on the Colors text box was called at least 4 times. This is true of almost every widget in figure 5.18. With a more complex structure as in figure 5.1 the widgets will be called many more times. A second problem is that the size of a window may be changed many times but the desires sizes of the various widgets do not change often. To resolve this, most widget systems will cache their desired sizes rather than recompute them. The problem then arises when a widget really does change size. For most such toolkits an invalidate() method is provided that informs the parent widget that the desired sizes are no longer correct. To accommodate this, our horizontal stack widget’s code would change as in figure 5.19. Note that the invalidate() method not only sets its own sizes to be invalid, but also sends the invalidate() message to its container. The size change may propagate all the way up. Java/Swing, however, does not automatically propagate invalidate() up the tree, which causes interfaces not to change when they might be expected to. This can be remedied by overriding invalidate() in a subclass and calling invalidate() on both the super class and the parent container.

```
public class HorizontalStack
{
    private Dimension minSize;
    private Dimension desiredSize;
    private Dimension maxSize;
    private boolean sizesAreValid;
    public Dimension getMinSize()
    {   if (sizesAreValid)
            return minSize;
        int minWidth=0;
        int minHeight=0;
        foreach child widget C
        {   Dimension childSize = C.getMinSize();
            minWidth += childSize.width;
            if (minHeight<childSize.height)
            {   minHeight=childSize.height; }
        }
        sizesAreValid=true;
        return new Dimension(minWidth,minHeight);
    }
    public Dimension getDesiredSize()
    {   similar to getMinSize using C.getDesiredSize() }
    public Dimension getMaxSize()
    {   similar to getMinSize using C.getMaxSize() }
    public void invalidate()
    {   sizesAreValid=false;
        if (myContainer!=null)
            myContainer.invalidate();
    }
}
```

Figure 5.19 – Desired size code with caching

Spatial arrangement with intrinsic size layouts

Simple stacks are not generally enough to handle all of our layout issues. Sometimes we want items centered. Sometimes we want items grouped at the bottom or top of a list or possibly both. Sometimes we want to add extra space as between the Cancel and Preview widgets in figure 5.18. We can do this with “spreaders” and “spacers”. In InterViews they were called “glue” and in Java they are the Box.Filler class. These are special widgets that do no drawing at all. They are invisible. However, they do have min, desired, and max sizes. For example, suppose we wanted 5 pixels of space between two widgets in a list. We could create a spacer of 5 pixels, which is a widget whose min, desired, and max sizes are all 5. We could add this to the list between the two widgets we want to separate. It looks to the layout algorithm like it is a widget and gets allocated the 5 pixels, thus creating the space, but otherwise it does nothing.

If we want to push all widgets to the top of a list as on the right side of figure 5.18, we can end the list with a spreader widget that reports a very small min and desired size but reports a very large maximum height. All of the other widgets would get their desired sizes, but the spreader, because of its large maximum height, would take everything else and thus push the other widgets up to the top. Putting the spreader first in the list would push everything to the bottom. We can center a widget horizontally by putting spreaders with large but equal maximum widths on both sides of the widget to be centered. They would compete equally for space and thus move the widget to the center. Changing the relative magnitudes of their maximum widths could adjust the widget off of center if desired. The use of spreaders and spacers is effective but not very intuitive for new designers.

Layout Managers

In the original InterViews system, container widgets carried their own layout algorithms. Thus one used a HorizontalStack or VerticalStack widget as a container to perform the desired layout. The problem with this is that container widgets are frequently the place where the view/controller code is placed. It is inconvenient to change the class of the container whenever it is desired to change the layout. Java handled this by creating the concept of *layout managers*. Layout managers are separate objects that handle a particular style of layout. The desired layout for a container now becomes a property that can be set rather than a change to the class hierarchy. All container widgets inherit the same standard layout code. The approach is shown in figure 5.20.

```
public interface LayoutManager
{
        public Dimension getMinSize(Widget containerWidget);
        public Dimension getDesiredSize(Widget containerWidget);
        public Dimension getMaxSize(Widget containerWidget);
        public void doLayout(Rectangle newBounds, Widget containerWidget);
}
public class Widget
{
        . . . other methods and fields . . .

        private Dimension minSize;
        private Dimension desiredSize;
        private Dimension maxSize;
        private boolean sizesAreValid;

        private LayoutManager myLayout;
        public LayoutManager getLayoutManager() { return myLayout; }
        public void setLayoutManager(LayoutManager newLayout )
        {       myLayout=newLayout;
                invalidate();
        }

        public Dimension getMinSize()
        {       if (sizesAreValid)
                        return minSize;
                minSize=myLayout.getMinSize(this);
                sizesAreValid=true;
        }
        public Dimension getDesiredSize() { . . . similar to getMinSize() . . . }
        public Dimension getMaxSize() { . . . similar to getMinSize() . . . }
        public void doLayout(Rectangle newBounds)
        {       myLayout.doLayout(newBounds,this); }
}
```

Figure 5.20 – Layout algorithm with a LayoutManager

Note that setting the layout manager also calls invalidate() so that all of the sizes will be recalculated and all of the containers of this widget will know that their sizes are also invalid and need recomputation. LayoutManager is implemented as an interface so that a variety of objects can serve as layout managers. Note that all of the LayoutManager methods have an additional parameter for the container being laid out. This is so that the layout manager will have access to the children of the container to actually do the layout.

The code for laying out vertical and horizontal stacks can now go in special layout manager classes rather than in containers. In addition there are other layouts that can be created. The FlowLayout will arrange as many widgets as possible in a horizontal row and then start a new row as with the left hand menu in figure 5.1. Some layout managers require an additional location property on the child widgets. The location property is a string or other object that gives information about where the child should be placed in its container's layout. One example is the BorderLayout found in Java/Swing. This layout, shown in figure 5.21, captures the most popular window organization with palettes and buttons around the outside and a large work area in the center. For this layout to work, each child must have a location property of "north", "south", "east", "west" or "center".

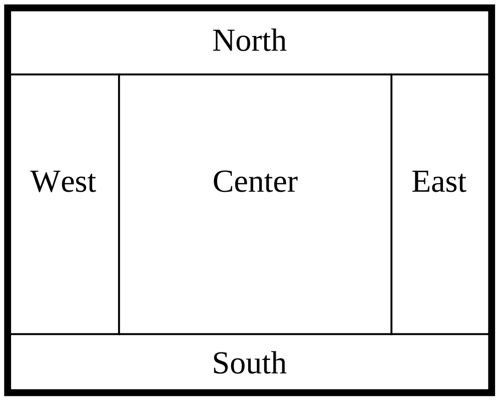


Figure 5.21 – Border layout

Layout Summary

The fixed and edge-anchored layouts place widgets by defining a rectangle location for each widget and then possibly tying that rectangle to the edges of its container. These are good for interactive drawing of widget layouts. The variable intrinsic size approach is more dynamic. We define a structure in which the widgets are placed and then let that structure make the choices. This mechanism allows for more extensive changes to be made to the interface content while retaining a reasonable layout for all of the components. However, variable intrinsic size can be frustrating to designers because widgets do not stay in place. They move depending on the needs of other widgets.

Constraints

The most common geometric problems in user interfaces are widget layouts. However, there are other kinds of geometry and we need mechanisms that support a consistent model of the geometric relationships between various parts of the model and the view. One such representation system is constraints.

A *constraint system* is a set of equations that relate important variables from the model, view and controller. A *constraint* is a single equation that relates some of the variables. It is sometimes helpful to separate the *static constraints*, those that must hold all of the time, and the *situation constraints*, those that only hold for the particular problem we are trying to solve. For example with the scroll-bar in figure 5.12 there are static constraints that define how min, current, and max values relate to the position of the slider. These constraints must always hold. However, when the user is dragging the slider there are situation constraints on the position of the slider, the min and the max values. There is no constraint on the current value because we want to solve for that value. A different situation would be when the max value is changed. We would place situation constraints on the min, max and current values because we know them. We would not have a constraint on the position of the slider because we would not want that to move. We would also not have a constraint on the mouse position because the mouse is not involved when the max property is set. We will look at static and situation constraints in more detail in our examples.

A system of constraints can be *under constrained*. This is when there are fewer constraints than there are variables. We generally handle these situations by solving the constraints that we know and then letting the other variables retain the values they had before. A system can be *over constrained* when there more equations than variables. It is common to handle this situation by defining constraint priorities. That is we solve the highest priority constraints first and then ignore any constraints left unresolved.

Example constraint systems

Constraints are best understood in the context of some examples. We will first look at constraints for performing widget layout. We will then look at a scroll-bar and a meter dial as representatives of more complicated constraints.

Layout constraints

The layout constraint problem is usually modeled as a system of constraints with only one situation. That is the bounds of the layout have been set and we need to compute the placement of the children. Figure 5.22 shows a container widget with three child widgets. Various values have been labeled that will be important to modeling their layout using constraints. The containing rectangle we will call **P** for parent.

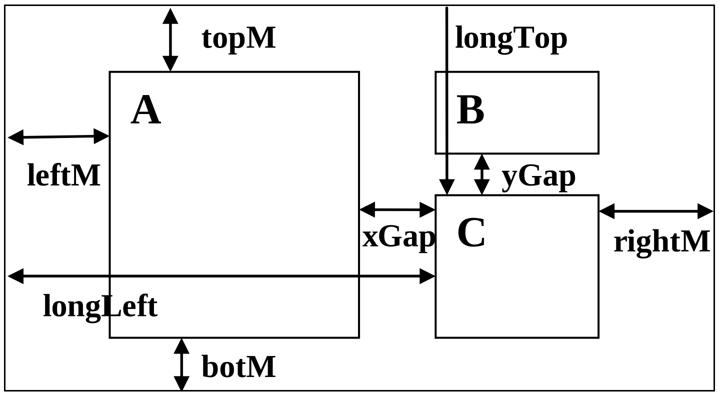


Figure 5.22 – Layout by constraints

The simple fixed layout system defines the position of **A, B** and **C** relative to the top left corner of **P**. The system constraints for fixed layout are shown in figure 5.23.

```
leftM=10                longLeft=120
topM=7                  longTop=40

A.width=100             A.height=150
B.width=60              B.height=25
C.width=60              C.height=120

A.left=P.left+leftM     A.right=A.left+A.width
A.top=P.top+topM        A.bot=A.top+A.height

B.left=P.left+longLeft  B.right=B.left+B.width
B.top=P.top+topM        B.bot=B.top+B.height

C.left=P.left+longLeft  C.right=C.left+C.width
C.top=P.top+longTop     C.bot=C.top+C.height
```

Figure 5.23 – System constraints for fixed layout

The first two sets of constraints are *design constraints*. These are the values set when the user draws the layout of these three widgets. The situation constraints for this system are constant values for **P**.left and **P**.right. Solving this system of constraints is trivial. We simply set the location of **P** and then evaluate each constraint in the order shown in figure 5.23. Because there is only one situation, this evaluation order always works. Note that this system of constraints does not use botM, rightM, xGap or yGap. P.right and P.bot are also ignored. Using fixed layout, there is no way to take any of these into consideration.

We can modify this system of constraints to use all of the edges of the bounds. The constraints in figure 5.24 are edge-anchored constraints.

```
leftM=10                longLeft=120
topM=7                  longTop=40
rightM=12               botM=7

A.width=100             A.height=150
B.width=60              B.height=25
C.width=60              C.height=120

A.left=P.left+leftM     A.right=A.left+A.width
A.top=P.top+topM        A.bot=P.bot-botM

B.left=P.left+longLeft  B.right=P.right-rightM
B.top=P.top+topM        B.bot=B.top+B.height

C.left=P.left+longLeft  C.right=P.right-rightM
C.top=P.top+longTop     C.bot=P.bot-botM
```

Figure 5.24 – System constraints for edge-anchored layout

This system of constraints is solved in the same fashion as edge anchored. We set the value P.left, P.right, P.top, P.bot and then evaluate in top to bottom order. When P changes size, A.bot and C.bot will stay a constant distance from the bottom of the parent. In addition, B.right and C.right will move as P.right moves.

This is a better layout, because it makes use of the size of P, but it is still not what we want because A is rigid in its width and B is rigid in its height. We also do not have the kind of control that we would like. What we really want is for B and C to stay a fixed distance from A no matter what the size of A may be. We also want C to be a fixed distance below B no matter what the size of B. In addition to all of this we would like the gaps to move proportionally relative to the size of the parent giving each widget its share of the space. One way to do this is with variable intrinsic size layouts. We could put B and C into a vertical stack with a spacer for yGap, then put that stack into a horizontal stack with A and another spacer for xGap. This whole package can be placed in other stacks with other spacers for leftM, topM, rightM and botM.

This combination of stacks and spacers will work, but it seems counter-intuitive. Figure 5.25 shows a system of constraints that will accomplish the same thing. The widths and heights of A, B and C are now used to compute two new variables xProp and yProp which are the fractional distance of A.right between the horizontal edges and the fractional distance of B.bot between the vertical edges. These in conjunction with xGap and yGap define our layout. As P gets wider A.right will move proportionally to the right and widgets B and C will stay a constant distance away. The design is manipulated by changing the constants for the various gaps and margins.

```
leftM=10                xGap=10
topM=7                  yGap=5
rightM=12               botM=7

A.width=100             A.height=150
B.width=60              B.height=25
C.width=60              C.height=125

xProp=(leftM+A.width)/(leftM+A.width+xGap+B.width+rightM)
yProp=(topM+B.height)/(topM+B.height+yGap+C.height+botM)

A.left=P.left+leftM     A.right=xProp*P.width
A.top=P.top+topM        A.bot=P.bot-botM

B.left=A.right+xGap     B.right=P.right-rightM
B.top=P.top+topM        B.bot=yProp*P.height

C.left=A.right+xGap     C.right=P.right-rightM
C.top=B.bot+yGap        C.bot=P.bot-botM
```

Figure 5.25 – Extended layout constraints

Constraints such as these are very powerful in providing explicit control over layout positions. However, they are problematic as a design tool. The difficulty lies in presenting an interactive layout tool that provides clear control over the constraints. Luca Cardelli proposed a mechanism for drawing layout constraints similar to those in figure 5.25. In his system each widget was drawn into position inside of a parent widget. Each edge of a widget had a *constraint handle*, represented as a small circle. If the user left the constraint handle undefined, then that edge was placed proportionally between the corresponding edges of the parent. The proportion was determined by where the edge had been drawn in the design. The constraint handle could also be connected to any edge of any other widget. This would create a constant distance constraint between those edges. Figure 5.26 shows a Cardelli-style design that is consistent with the constraints in figure 5.25. Since the Cardelli work there have been systems such as Apogee[6] that allow designers to define invisible guidelines similar to what draftsmen do to create alignments. Apogee also introduced *maximum constraints* that use the maximum values of other constraints. This allows consistent alignments relative to widgets of varying sizes. Other systems introduced squiggly lines to represent spring or spreader style constraints that expand to fill whatever space is available. All of these systems resolve to a set of constraint equations like those we have been discussing.

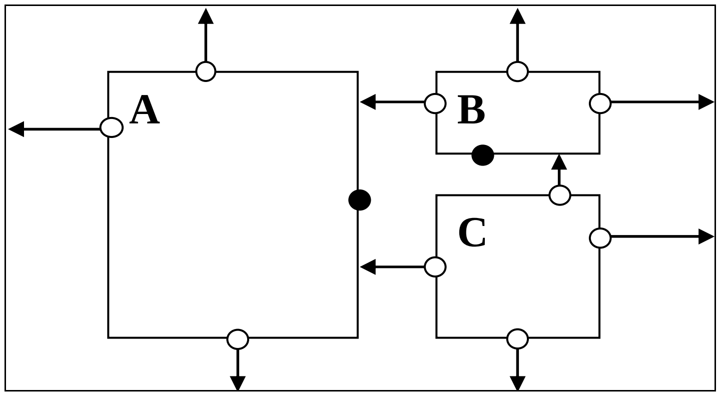


Figure 5.26 – Layout design with Cardelli constraints

Interactive constraints

In layout constraint systems the size and location of the parent rectangle are specified as the only situation constraints and all other positions are directly or indirectly derived from those values using other constraints. There are, however, interactive constraints that connect the geometry of a widget's view and controller to the values in the widget's model. Changing parts of the model or dragging parts of the geometry all cause updates to occur. The system constraints preserve the consistency and the situation constraints define what is known for a particular interactive task.

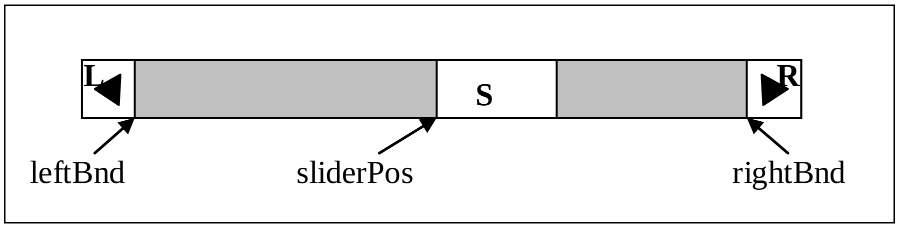


Figure 5.27 – Scroll-bar geometry

As our first example, consider the scroll-bar shown in figure 5.27. There is an enclosing rectangle for the scroll-bar called P and three rectangular regions within the scroll-bar called L, S and R. In addition, we have defined three variables leftBnd, sliderPos and rightBnd. In our constraints we will define other variables to simplify their creation. The goal of a scroll-bar is to present and manipulate a model. The model for our scroll-bar is shown in figure 5.28. This particular scroll-bar is intended to scroll a window across a larger region. We want the width of the slider to reflect the width of the window. Therefore we have added the windowWidth value to our model to represent how much of the area between min and max is actually being displayed.

```
Scroll-bar model
    int min;
    int windowLeft;
    int max;
    int windowWidth;
```

Figure 5.28 – Scroll-bar model

Our system constraints for the scroll-bar are shown in figure 5.29. A most common situation for this set of constraints is when the model is known and P is known. From this information we need to compute the geometry of the view. This would occur whenever there is a model change or whenever essential geometry is required. Formally we would represent this situation by providing constant constraints for all of the known values. In figure 5.29 we have simply placed known constant variables in bold face. In any constraint where the value of a variable was computed by a previous constraint, we underline that variable.

```
L.left=P.left
L.top=P.top
L.right=P.left+P.height        // makes L square
L.bot=P.bot

R.right=P.right
R.top=P.top
R.left=P.right-P.height        // makes R square
R.bot=P.bot

windowLeftRange=max-min-windowWidth   // window moves between min and max but
                                      // the left does not move all the way to max
leftBnd=L.right+1
rightBnd=R.left-1

S.width = (windowWidth*(rightBnd-leftBnd)/(max-min))
                               // compute slider width proportional to window width
S.top=P.top
S.bot=P.bot
sliderLeftRange=rightBnd-leftBnd-S.width
(S.left-leftBnd)/(rightBnd-leftBnd-S.width) = (windowLeft-min)/(max-min-windowWidth)
```

Figure 5.29 – System constraints for scroll-bar

Looking carefully at figure 5.29 we see that by evaluating these constraints from top to bottom, each constraint has only one unknown variable, given the constant constraints of the situation and previously compute variables. We thus have a simple constraint solution process. There is a challenge on the last constraint. This is not a simple assignment. This constraint specifies the relationship between the slider position and the windowLeft position. In the constant model situation, we would solve for S.left because all other variables will be known when we reach this constraint.

When the user drags the slider using the mouse, the final constraint is in a different situation. The S.left is known because of the mouse position. The windowLeft is not known because we need to compute that value from the mouse position. In the slider dragging situation, we solve for windowLeft rather than S.left. The system constraints are the same and retain the relationships across all situations. The difference is in the situation constraints that provide our initial constants.

A geometrically more complicated widget is shown in figure 5.30. Here we have a meter whose needle shows a cur value between min and max, with min being displayed to the left and max to the right. Our other variables are the points C, where the center is, N where the needle meets the edge of the meter, M where the mouse is dragging the meter needle, R where the max point of the meter arc should be and L, which is a construction point we will need.

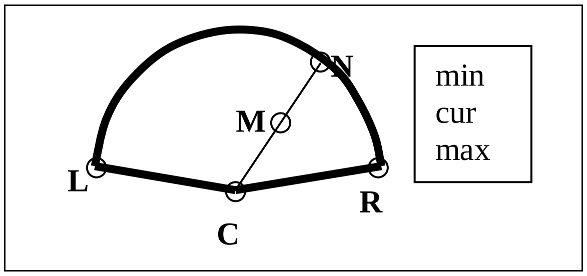


Figure 5.30 – Meter widget geometry

In this widget many of the variables are points and in some cases the constraints are defined as equations on points. Note that a point is really two variables and an equation involving points is actually two equations, one for X and one for Y. Figure 5.31 shows a possible set of system constraints for the meter widget. The constraints in figure 5.31 are designed to mathematically capture the relationships among all of the points and the model. They were not necessarily designed to evaluate the constraints for any particular situation. These constraints are also not all linear.

1. radius = sqrt((R.x-C.x)^2+(R.y-C.y)^2) // establish L, N and R at same distance from C
2. radius = sqrt((L.x-C.x)^2+(L.y-C.y)^2)
3. radius = sqrt((N.x-C.x)^2+(N.y-C.y)^2)
4. L.y = R.y

5. (M.x-C.x)/(N.x-C.x) = (M.y-C.y)/(N.y-C.y) // establish colinarity of C, M and N

6. cos(angleR)=(R.x-C.x)/radius
7. cos(angleL)=(L.x-C.x)/radius
8. cos(angleN)=(N.x-C.x)/radius

9. (cur-min)/(max-min) = (angleN-angleL)/(angleR-angleL) // relate model to geometry

Figure 5.31 – System constraints for meter widget

Our first situation is where the model, C and R are known. What we need is the necessary information to draw the meter. To do this we will need to compute point N to draw the needle as well as radius, angleR and angleL to draw the meter arc. We start with the situation constraints that give constant values to min, cur, max, C.x, C.y, R.x, and R.y.

- Constraint 1 give us radius from C and R.
- Constraint 4 can give us the value of L.y.
- Solve constraint 2 for L.x.
- Solve constraint 6 for angleR.
- Solve constraint 7 for angleL.
- Solve constraint 9 for angleN.
- Solve constraint 8 for N.x.
- Solve constraint 3 for N.y.

For this situation we used an algorithm known as the propagation of known states[7]. This algorithm will provide an ordering to the constraints that the variable that must be solved for each constraint. Given the situation constraints there are several variables that are automatically known. We search our list for a constraint that has exactly one unknown variable. We solve the constraint for that variable and then add the solved constraint to our list of solutions. We mark the solved variable as known and continue.

This algorithm stops when there are no remaining constraints with exactly one unknown. If there are still unknown variables then either the system is underconstrained or simultaneous solutions are required. In this particular system we avoided solving for the intersection of the needle line equation and the arc, which would have involved simultaneous equation solutions. The constraint system in figure 5.31 is underconstrained because we cannot solve for M. However, we do not need M in this situation and therefore we do not care.

A second situation for our meter widget is when min, max, C, R and M are known. This occurs when we use the mouse to set the needle position and thus change the value of cur.

- Solve constraint 1 for radius.
- Solve constraint 4 for L.y.
- Solve constraint 2 for L.x.
- Solve constraint 7 for angleL.
- Solve constraint 6 for angleR.
- Simultaneously solve constraints 3 and 5 for point N.
- Solve constraint 8 for angleN.
- Solve constraint 9 for cur.

This situation demonstrates the need to solve simultaneous equations and some of the challenges of working with constraints. We could have escaped the simultaneous equation problem by adding constraints that tied M to the model rather than to N. The mouse input situation could then have been solved by simple propagation of known states.

We have also ignored the situation where there are multiple solutions to a constraint. Constraints 1-3 are quadratic and may have zero, one or two solutions. A simple technique in the case of multiple solutions is to choose the solution closest to the previous value of the variable. This works in many cases because interaction is generally incremental, but not in all cases. Another alternative is to add inequality constraints that restrict possible solutions.

Constraint solution techniques

In the preceding section we have seen how constraints can be used to model the relationships between widgets and portions of a view. A set of constraints is not the same as executable code that we can put into widgets. At one level we can use the constraints as a design tool and then write code from the constraints. This is a useful exercise where there is a tight relationship between geometry and the model because it is very easy to get these issues wrong and produce a hideous tangle of code. Some researchers have pursued the creation of constraint systems with automatic techniques for solving them.

There are three major techniques for solving systems of constraints. There are the iterative techniques derived from numerical analysis and optimization theory. These techniques are sometimes used for graph layout problems but are generally too slow for use as an integral part of most user interfaces. However, increasing processor speed may remove that barrier. There are the symbolic equation solvers using techniques from programs like MatLab[8], Maple[9] and Mathematica[10]. These applications are designed as automatic aids for mathematicians, but most have the nice property of generating C or Java code from their resulting solutions. They can be useful in developing a solution by hand for a particular widget design problems.

In user interface work the goal is usually to convert the constraints of a particular situation into a sequence of program statements that will compute the solution. There are two parts to this problem. The first is to solve a particular constraint for a particular variable. The second is to plan or order the constraints so that all variables except the one being solved for are known before a constraint is evaluated.

One-way constraints

A popular set of constraints are the one-way constraints. In such systems every constraint has the form $y=f(x_1,x_2,...)$. Such a constraint maps directly to an assignment statement. Figure 5.25 is an example of a one-way constraint system. With such a system of constraints we only need to order them so that for each constraint all of the arguments $(x_1,x_2,...)$ have been computed before the constraint is evaluated. Because one-way constraints are already in the form of an assignment statement there is no automatic algebra to be performed. One-way constraints are the basis for spreadsheets. The early spreadsheet systems used the fixed point algorithm to iterate over all constraints until there is no change. This is simple to implement but not very efficient. The propagation of known states algorithm described earlier will produce an efficient solution to a set of one-way constraints.

One-way constraints are also found in attribute grammars[11] from compiler theory. Attribute grammars are designed to propagate semantic information through the parse tree of a program fragment. This concept of propagation through a tree also applies to hierarchic visual models such as widget trees and other models that we will see.

Incremental one-way constraints

There is also an efficient algorithm for incremental evaluation of one-way constraints. In interactive settings generally one or two variables are changed. What we want is an efficient algorithm for evaluating only those constraints that must be evaluated to be consistent with the change. In the simple constraint systems that we have looked at, this is not an issue, but in more complex systems involving hundreds to thousands of objects with many interconnecting constraints this can be a serious issue.

A simple incremental algorithm is recursive evaluation of all affected constraints. Figure 5.32 shows an algorithm to update all necessary constraints when some variable C is changed. When a variable is changed, all constraints that use that variable are changed and they must be updated also. This updating propagates recursively until all changes have been recomputed.

```
{ . . .
    updateVariable(C);
}
public void updateVariable(Variable V)
{
    For each constraint C where V appears as an argument
    {
        C.evaluate();
        updateVariable(C.result);
    }
}
```

Figure 5.32 – Simple incremental constraint evaluation

This recursive incremental evaluation has serious problems. If some variable is used in multiple constraints and those results are then used in multiple constraints, a given constraint may be evaluated many times based on all of the ways in which values could have changed. There may also be partial evaluations because some of the argument changes will propagate later through different paths. In the extreme this algorithm will evaluate a constraint an exponential number of times.

A more efficient algorithm is based on an incremental attribute flow algorithm[12]. In this algorithm, shown in figure 5.33, every variable has a value and in addition a boolean "known" flag to identify that this variable has a known value. Using this flag, a more efficient two pass algorithm can be designed that computes only the necessary constraints and only computes them once. Figure 5.33 shows a revised algorithm for updating variable C. This algorithm first propagates the "unknown" state through the constraint system to mark all variables to be changed as unknown. The pass that updates the variables will only update a variable if all arguments are known. Thus the actual evaluation of a constraint is put off until it has all changes, not just the first one.

```
{ . . .
        markUnknown(C);
        assign new value to C
        updateVariable(C);
}
public void markUnknown(Variable V)
{
        if (!V.known) return
        V.known=false;
        For each constraint C where V appears as an argument
        {       markUnknown(C.result); }
}
public void updateVariable(Variable V)
{
        V.known=true;
        For each constraint C where V appears as an argument
        {       if (for all arguments A of C, A.known is true and C.known is false)
                {
                        C.evaluate();
                        updateVariable(C.result);
                }
        }
}
```

Figure 5.33 – Efficient incremental constraint evaluation

Hudson[13] observed that in user interfaces, the existence of scrolling, zooming and other techniques means that only a fraction of the geometry of many widgets is on display at any one time. Rather than recompute all constraints affected by a change we should only recompute those that are actually visible. This created a "push-pull" algorithm where the model would "push" changes through the system as in figure 5.33 while the view would "pull" visible values. This algorithm is shown in figure 5.34.

```
{  . . .
        markUnknown(C);
        assign new value to C
        For each visible variable V
        {       computeValue(V); }
}
public void markUnKnown( Variable V)
{       . . . as in figure 5.33 . .  }
public void computeValue(Variable V)
{
        if (V.known) return;
        C=the constraint that will compute V;
        For each argument A of C
        {       computeValue(A); }
        C.evaluate();
        V.known=true;
}
```

Figure 5.34 – "Push-pull" incremental constraint evaluation

The "push" part of the algorithm in markUnknown() will mark all changes as unknown. However, the "pull" part in computeValue() will only recompute values as they are actually needed. Values that are not needed are not computed and remain marked as unknown. The "push" will not reenter those constraints because they are already unknown and they will never be visited again until there is some need. When scrolling or some other change of the view causes new values to be exposed, the view should call computeValue() on them. If they have been changed, the new values will be computed. If they have not, then nothing is done. This is a very efficient model for managing update of change.

Multi-way constraints

The biggest disadvantage of one-way constraints is that they are one way while interaction is inherently two-way. Sometimes the model changes and the view must update and sometimes the controller changes the geometry of the view and the model must update. One-way constraints do not capture these multidirectional changes.

In figure 5.35 we show a constraint taken from figure 5.31 that describes the relationship between the angle of the meter needle and the model variables. Algebraically we could solve for any of the variables in this constraint. However, we do not want to code up an algebra solver and we really only need two solutions. We need to solve for angleN when the model has changed and we need to move the needle, and for cur when the needle has moved and we want to update the model. In figure 5.35 we augment the general constraint with two one-way constraints. Each of these is identical to the original constraint. In different situations different variables will be known and either of these two variants can be computed. What we have done is use a human algebra solver to give us something that our propagation of known states algorithm can readily use.

```
(cur-min)/(max-min) = (angleN-angleL)/(angleR-angleL)

        cur=min+(max-min)*(angleN-angleL)/(angler-angleL)
        angleN=angleL+(angleR-angleL)*(cur-min)/(max-min)
```

Figure 5.35 – Using multiple one-way constraints

Simultaneous constraints

Most of the work involving simultaneous constraints uses iterative solutions. Borning[14] reports a very efficient algorithm using linear programming to solve a variety of interactive problems. Juno[15] constructed many geometric relationships using the standard constraints from compass/straightedge geometry. This work showed a variety of relationships with iterative numerical solving. Olsen and Allan[16] observed that most simultaneous equations involved the intersection of pairs of geometric equations. In figure 5.30 the simultaneous constraints solve for point N, given the line of the needle and the circle of the meter boundary. All possible intersections of lines and circles were solved by hand and encoded. The propagation of known states algorithm was extended so that when no constraints were available with a single unknown variable, a pair of constraints with a shared point was found and the simultaneous solution was selected from the set of pre-solved solutions. This created a tool for designing a variety of geometric widgets by drawing their view and their constraints.

Constraint Summary

Constraints are equations that define the relationship between geometric entities in a view, mouse inputs in the controller and model information. They provide a mathematical basis for many layout mechanisms as well as a representation of a variety of view/model problems. Solving a system of constraints involves finding a set of variable values for which all of the constraint equations are true. There are very general numeric solutions, but they are generally too slow for interactive use. The propagation of known states algorithm works from known values looking for constraints that have exactly one unknown variable, solving for that variable and adding the solution to the list of solutions. This produces a pre-solved set of assignment statements that are readily translated into code.

The simplest constraint systems use functional or one-way constraints. Here the constraint equations are already solved, however, for only one variable. Solving the constraint system involves ordering the constraint evaluation so that all arguments are known before they are required. The one-way constraints also have an incremental solution that minimizes the number of constraints that must be evaluated in response to a small change in variable values. By specifying more than one solution for a constraint, the propagation of known states algorithm can be extended to the multi-way constraints required for interaction.

---

[1] Buxton, W., Lamb, M.R., Sherman, D., and Smith, K. C., "Towards a Comprehensive User Interface Management System", *Computer Graphics* 17(3), 1983.

[2] Goodman, D. *The Complete HyperCard Handbook*, New York: Bantam Books, 1987.

[3] Cardelli, L. "Building User Interfaces by Direct Manipulation," *ACM SIGGRAPH Symposium on User Interface Software*, October 1988, p 152-166.

[4] Knuth, D. *The TeXbook*, Addison-Wesley, Reading Mass., 1984.

[5] Linton, M.A., Vlissides, J.M., and Calder, P.R., "Composing User Interfaces with InterViews", *IEEE Computer*, 22(2), Feb 1989, pp 8-22.

[6] Hudson, S. E. "Graphical Specification of Flexible User Interface Displays," *User interface Software and Technology (UIST 89)* ACM Press, New York, NY, (1989) 105-114.

[7] Borning, A. and Duisberg, R. "Constraint-based Tools for Building User Interfaces." *ACM Trans. Graph.* 5, 4 (Oct. 1986), 345-374.

[8] Palm, W. and Palm, W. J., *Introduction to MATLAB 7 for Engineers*, McGraw-Hill, (2003).

[9] Heck, A. *Introduction to Maple*, Springer (2003).

[10] Wolfram, S. *The Mathematica Book*, Wolfram Media (2003).

[11] Aho, A. V., Sethi, R., and Ullman, J. D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, (1986).

[12] Reps, T., Teitelbaum, T., and Demers, A. "Incremental Context-Dependent Analysis for Language-Based Editors." *ACM Trans. Program. Lang. Syst.* 5, 3 (Jul. 1983), 449-477.

[13] Hudson, S. E. “Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update. *ACM Trans. Program. Lang. Syst.* 13, 3 (Jul. 1991), 315-341.

[14] Badros, G. J., Borning, A., and Stuckey, P. J. “The Cassowary Linear Arithmetic Constraint Solving Algorithm.” *ACM Trans. Comput.-Hum. Interact.* 8, 4 (Dec. 2001), 267-306.

[15] Nelson, G. “Juno, A Constraint-based Graphics System.” *Computer Graphics and Interactive Techniques (SIGGRAPH ’85),*

[16] Olsen, D. R. and Allan, K. “Creating Interactive Techniques by Symbolically Solving Geometric Constraints,” *User Interface Software and Technology (UIST ’90)* ACM Press, New York, NY, (1990) 102-107.