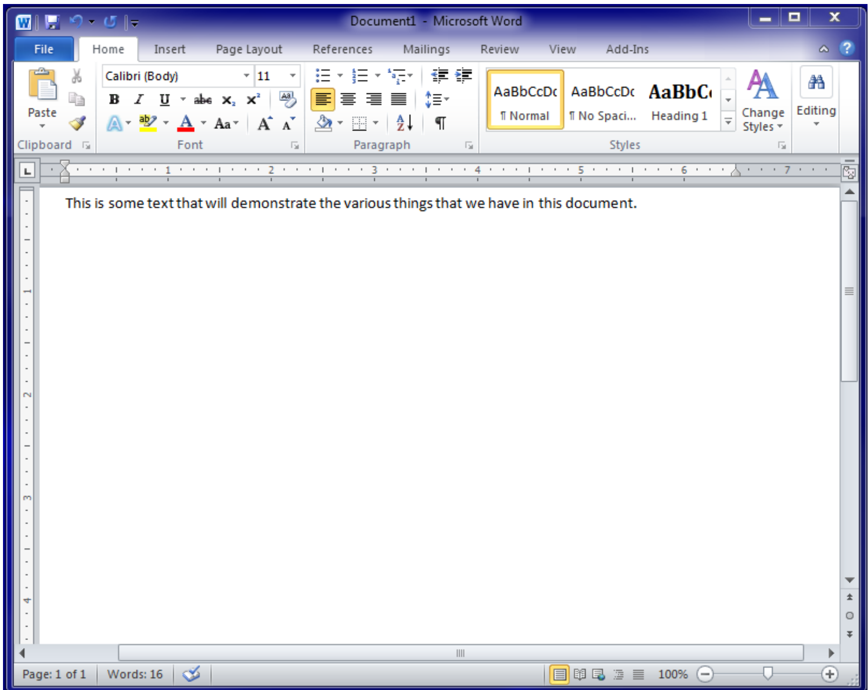
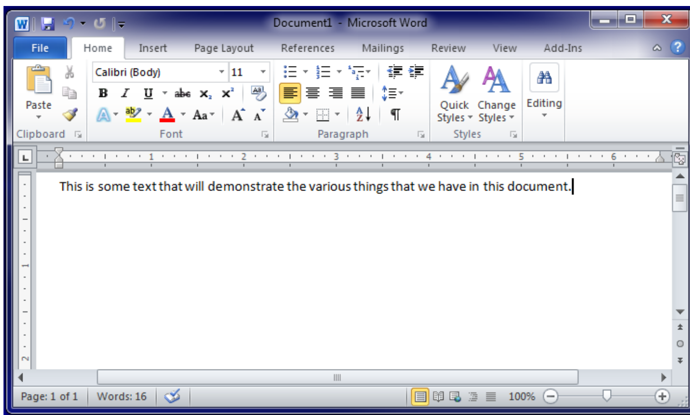
Layout

Up to this point we have directly specified the position of various graphical elements in a widget. This direct specification is easy to build as a system but quite cumbersome to use in practice. Figure 11-1 shows a window in Microsoft Word at a particular size. In figure 11-2 we see the same window at a smaller size. The user has interactively changed the size of the window. If the bar across the bottom had stayed at a fixed location, we would not be able to see it because it would be outside of the window. Instead, its location has changed to reflect the new window size. It still naturally appears across the bottom of the window. Similarly the close button (X) in the upper right has move to a new location. With the narrower window the old location would have been clipped away. If we want widgets to “stick to the boundary” we need to do something other than pin them to a fixed location.

In figure 11-1 we also sty the “Styles” button group in the ribbon across the top. However, in figure 11-2 that “Styles” group has changed a great deal. Instead of the scrolling list of individual styles it has switched to two buttons. What happened is that as widgets were being laid out, there was no longer enough room for the scrolled list of styles and this has been replaced by a more compact representation.



11-1 – Microsoft Word large window



11-2 – Microsoft Word small window

The problem is that window sizes are variable and the positions of widgets within the available space must adapt to that variability. This is the layout problem, which we will address in this chapter. There are a

variety of techniques which each have relative advantages and disadvantages.

Window resizing is not the only cause for layout issues. Personal computing devices have many different screen sizes, from small phones, to large phones, to tablets, to laptops, to multi-screen desktop workstations. We want to write applications that deal with all of these effectively without having to write many different applications for each situation. In portable devices this is further complicated by how the device is held, either vertically (portrait) or horizontally (landscape). The layout must dynamically adapt.

Data contents can also cause layout issues. If we are looking at a list of things, we want that list to dynamically change as items are inserted, deleted or reordered. The things we put in the list may not all be the same size, but we want it to place items correctly. The menu bar and the individual menus are examples of this situation. We just want them to stack up correctly no matter what items have been added or removed.

Styling changes may also cause layout problems. If, for example, we change the font size for menu items they might all get taller and wider. We want the menu to lay out correctly without requiring us to modify menu button positions by hand. Similar things occur when we change languages to internationalize our application. Changing the word “open” in English to “geöffnet” in German will require a wider menu button and may thus move other things around.

Another big factor in choosing a layout model is the design tools. Many of the UI design problems are best handled visually. It is easier to see and modify a layout/design than it is to encode the design, view the results and then modify the code. However, some of these layout models do not lend themselves well to visual manipulation.

Layout algorithm issues

The layout problem is generally cast as a problem of placing nested rectangles. The size and placement of a parent rectangle is decided by the user or the layout algorithm and then the children are arranged within that fixed space. This proceeds recursively to whatever level is necessary.

When a parent rectangle changes size or position, this is well known. The parent rectangle executes the layout algorithm and the children are recursively resized and repositioned. However, sometimes it is the children that cause the layout issue. The label on a menu item might get bigger or smaller which necessitates relayout. A list might get longer. An item might be removed. In such cases the toolkit will provide an invalidate(), layout() or repack() method. This tells an item that its size needs have changed. This is propagated to the parent rectangle and so on up the tree until the top level is reached and the layout algorithm is reexecuted.

Fixed layout

The simplest algorithm is the fixed layout. Every rectangle is given a fixed value for left, top, width and height. If everything stayed constant this algorithm works great. This is the algorithm that most visual designers prefer. It is also the easiest to construct interactive design tools. The positioning of a widget is just a matter of dragging and resizing rectangles. Even nested structures are easy to draw. In most systems the positions of the rectangles are defined relative to the position of their parent rectangle. This simplifies positioning groups of widgets as a unit.

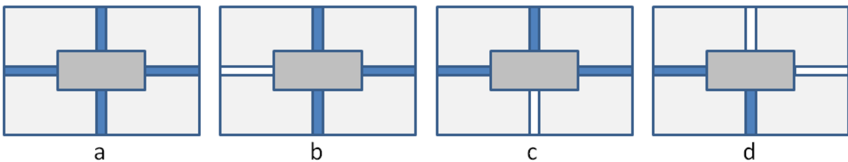
A key reason why visual designers like fixed layout is the simplicity of the model and the control it gives them. It also conforms to the design skills that they acquired in poster and page design. It allows them to carefully and iteratively enhance a design with the knowledge that everything will appear as they have designed it. However, this is not

interactive reality. People resize, move, interact and change things. We need layout models that can adapt to this dynamic reality.

Edge-Anchored

In figures 11-1 and 11-2 we see two kinds of layout going on. The first are the items that stick closely to their adjacent edge. Many applications use the technique of decorating the borders with buttons, controls and information. The second layout is the heart of the application. In figure 11-1 this is the document being edited. This widget generally occupies the center of the area and grows as large as the parent window will allow. In Visual Studio, Microsoft introduced the edge-anchored layout that slightly extends the fixed layout to accommodate the kinds of layout problems we have just described.

In edge-anchored layouts, the designer positions items with a drawing tool just as with fixed layouts. In addition to the position of the rectangles, each rectangle gets four anchor attributes. These are represented in an anchor control for which examples are shown in figure 11-3.



11-3 – Edge-anchor controls

The anchor control is operated by clicking on the four anchor bars. This will toggle a bar between anchored (dark) and unanchored (light). Anchored means that the corresponding edge of the rectangle is a fixed distance from the corresponding edge of its parent. The distance is determined by the designer when the rectangle is interactively placed in the design. The design flow is to draw widget rectangles wherever they are to be placed and then set the anchor control to manage their position when layout changes.

Control “a” would be the setting for the main document area in figure 11-1. Each of edges of this main area is a fixed distance from the window border. Whenever the window is resized this area takes a position just inside using the offsets specified when the area’s rectangle was placed. The control “b” would be used for the scrollbar on the right side of figure 11-1. The top and bottom of the scrollbar are a fixed distance from the top and bottom of the containing window. The right edge of the scrollbar is tied directly to the right edge of the containing window. There is no anchor for the left edge in control “b”. In this case the width of the rectangle as it was drawn is used to place the left edge. So one anchor means to fix that edge and then use the width.

Control “c” is the same case as control “b” only flipped 90 degrees. This would be used for the ruler bar just above the document area in figure 11-1. It is tacked to the left and right edges and will grow or shrink with the window’s width and then is stuck to the top of the window with a constant height.

Control “d” would be the case for the widget in the lower right of figure 11-1 that contains the text “Page: 1 of 1”. This would have a fixed width and height and would be stuck to the bottom and left of the parent rectangle. This would also be the case for the widget that says “Words:16”. The difference is that the fixed distance from the left edge is larger to account for the other widget. If both edges are unanchored then the rectangle has a constant width or height and floats proportionally between the edges based on where it was drawn.

The algorithm for this layout technique is shown in figure 11-4.

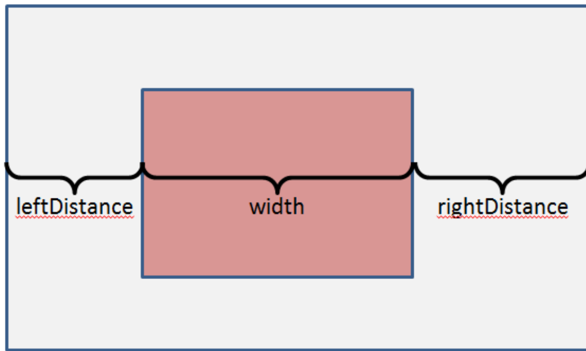
```
public class Widget
{   Rect bounds;
    int left,top, right, bottom;
    int parentDrawWidth, parentDrawHeight;
    boolean anchorLeft, anchorTop,
            anchorRight, anchorBottom;
    public void doLayout(Rect newBounds)
    {   bounds = newBounds;
        foreach C in children
        {   Rect cb = new Rect();
            if (C.anchorLeft)
            {   cb.left=newBounds.left+C.left;
                if (C.anchorRight)
                {   cb.right=newBounds.right-C.right; }
                else
                {   cb.right=cb.left+(C.right-C.left); }
            }
            else if (C.anchorRight)
            {   cb.right=newBounds.right-C.right;
                cb.left=cb.right-(C.right-C.left);
            }
            else // no horizontal anchors
            {   cb.left=C.left*
                    (newBounds.width/parentDrawWidth);
                cb.right=cb.left+C.width;
            }
            . . . .do the same in Y . . .
            C.doLayout(cb);
        }
    }
}
```

11-4 – Edge-anchored layout algorithm

Proportional

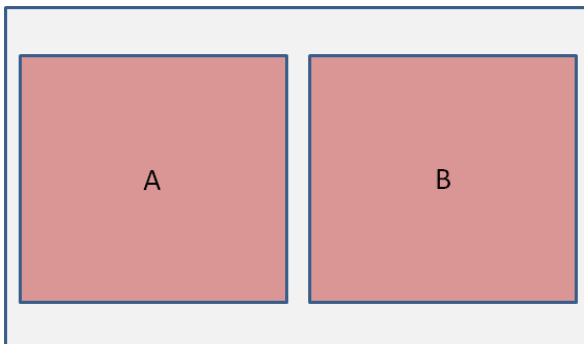
Another layout technique that adapts to window size and is relatively simple is the proportional technique. As with the other techniques the vertical and horizontal layouts are independent of each other and use the same algorithms. We will demonstrate the technique for the horizontal case.

In figure 11-5 we see a widget rectangle embedded in its parent rectangle. The location of the widget is determined by any two of three possible values. When specifying these values, we can either supply an integer pixel value or we can specify a fraction of the parent's width, such as 40%. By specifying a proportional value, the width of the parent determines all or part of the horizontal layout.



11-5 – Horizontal position values

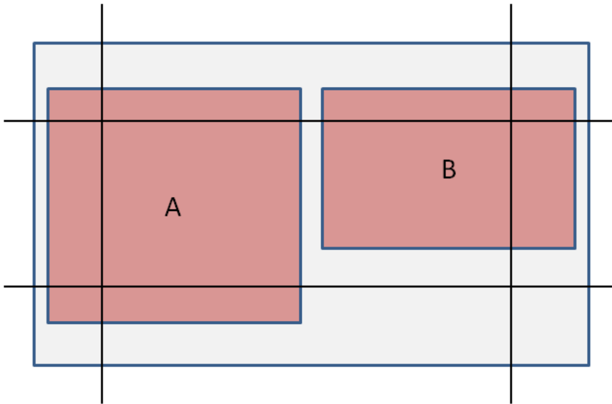
For example in 11-6 might be specified with A being 10 pixels from the left and 48% of the parent width. Widget B would be 10 pixels from the right and also 48% of the parent width. As the parent gets bigger and smaller the two widgets stay stuck to their respective edges and get wider or narrower. It is also possible to use centerLeftDistance and centerRightDistance as additional possibilities. In conjunction with the width these can place a widget relative to its center.



11-6 – Proportional sizes

There are two advantages to this layout technique. The first is that it is easy to implement. Once the parent's dimensions are known it is trivial to calculate the child's position. The second is that it is easy to draw in an interactive design tool. For example we could draw the layout in figure 11-6 and then specify that the widths are proportional rather

than fixed. Based on the drawing, the design application can easily figure out the proportion percentage. One can also use the guides shown in figure 11-7. Any edge that falls between a guide and the edge of the parent is fixed at its drawn distance from the parent. Any edge that is drawn between the guides is place proportionally.



11-7 – Drawing guides for proportionality

In figure 11-7 widget A has its top, left and bottom edges fixed to the parent edges with its right edge floating about 48% of the way between across. Widget B has its top and right fixed with its left and bottom edges floating proportionally. The advantage of drawing with the guides is that the common and natural thing happens just from drawing the widget placement. The guides can also be used to set default proportionality values with the designer free to change the defaults when they desire. The guides in figure 11-7 can also be used to set the edge-anchors shown in 11-3.

The biggest disadvantage of the proportional system is that it does not take into account the space needs of the various child widgets. When labels or languages change, space needs change and the proportional system will not adapt to those changes.

Intrinsic size

All of the layout techniques we have discussed so far have only considered the size of the parent rectangle. In many cases we need to consider the size needs of the child widgets. Figure 11-8 shows a menu bar. Note that the widths of the items are based on the amount of text in the label. Menu items and lists of items frequently are sized by the number and space needs of the individual items.

Insert Design Transitions Animations Slide Show

11-8 – Menu bar layout

This particular layout algorithm is called the intrinsic size algorithm because it is based on the size needs of the individual children. The most common way to compose such layouts is using a stack, either vertical or horizontal. The menu bar in figure 11-8 is a horizontal stack.

For this to work each widget must implement the Layout interface shown in figure 11-9.

```
interface Layout
{   int desiredWidth();
    int desiredHeight();
    void setBounds(int left, top, right, bottom);
}
```

11-9 – Intrinsic size layout

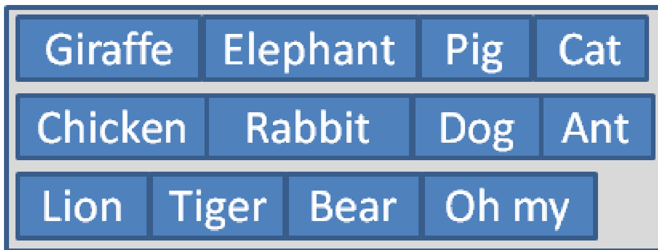
The stacks are just Groups that have additional code to perform layout function on their contents. So an HStack (horizontal stack) group would report its desiredWidth() to be the sum of all of the desiredWidths of its children. It would report its desiredHeight() to be the maximum of the desiredHeights of its children. A vertical stack would perform similarly with width and height reversed. When the setBounds method is called on an HStack it will arrange its children in order according to their desired width. The algorithm is in figure 11-10.

```
class HStack implements Layout
{   . . . . .
    void setBounds(int left,top,right,bottom);
    {   int wLeft=left;
        for(int i=0;i<contents.length;i++)
        {   Layout w = contents[i];
            int width = w.desiredWidth();
            w.setBounds(wLeft,top,wLeft+width,bottom);
            wLeft+=width;
        }
    }
}
```

11-10 – Horizontal layout

Each of the children widgets is given its desired width and they all share the top and bottom of the HStack itself. It is possible that the bounds did not contain enough space for all of the children. In such a case the window for HStack is used to clip off extra children. We will see mechanisms for dealing with this in later layout techniques.

Many times a simple vertical or horizontal stack is not enough. We generally want more than just a row or column of things. One approach is to create a vertical stack of horizontal stacks, as in figure 11-11.



11-11 – Vertical stack of horizontal stacks

As you can see, this is not very satisfactory. Nothing is lined up and it looks rather ragged.

As an alternative we can use a grid layout to better organize the contents. Each child is given a row and a column attribute that tells where in the grid it should go. For example "Giraffe" goes in row 1,

column 1 and “Dog” goes in row 2, column 3. The code for the Grid layout is shown in figure 11-12.

```
class Grid implements Layout
{   int colWidths[];
    int desiredWidth()
    {   for (int i=0;i<contents.length;i++)
        {   Layout w = contents[i];
            int c = w.getInt(“column”);
            int width = w.desiredWidth();
            if (colWidths[c]<width)
                colWidths[c]=width;
        }
        int dw=0;
        for (int col=0;col<colWidths.length;col++)
            dw+=colWidths[col];
        return dw;
    }
    int rowHeights[];
    int desiredHeight()
    {   similar to desiredWidth() }
    void setBounds(int left,top,right,bottom)
    {   int colRight[];
        int right=0;
        int colRight[0]=0;
        for (int i=1;i<colWidths.length;i++)
        {   right+=colWidths[i-1];
            colRight[i]=right;
        }
        colLeft[colWidths.length]=left;
        . . . similar for rows . . .
        for (int c=0;c<contents.length;c++)
        {   Layout w = contents[c];
            int row = w.getInt(“row”);
            int col = w.getInt(“column”);
            w.setBounds(colRight[col-1],rowBot[row-1],
                colRight[col],rowBot[row] );
        }
    }
}
```

11-12 – Grid layout algorithm

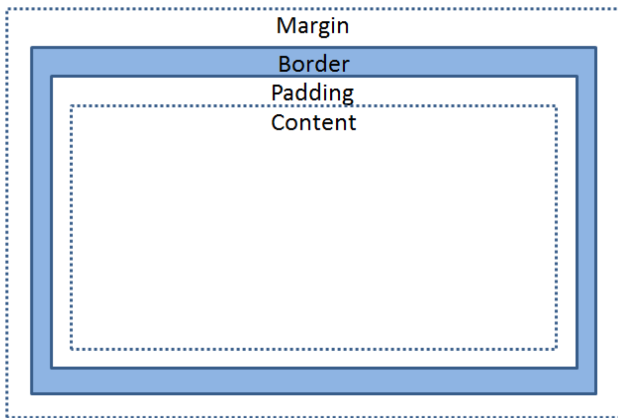
In the Grid technique each row’s height is the maximum of all elements in that row and each column’s width is the maximum of all elements in the column. Once the row and column widths are known we can easily place each item in its correct cell location. The result is shown in figure 11-13.

Giraffe	Elephant	Pig	Cat
Chicken	Rabbit	Dog	Ant
Lion	Tiger	Bear	Oh my

11-13 – Grid layout

Box model

The remaining issue is how large the leaf items (buttons, icons, text boxes, etc.) should be. One popular approach that is found in HTML/CSS is the box model. The box model defines a series of widths as shown in figure 11-14.



11-14 – Box model

The margin is the distance between the item and its adjacent item. This is intended to separate items from each other. The border is the width of the border if there is one. There can be a variety of borders drawn but only the border width matters to the layout algorithm. The padding is the distance between the content and the border. Each of these widths (margin, border, padding) can be set to control how the content will be positioned. In some systems like HTML/CSS the left, top, right and bottom values for each of these widths can be set separately.

The computation of the size of a widget that uses the box model would be the size of the content plus the sizes of the margin, border and padding. For example, in the case of a button with a label, we would use the font metrics information to compute the width and height of the label string. The width of the widget would be the width of the label string plus the widths of margin, border and padding on either side. A similar approach can be taken with icons, text boxes and a variety of other simple widgets.

Caching sizes

When using intrinsic size layouts, many times the layouts are composed of stacks within stacks within stacks. A simple implementation would call desiredWidth() recursively to get the width computed. The widths of the children are then set which would cause any child groups to compute their own layouts. These would recursively call their own children's desiredWidth() again. If there is a complex layout this duplicate calling can be a problem. Therefore many widgets will save their width information and not recursively call their children. The problem here is if one of the child's size needs were to change the saved sizes would be wrong. Many layout systems have an update() method that notifies a parent whenever a child's size needs change so that the parent or higher ancestor can recomputed the layout. On modern machines, with faster than gigahertz processors the duplication of size computation does not matter, but the layout system may still have caching built in and attention to update() may be required.

Variable intrinsic size

The intrinsic size technique allows the layout to adapt to the needs of the widgets it contains. If a language changes or any other information in a widget needs more space that layout algorithm can adapt. The proportional algorithm could adapt to the size of the parent window or to various aspect ratios of mobile devices, but could not adapt to the needs of widgets themselves. For example, the vertical scroll bar in figures 11-1 and 11-2 needs to grow and shrink with the size of the widget that contains it. The variable intrinsic size algorithm extends

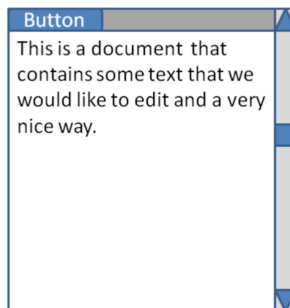
intrinsic size so that it can adapt to the space available as well as to the needs of the child widgets. This is the layout algorithm used by Java Swing. It is also used for mathematical formulas in TeX and LaTeX.

Figure 11-15 shows the Layout interface updated to accommodate the needs of variable intrinsic size. The big difference is that instead of a single desired(width/height) we elicit three values of min, desired and max. We will use these three values to create layouts that adapt to available space.

```
interface Layout
{   int minWidth();
    int desiredWidth();
    int maxWidth();
    int minHeight();
    int desiredHeight();
    int maxHeight();
    void setBounds(int left, top, right, bottom);
}
```

11-15 – Variable intrinsic size layout

The min value is the absolute least size that a widget can use to function properly. Any less and it just will not interact in any effective way. The desired size is what a widget would like to have to function comfortably. The max size is the most that the widget could possibly use to good effect.



11-16 – Variable sized widgets

Figure 11-16 shows some example widgets that will illustrate variable size needs. The button at the top does not vary much in its needs. It is

the size that it is. Its max and desired sizes would be the same. It is making no requests to grow if possible. Perhaps the min size could be smaller by reducing the padding and border widths. This would allow for a little variation, but not much.

The document area has different needs. For its width, it might report a minimum of 100 pixels. You can see the document, but it would not be comfortable to work. Its desired width would be the width of the page. That is what it really wants. It would probably report a maximum width that was the same as its desired. Its height would be different. The minimum height could be the height of 2 or 3 lines of text. That would not be comfortable, but you could get work done. Its desired height would be the height of a page or maybe ½ page. That would be comfortable work. The maximum height would the height of all the pages in the document. It could use all of that space to show the user more of what is being written.

The scroll-bar would report a width where min, desired and max are the same. It does not want to shrink or grow. For height it would report a moderate min and desired with a very large max so that it could grow to fill the space.

As with the intrinsic size algorithm, we generally compose groups of widgets using vertical and horizontal stacks as well as grids. The computation of min, desired and max for each of these is the same as the desired computation for intrinsic size. The difference lies in the actual setBounds() layout algorithm itself.

As with intrinsic size, the vertical and horizontal layouts are independent of each other, but similar. For the algorithm discussion we will only consider the HStack. The vertical and grid layouts are handled similarly. For the HStack, every widget is given the top and bottom values for the stack itself. For the horizontal layout we need to compute the width for each child widget. Once we have the widths we can lay them out the same as with the intrinsic size layout.

There are four cases to consider when performing variable intrinsic size layout. They are:

1. There is less than the minimum width available.
2. There is at least the minimum available but less than desired.
3. There is at least the desired width available but less than the maximum.
4. There is more than the maximum available.

Case 1 is easy. We give every child its minimum and let clipping deal with the overflow. There is no good answer and shrinking everything makes it uniformly bad.

In case 2 we want to give every widget its minimum and then give each one as much of its remaining desired width as possible. This is controlled by two formulas:

$$ratio = \frac{availableWidth - minWidth}{desiredWidth - minWidth}$$

$$wgtWidth = wgtMin + ratio * (wgtDesired - wgtMin)$$

The ratio computes how much of the difference between the desired width and the min width is actually available. The wgtWidth computes the width to be given to each widget based on its own difference between desired and minimum.

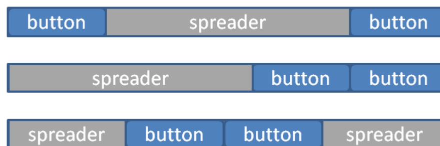
In case 3 we do the same as case 2 except that we give every widget its desired and then as much of its maximum as possible.

Case 4 can be handled in two ways. The first is to just give every widget its maximum and leave the extra space unused. This assumes that the widgets really cannot use the extra space beyond their max. The other alternative is to apply case 3 and let the ratio grow beyond 1.0. This proportionally gives more space to the widgets with large max. Either strategy will be fine because the widgets are always getting at least as much as they asked for.

Spacers and spreaders

There are two special widgets that can be used to augment the variable intrinsic size layout. A spacer is a transparent widget that has the same min, desired and max values. It is a rigid size and shape. Placing this in a vertical or horizontal stack will create a visual gap between other widgets. It can separate categories or just provide some openness to the visual layout.

A spreader is also a transparent widget except that it has a fixed min and desired with a very large max. It usually comes in vertical and horizontal variants. For example a horizontal spreader would have a fixed height of 1 some fixed min and desired width and then a large maximum width. It would not expand at all vertically but would grab as much space as it could horizontally. Figure 11-17 shows three examples of how spreaders can be used.



11-17 – Use of spreaders to manage layout

Vertical expansion variation

There are many situations such as word-wrapping text or the flow layout described below where it is desirable to first fix the width and then ask for the widget's height. For example, if a text field contains 300 words of text and is written in English, it would help to first tell it how many pixels wide it can be. Once it knows its width, it can flow the words and determine how much height it would like. This is widely used for HTML text layouts. The change to the algorithm is small. We change the Layout interface to that shown in figure 11-18.

```
interface Layout
{   int minWidth();
    int desiredWidth();
    int maxWidth();
    void setHorizontal(int left, right);
    int minHeight();
    int desiredHeight();
    int maxHeight();
    void setVertical(int top, bottom);
}
```

11-18 – Vertical expansion layout

In this variation the children are asked for their min, desired and max widths, the horizontal layout is computed and each child is given its horizontal layout using setHorizontal(). Now that each child knows its horizontal space it can respond to calls for min, desired and max height in a more appropriate way. Then setVertical() is called to complete the layout.

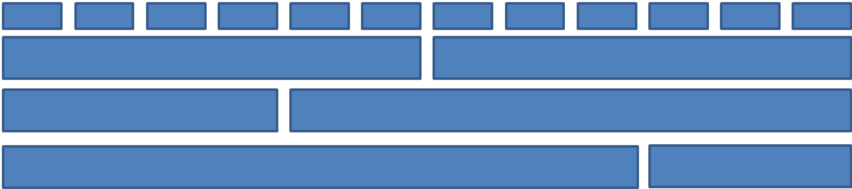
Grid/gutter

All of the layout systems described above are based on pixels. However, visual designers learned long ago that pixel-based widths and heights are awkward to deal with and require too much detail on the part of the designer. Visual designs such as poster or page layout are designed with a grid. This is a different grid than that layout describe above. It is a standardized set of places where things fit on a page. Magazine layouts have well defined grids that establish the look of the magazine. There are generally a few basic layouts into which writers and photographers pour their content and a consistent look for the magazine is easily achieved.

In paper page layout, grids are both vertical and horizontal. In web pages and other uses where scrolling is possible it is common to only establish a grid of columns with the content itself determining the height of the columns. The width of a widget is defined not in pixels but in the number of columns it will occupy.

A widely used grid system for web pages and other user interface layouts is the 960 grid. It was originally established to be 960 pixels

across. This accounts for the many web pages that will not adapt to smaller window sizes. Visual designers historically struggle with layouts that change size. They brought their page design techniques with them and created rigid layouts. We discuss a little later how to remedy this.



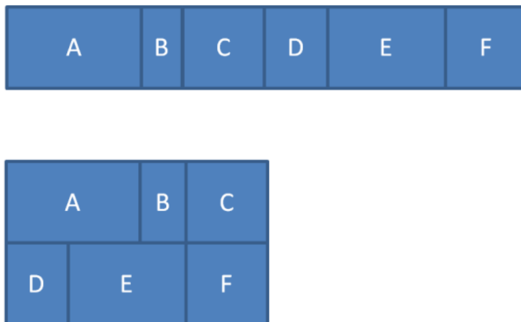
11-19 – Variations on 12 columns

The reason for 960 pixels is that it easily divides into 12 equal columns of 80 pixels each. The reason for 12 columns is that it divides into many different layouts, as shown in figure 11-19. It can be divided into two halves of 6 columns each or into thirds of 4 columns or fourths of 3 columns each. More interesting layouts of 9 and 3 columns or 4 and 8 columns are easily laid out. The key advantage is that one quickly picks a number of columns and everything else works out nicely.

In this system the width of a widget is specified in columns, which is multiplied by 80 to get the total width of the widget. The column/grid layout is used widely with the box model described earlier. By setting the margin of the widgets to a uniform value, the gutter space between columns is uniformly achieved. This rigid column model can be improved by replacing 80 pixels with 80/960 or 8.333%. If the fixed column widths are replaced by corresponding percentages, then the widths of the columns will flex with changes in window size. If, as discussed in the vertical expansion variation on variable intrinsic size, the width is set before asking the widget how tall it wants to be, the layout adapts vertically as well. This is the approach used by the Bootstrap styling system. Designing layouts in columns and margins is much easier than in pixels.

Flow

The flow layout is a larger scale version of what happens when a text box uses word wrapping. If there are too many words to fit across a line then words are moved to the next line so that the sentence still flows. A similar thing can be done with widgets in rows of layout. At the top of figure 11-20 is a set of widgets laid out in a row. If the parent width shrinks for some reason, then widgets D,E and F can flow into a new row.

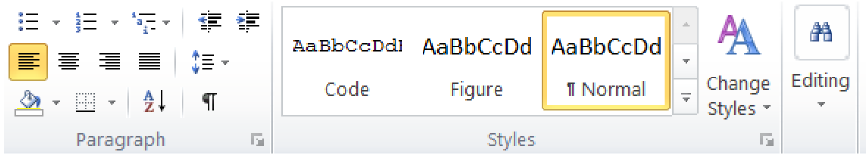


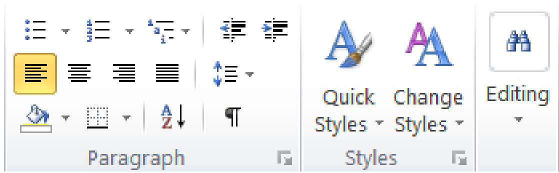
11-20 – Flow layout

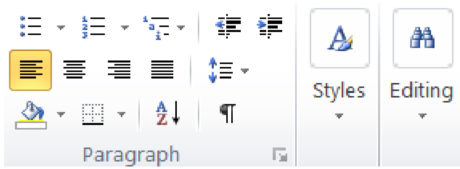
If the widths of the widgets were defined in columns rather than pixels, as described above, then they will flow to form generally pleasing alignments. For this to work, simple percentages for column widths will not work because instead of flowing to a new row the columns will just get thinner and thinner as the available width is reduced. A solution to this is to ask for the minimum widths of all of the child widgets. The maximum of this can be used to set the minimum column width. This guarantees that every widget will get at least its minimum width. By not allowing columns to go below a minimum width, widgets will flow to a new row rather than get smaller. For larger widths the columns still flex with any additional width.

Conditional

In some cases simply manipulating the sizes and flow of widgets will not adequately account for changes in the available space. Figure 11-21 shows the Microsoft Word button ribbon at three different widths.







11-21 – Adapting layouts

Note that the “Styles” group of widgets does not just change width it completely changes its presentation. At the first step it collapses the list of styles into a drop-down button called “Quick Styles” and as the space becomes even more limited the entire “Styles” group becomes a single drop-down button. Note also that the “Paragraph” group is not changing. The “Styles” group has reported a minimum width that is consistent with the single drop-down button. This allows other widgets or groups of widgets to use the space.

One way we could represent this is with a CondLayout object as shown in figure 11-22. The CondLayout reports its minimum width as 100 because that is the smallest possibility in its list of layouts. It reports its desired width as 200 as specified by the attribute and it computes its maximum from the maximum layout of the largest layout given.

When CondLayout receives its width setting it will pick the largest of its layouts that will fit within the available space. Each layout is completely different. This allows for radical changes in the offered presentation as the available space changes.

```
CondLayout{ layouts: [
      { minWidth:300,
         [ widgets for the large layout ] },
      { minWidth:200,
         [ widgets for the medium size layout ] },
      { minWidth: 100,
         [ widgets for the smallest layout ] }
   ], desiredWidth:200
}
```

11-22 – Conditional Layout

This kind of conditional layout is supported in HTML5/CSS using the @media query. Widths are specified in HTML5 as attributes that can be set using CSS (Cascading Style Sheets). Style specifications are processed from beginning to end with latter definitions overriding earlier definitions. The @media query has a set of possible rules that can test a variety of situations, such as page width, presentation on a projector and several other things. If the test associated with an @media query is true then all of the CSS definitions inside of that block are executed. If it is false then they are skipped. This is widely used in a fashion similar to figure 11-22 to change the styling of a page depending upon the available space.

Summary

In this chapter we have discussed a variety of layout mechanisms. We started with the simple fixed layout that is easy to define, but does not flex with the available space. The edge-anchored layout is an improvement on the fixed layout in that it varies the edges to which widgets are fixed so that they can be fixed to the right or bottom instead of just left and top. The proportional layout simply gives each widget a fraction of the available space.

None of the above account for the actual need of the various widgets to display their contents. The intrinsic size layout asks each child how big they want to be and then gives them that space. This allows for dynamic adjustment when a label or icon is given a different size. The variable intrinsic sized layout allows widgets to specify their flexibility by reporting their minimum, desired and maximum sizes. This provides a more adaptive layout.

Grid systems, taken from visual design, assert a more global layout of a display so that everything aligns well and has good whitespace structure. This is made more flexibly by specifying column widths as percentages rather than fixed widths. The flow layout allows widgets to flow to additional rows if the space becomes too small and conditional layouts allow for completely different widget arrangements to be used as the available size changes.

Exercises