

# Resolution Independent Layout

Stratovan Corporation  
David F. Wiley  
October 10, 2007

## Summary

We present a method for interpolating layout design to arbitrary dimensions based on two or more layout definitions. At least two layout definitions are required as input to our method and can be generated by either a layout designer or determined by some other means. The layout definitions are interpolated to fit the runtime layout size. We discuss the issue in terms of user interfaces for computer software.

## Definitions

- 1) **User Interface (UI)**: the human-computer interface allowing a computer user to interact with computer software.
- 2) **Widget**: a visual object representing some sort of data item or action the user can interact with on the computer monitor through the use of a keyboard, mouse, or some other means. Well known user interface widgets are: button, listbox, combobox, tree, menu, scrollbar, etc.
- 3) **User Interface Element**: used interchangeably with widget
- 4) **User Interface Layout**: the organization of one or more widgets together forming a user interface for interacting with and guiding program flow.
- 5) **Layout Definition**: the exact coordinates defining the location of or bounding region(s) of widget(s) in a user interface layout.
- 6) **Resize event**: any event in computer software causing the layout to be determined, either for initial creation of the UI or for dynamic resizing initiated by the user at runtime or some other means.
- 7) **Design time**: the planning process, act of designing, or laying out of a user interface layout manifesting as either a visual tool or hand coding of the layout definition.
- 8) **Runtime**: any point in time where the design-time specified layout definition may be manipulated, extrapolated from, or examined in order to determine a layout for a size other than and including the design specified size, which can include determination for preview or other means during design time.
- 9) **WYSIWYG**: “What You See Is What You Get” refers to a visual design tool for graphically manipulating objects that during runtime manifest as visual objects on the user’s computer monitor. This is considered the opposite of hand coding, but results in substantially the same input to the computer software for determining the layout definition.
- 10) **Hand code**: to write, type, or otherwise produce instructions that are used in some way by computer software either through being compiled into object/program code or by being read/accessed during design time, runtime, or by other means.
- 11) **Resolution Dependence**: refers to the problem UI visual elements looking “different” at various screen resolutions or layout sizes due to higher or lower pixel granularity than that used at design time.
- 12) **Application Developer**.

## Background

We describe a method for relieving the resolution dependency of user interface (UI) visual elements prevalent in a variety of application areas requiring the layout thereof. For clarity, we focus our discussion in the space of UI layout design for computer software.

Two main stages regarding computer software UI layout are that of layout design (development or design time stage) and runtime layout (execution of the program). Layout design pertains to a UI designer specifying the locations of individual UI interface elements or widgets (i.e., buttons, list boxes, edit boxes, custom controls, etc.) as input to computer software. For the purposes of this patent, we are only concerned with widget position while the widget type is irrelevant.

A variety of mechanisms are available for specifying widget locations that are later used during runtime to perform the actual UI layout. These range from WYSIWYG design tools allowing drag-n-drop placement of widgets to hand-coded program code placing widgets at specific locations. Runtime UI layouts are either (i) fixed, (ii) dynamic, (iii) user adjustable, or (iv) some combination of those. A fixed UI uses the same layout definition for all cases or limits the runtime layout dimensions to a fixed size and is trivial for an application developer to implement for runtime by using the source design throughout the software session. Any sort of dynamic UI poses the difficult problem of resizing (or repositioning) widgets to their appropriate positions for arbitrary and otherwise unknown layout dimensions. Conventionally, for complex UI layouts, the only reasonable solution to this problem has been to hand code the dynamic layout logic. This fact hinders any UI modifications due to the complexity associated with integrating design changes into the underlying dynamic-repositioning algorithm.

Due to this complexity, most software applications have minimal support for dynamic user interface layouts and i) have fixed size layouts (resulting in dialog boxes or application windows that cannot be resized); ii) are design for a selected set of layout sizes (i.e., 640 x 480, 800 x 600, 1280 x 1024, etc.) and subsequently select the layout definition matching the runtime dimensions; or iii) have resizing logic moving widgets based on rules derived from a single layout definition.

Layout logic describing how the boundaries for a widget change during a resize event is usually based on some combination of the following rules:

- 1) Leave the widget boundary in its current state.
- 2) Use a fixed boundary width, height, or position for all layout sizes.
- 3) Variable boundary width or height filling a region of space.
- 4) Adjust the boundary relative to another boundary (possibly that of another widget), for example, having a constant offset from an edge.
- 5) Scale the boundary based on a percentage of the new dimensions derived from its current location and/or the single input layout definition.

Conventional UI layout design specification systems/methods are based on a single UI layout definition and are either i) implemented in a WYSIWYG tool; ii) hand-coded by a programmer; or iii) some combination thereof. Some frameworks dealing with this problem allow the specification of relationships between widgets, for example, tying the edge of one widget to that of another such that, if one widget is moved, those dependent on it move accordingly. Regardless

of these tools, complex UI layouts involving multiple configurations require some amount of hand coding by a programmer to achieve optimal runtime results.

### ***Resolution Independence***

An ancillary problem to dynamic UI layout is that of widget-content resolution matching for a given runtime environment (usually screen pixel resolution). This primarily involves scaling widget content to appropriately match the layout size. This is most apparent in the font(s) used to render widget text or any image displayed as widget content. Currently, UI designers construct a layout for each layout size and resolution that is to be expected in the runtime environment for their software. Since creating a single UI layout definition can be an extremely time consuming process under the conventional system, not all sizes and resolutions are specified resulting in software that does not always resize or scale well in all runtime environments.

## **Description**

Our method uses two or more layout definitions to determine runtime layout. This is not to be confused with selecting a single runtime layout from a predefined input set for a given runtime size. It differs in that at least two layout definitions are interpolated to produce the runtime layout. Thus, the runtime layout is derived directly from at least two layout definitions rather than a single definition as in conventional methods, see Fig. 1.

Our method interpolates two or more layout design specifications at runtime in order to determine the desired widget locations for arbitrary layout dimensions. Additionally, this technique provides resolution independent layout control (i.e., widget content or font scaling) by providing a means to scale widget content at runtime by a user-specified amount. This is described later on.

The layout designer must, at a minimum, specify two layout definitions as input to our method. Our runtime-layout method interpolates the provided layout definitions at runtime fitting the widgets to arbitrary window dimensions. Fig. 2 shows an example having two design definitions that are linearly interpolated to fit a runtime dimension. The general idea is to specify “small” and “large” versions of the UI. For each layout size, the designer can optimally place widgets. Then, at runtime, the two designs are interpolated to match the runtime size, producing a smooth blend between the two layouts as the user changes the runtime size between the two input definitions.

This method can be used for runtime dimensions larger or smaller than the design definitions. The following formulas can be used to perform the interpolation between two designs

$$t_w = \frac{(w - w_1)}{(w_2 - w_1)} \quad \text{and} \quad t_h = \frac{(h - h_1)}{(h_2 - h_1)}$$
$$x_w = x_1(1 - t_w) + x_2 t_w \quad y_h = y_1(1 - t_h) + y_2 t_h$$

where  $w$  and  $h$  are the target dimensions for the new layout,  $\{w_1, h_1\}$ , and  $\{w_2, h_2\}$  are the width and height of the two input layout definitions,  $(x_1, y_1)$  and  $(x_2, y_2)$  are the coordinates for any homologous point in both layout definitions (not necessarily enclosed within the boundaries of the layout definition), and  $(x_w, y_h)$  is the desired coordinate in the new layout.

An example flowchart describing the conventional UI layout process is shown in Fig. 3. It is important to note that it is possible under the conventional paradigm to produce multiple layout definitions that are selected from at runtime to choose the best single layout definition for the given runtime dimensions. This is however not the same as interpolating between multiple definitions to arrive at the runtime layout.

The layout processes change to that shown in Fig. 4 when using multiple layout definitions to determine the runtime layout. In this case, all layouts are treated as dynamic layouts. Layout definitions can be created either by a WYSIWYG tool or by some other means, for example, hand coding or capturing an existing UI layout in multiple states. Regardless of how they are obtained, multiple definitions are interpolated at runtime producing the runtime layout.

The difference between this method and the conventional method is that multiple layout definitions play a direct role in computing runtime widget location rather than only deriving this location from a single layout definition.

Considering that the interpolation method can be implemented by the widget toolkit provider, the brunt of the user interface design specification (and repositioning) is moved out of the hands of the application developer and into that of the user interface designer.

### ***Resolution Independence***

Our method for resizing UI layouts using two input layout definitions suffers from resolution dependence just as other methods. Figs. 5, 6, 7, and 8 demonstrate the problem. In this example, Figs. 5, 6, and 7 use an input layout definition that is simply a rule dividing the space into two widget regions at one third of the runtime width.

Fig. 5 shows the four runtime resolutions in this example (320 x 240, 640 x 480, 1024 x 768, and 1920 x 1200 pixels) in which the runtime layout is divided into two regions divided at one third of the width and spanning the full height at all layout sizes.

Fig. 6 shows the results of using a constant 12pt font size (constant scale factor) for all layout sizes (which happens to work well for the 640 x 480 layout). This is obviously not desired since the text does not fit properly into the delineated regions for all resolutions.

If the font is scaled by a factor of 0.5, 1, 1.6, and 3 for each resolution (obtained from the relative resolution widths), respectively, the font size will match the change in layout width relative to the 640x480 layout (since 640x480 looks right), as shown in Fig. 7. However, users do not always want the font to scale in this manner, for example, on large 24in monitors having 1920 x 1200 pixel resolution, the text shown in the widget would occupy about 8in of horizontal screen space thus squandering the increased pixel resolution.

A more desirable layout-resizing scheme produces that shown in Fig. 8, which holds the font size constant throughout the layout size change while resulting in a layout in which the text fits.

Thus, the resolution dependence problem lies in the production of the result shown in Fig. 8 due to the widget regions being variable (dynamic) and the content scale remaining constant between runtime resolutions. This can be handled using our previously described layout method using multiple design definitions.

Additionally, the user may decide they want some combination of the layouts shown in Figs. 7 and 8 (i.e., some amount of content scaling and some amount of dynamic layout). This can be achieved by creating three layout definitions that exhibit the desired results, as shown in Fig. 9. In this situation, the layout size varies along the horizontal x-axis and the widget content size varies along the vertical y-axis. This is sufficient to define a resolution independent parameter space as shown in Fig. 10.

In this case, the three layout definitions form a triangular parameter space that can be represented by Barycentric coordinates. During runtime, the coordinate along the x-axis is determined for the desired layout size in addition to that of the y-axis for the content size. This identifies the corresponding interpolation weights applied to each layout definition to determine the runtime layout size.

### ***Runtime User Adjustable***

It is often desirable to allow the user to move widgets in some manner during runtime. Runtime manipulation of the input layout definitions can accommodate this. An example flowchart of this process is shown in Fig. 11.

Either the original or a copy of the input definitions can be modified during runtime. The adjustment process uses “inverse” interpolation to determine the locations of the widget in the input layout definitions needed in order to move the widget to the desired location at runtime as specified by the user.

### ***Higher-order interpolation***

In the examples presented thus far, we have used linear interpolation to interpolate between two (or three) layout definitions. It is possible to use higher-order interpolation methods such as cubic splines in the case of three or more layout definitions.

## **Application Spaces**

Our layout method thus far has been described in the space of user interfaces for computer software; however, our layout method can be used to solve a number of issues in a variety of related application spaces, for example, HTML layout, desktop publishing and document layout, and cell phone interfaces.

### ***HTML Layout***

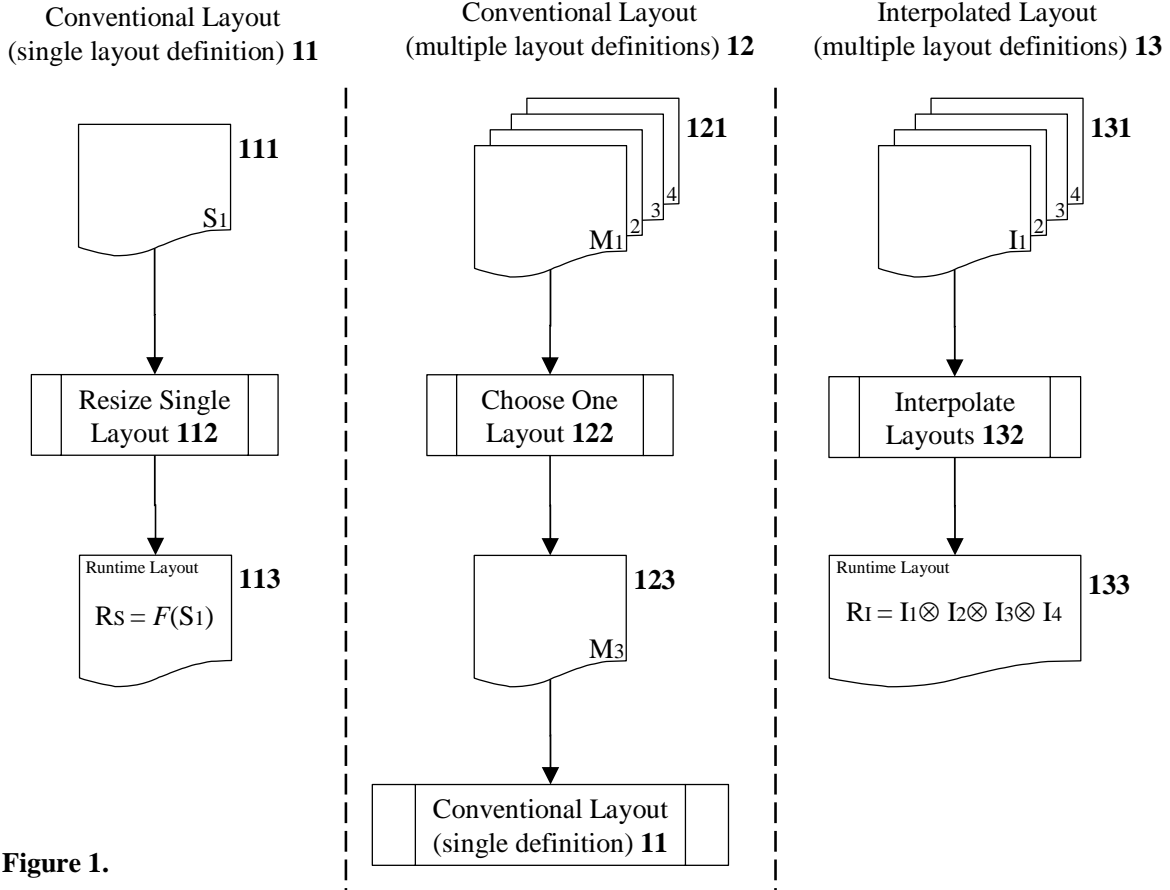
Just as in the UI layout and resolution dependence for computer software applications, the conventional HTML layout mantra is to provide one universal layout accommodating the majority needs of visitors to a particular website. Some websites provide alternative layout configurations depending upon the user’s browser window size, but none interpolate between two or more designs. Difficulties in this case are primarily rooted in how HTML layout occurs.

In principle, HTML layout can be described as a bottom-up layout mechanism where nested content size is first computed and then propagated upwards to parenting regions in turn defining their size and further propagating upwards in the hierarchy as shown in Fig. 12.

There is already a mechanism within HTML browsers to adjust the content scale (usually described as font size). HTML fonts depending on this setting vary in size and thus change the size of the overlying region they are contained within. This ends up exacerbating the resolution dependence problem discussed earlier.

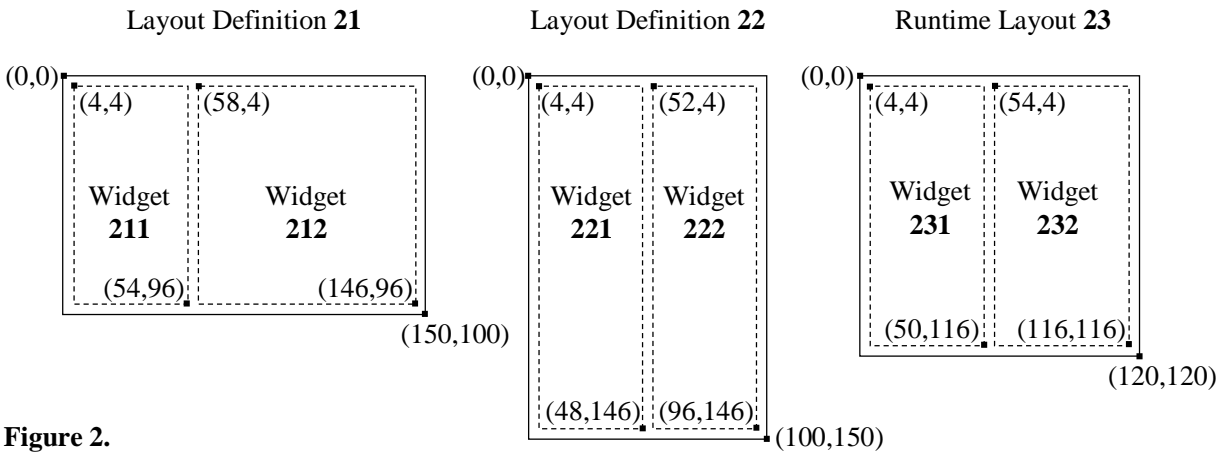
Additionally, in the case of cell phone (or PDA) HTML browsers, the problem is further exacerbated due to the small size of the screen (and corresponding pixel resolution).

This problem can be relieved by using the tri-layout solution discussed above for resolution independence. This process is shown in Fig. 13.



**Figure 1.**

Fig. 1 shows the process by which layout design definitions are converted into runtime layout specifications. The conventional layout (single definition) 11 uses one source layout 111 that is resized during runtime 112 based on some function  $F$  (or rules) dictating how the widgets on the layout are modified given a certain runtime environment. The result is a layout 113 used to determine the runtime widget positions, which is a function of only one source definition. Multiple layout definitions can also be used in the conventional method by first selecting one of the input definitions. Thus, the conventional layout (multiple definitions) 12 has several input layouts that are selected from at runtime 122. The result is one of the source definitions 123 that is then fed into the conventional layout process 11. Again, the resulting runtime layout is a function of only one source layout definition even though multiple definitions began the process. Our method differs in that the runtime layout is a combination, interpolation, and/or convolution of more than one layout definition. Our interpolated layout method 13 processes multiple input layouts 131 (usually through interpolation 132) and results in a runtime layout definition 133 that is a function of more than one input definition.



**Figure 2.**

Fig. 2 shows a runtime layout 23 determined from two source layouts 21 and 22. Layout definition 21 is designed for a width and height of 150 and 100 pixels, respectively, and has upper-left and lower-right corner coordinates of (0, 0) and (150, 100), respectively. Two widget regions 211 and 212 are defined for layout definition 21. Widget 211 has upper-left and lower-right corner coordinates of (4,4) and (54,96), respectively. Widget 212 has upper-left and lower-right corner coordinates of (58,4) and (146,96), respectively. Layout definition 22 is designed for a width and height of 100 and 150 pixels, respectively, and has upper-left and lower-right corner coordinates of (0, 0) and (100, 150), respectively. Two widget regions 221 and 222 are defined for layout definition 22. Widget 221 has upper-left and lower-right corner coordinates of (4,4) and (48,146), respectively. Widget 222 has upper-left and lower-right corner coordinates of (52,4) and (96,146), respectively. Runtime layout 23 is interpolated from layout definitions 21 and 22. The runtime width and height are 120 by 120 pixels, respectively, and is specified by the upper-left and lower-right coordinates (0, 0) and (120, 120). Runtime interpolation for the size of 120 by 120 pixels, between layout definitions 21 and 22, results in two runtime widget locations 231 and 232. Widget 231 has upper-left and lower-right corner coordinates of (4,4) and (50,116), respectively. Widget 232 has upper-left and lower-right corner coordinates of (54,4) and (116,116), respectively.

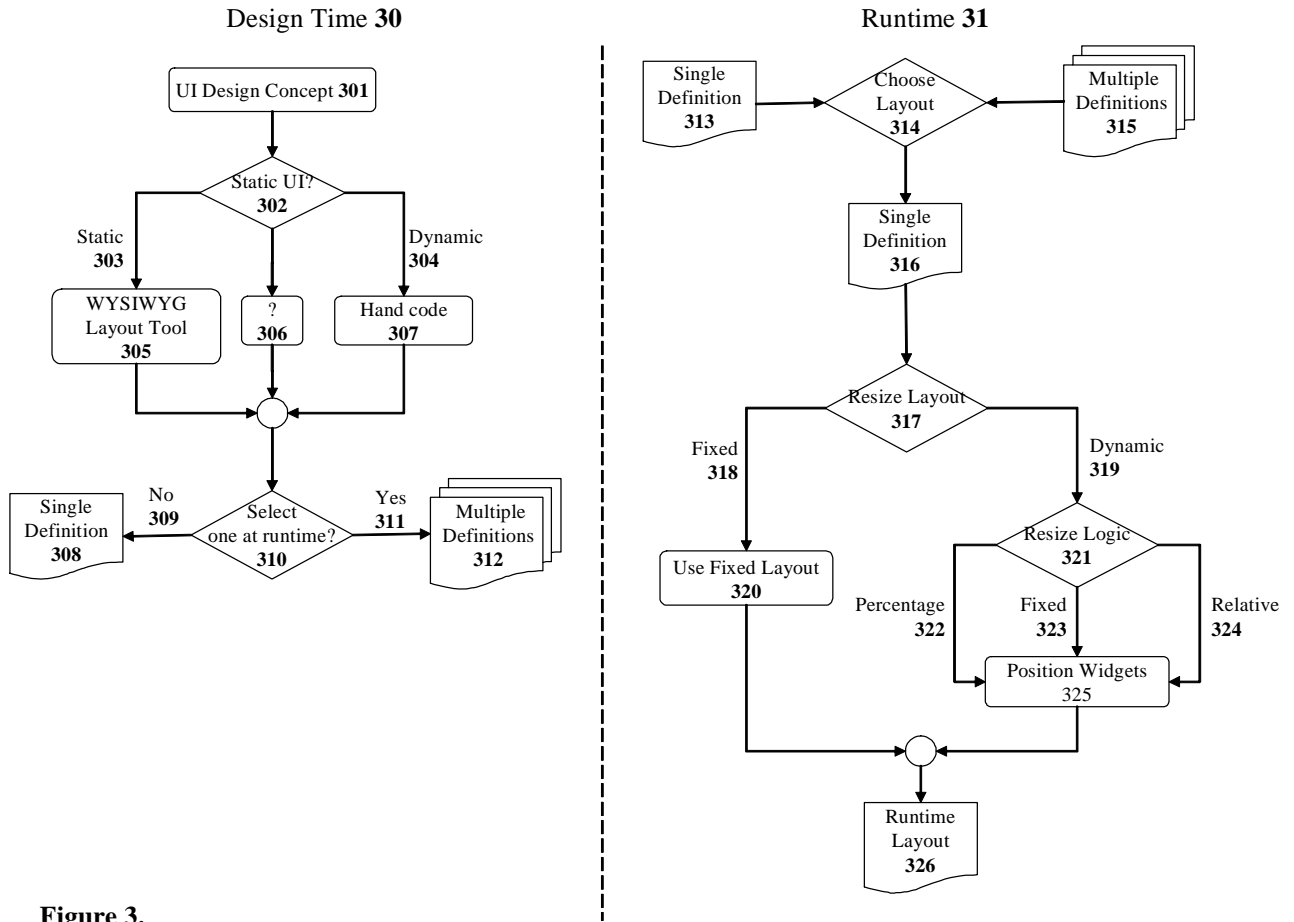
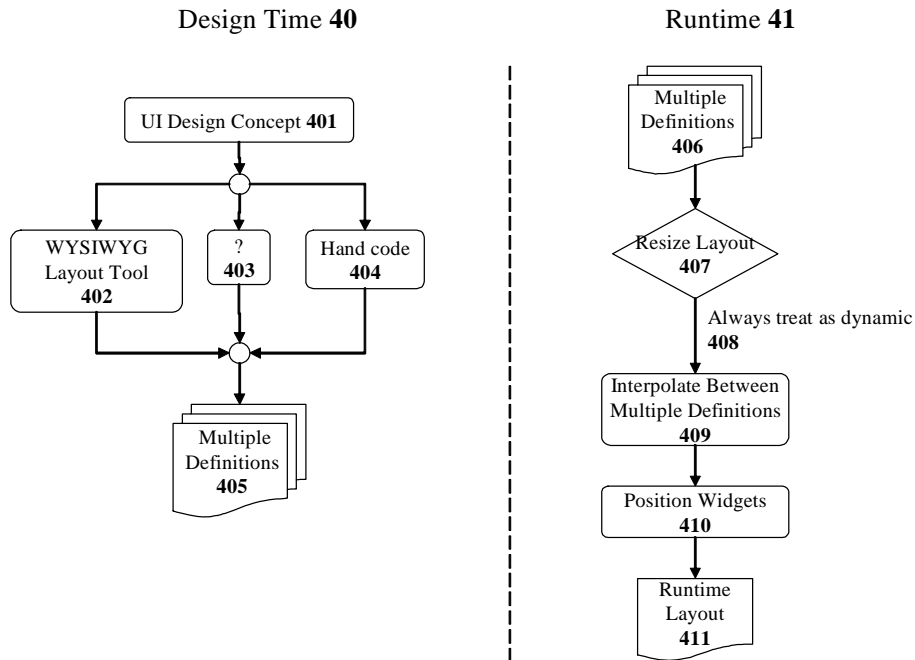


Figure 3.

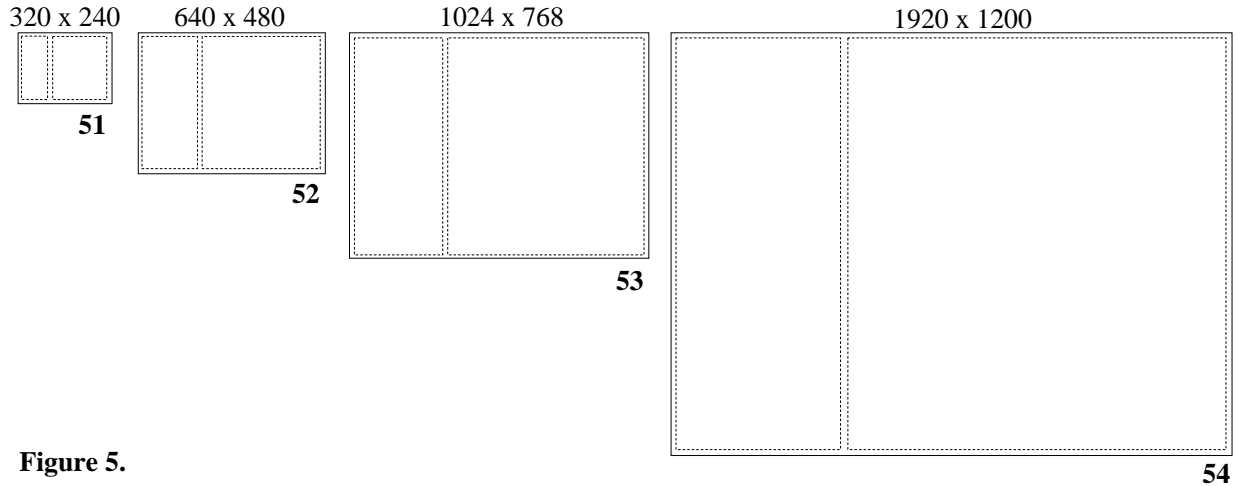
Fig. 3 shows conventional design 30 and runtime 31 processes for determining runtime layout 326 of user interfaces for computer software. The design time flowchart 30 describes the process for specifying layout definition input to computer software. The runtime flowchart 31 describes how the design time layout definition is used to determine runtime layout. For design time, a layout design concept 301 is converted into software input specification by first determining whether the layout will be static or dynamic, decision 302. If static 303, then a WYSIWYG layout tool 305 can be used to specify widget locations and region boundaries for the static layout. If the layout is dynamic 304, the layout itself may need to be fully (or partially) specified by hand coding 307 the layout definition in program source code. Depending upon the complexity of the user interface, some combination of WYSIWYG layout tool 305 and hand coding 307 can be combined in 306 to somewhat simplify the hand coding work. The result of all specification methods is either a single layout definition 308 or multiple definitions 312 that can be used as input to computer software for specifying widget locations at runtime. During runtime, either the single definition 313 is used or one of the multiple definitions 315 is selected (that best matches the runtime dimension) to position widgets, as indicated by single definition 316. During a resize event 317, the type of layout dictates how the widgets are positioned. If the layout is fixed 318, then the input layout definition is used to directly specify widget locations 320 and ultimately the runtime layout 326. If the layout is dynamic 319, then the logic driving the layout resizing determines how best to treat each widget based on how that widget's

boundaries are supposed to change, for example, as a percentage 322 of the runtime size, as a fixed size 323, or relative 324 to some other boundary or edge. Each widget is then positioned 325 at the newly computed location; resulting in runtime layout 326.



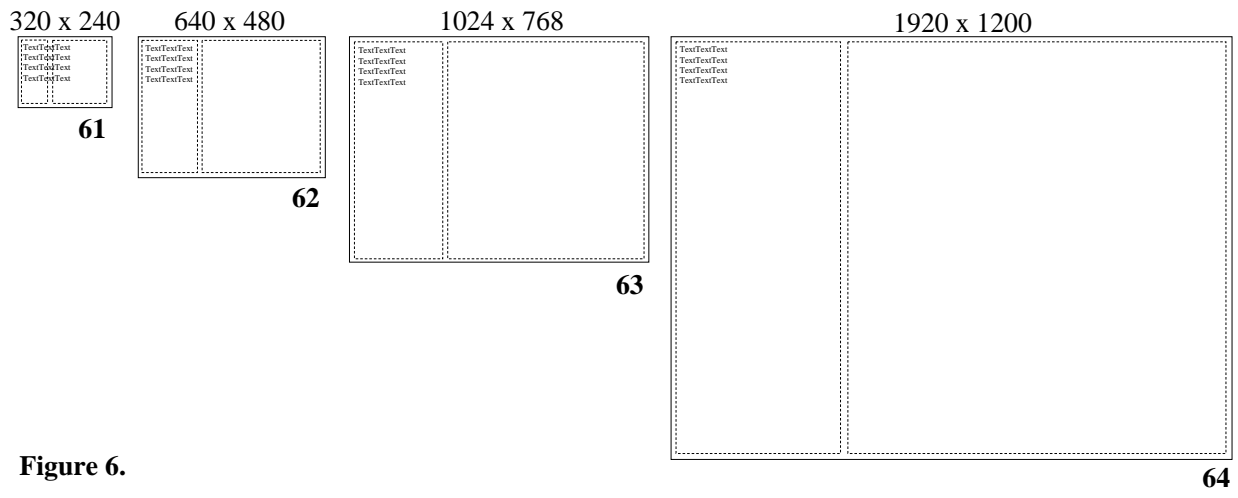
**Figure 4.**

Fig. 4 shows the new design 40 and runtime 41 processes for determining runtime layout 411 of user interfaces by interpolating from multiple layout definitions. The design time flowchart describes the process for specifying layout definitions input to computer software. The runtime flowchart describes how the design time layout definitions are used to determine runtime layout. For design time, a layout design concept 401 is converted into software input specification by either a WYSIWYG layout tool 402, hand coding 404, or some other process 403 that results in the definitions, for example, some combination between WYSIWYG layout tool 402 and hand coding 404 or some other process such as capturing existing layouts. The result of all specification methods is multiple layout definitions 405 that can be used as input to our method for specifying widget locations at runtime. During runtime, the multiple definitions 406 (derived directly from 405 if not the same) are used to position widgets. During a resize event 407 the layout is always treated as dynamic 408 in that the layout is always interpolated from multiple definitions (even if the runtime size coincides with one of the input definitions, this is a special case where the interpolation weight is 100% for that layout and 0% for the others). The logic driving the layout resizing determines each widget position by interpolating widget positions 409 from input layout definitions. Each widget is then positioned 410 at the newly computed location; resulting in runtime layout 411.



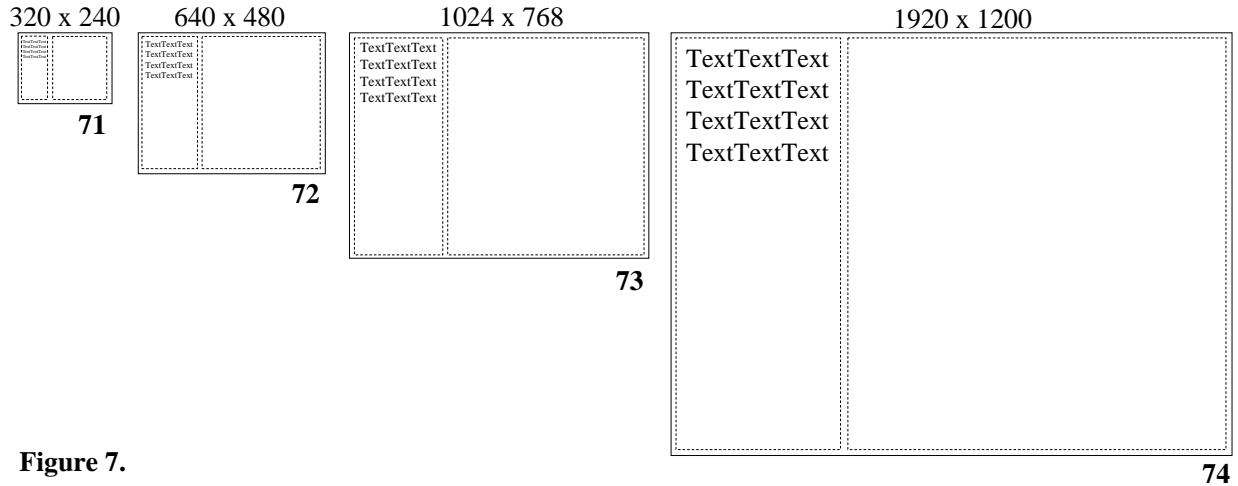
**Figure 5.**

Fig. 5 shows four layouts 51, 52, 53, and 54 having widths and heights of 320 x 240, 640 x 480, 1024 x 768, and 1920 x 1200 pixels, respectively. The same two widget regions are delineated by dotted lines on each layout. The vertical division (for the widgets) is located at one-third of the layout width while widget heights span the full height of the layout.



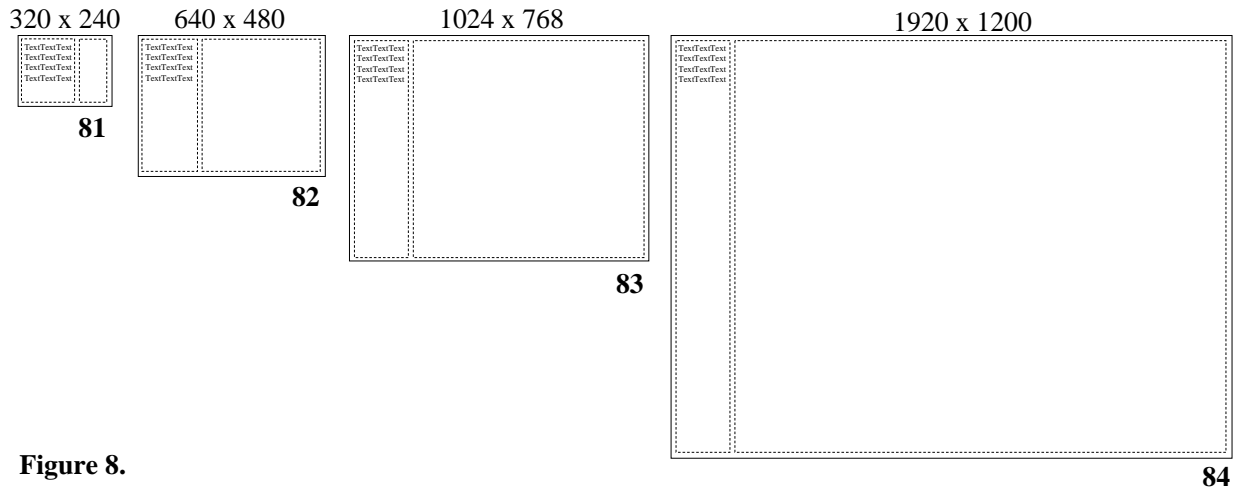
**Figure 6.**

Fig. 6 shows four layouts 61, 62, 63, and 64 having widths and heights of 320 x 240, 640 x 480, 1024 x 768, and 1920 x 1200 pixels, respectively. The same two widget regions are delineated by dotted lines on each layout. The vertical division (for the widgets) is located at one-third of the layout width while widget heights span the full height of the layout. A constant 12pt font size (constant scale factor of 1) is used for each layout.



**Figure 7.**

Fig. 7 shows four layouts 71, 72, 73, and 74 having widths and heights of 320 x 240, 640 x 480, 1024 x 768, and 1920 x 1200 pixels, respectively. The same two widget regions are delineated by dotted lines on each layout. The vertical division (for the widgets) is located at one-third of the layout width while widget heights span the full height of the layout. A variable font size is used for each layout. Layouts 71, 72, 73, and 74 use a factor of 0.5, 1, 1.6, and 3, respectively, for adjusting the font size.



**Figure 8.**

Fig. 8 shows four layouts 81, 82, 83, and 84 having widths and heights of 320 x 240, 640 x 480, 1024 x 768, and 1920 x 1200 pixels, respectively. The same two widget regions are delineated by dotted lines on each layout. The vertical division (for the widgets) is variable depending upon the layout size while widget heights span the full height of the layout. A constant 12pt font size (constant scale factor of 1) is used for each layout.

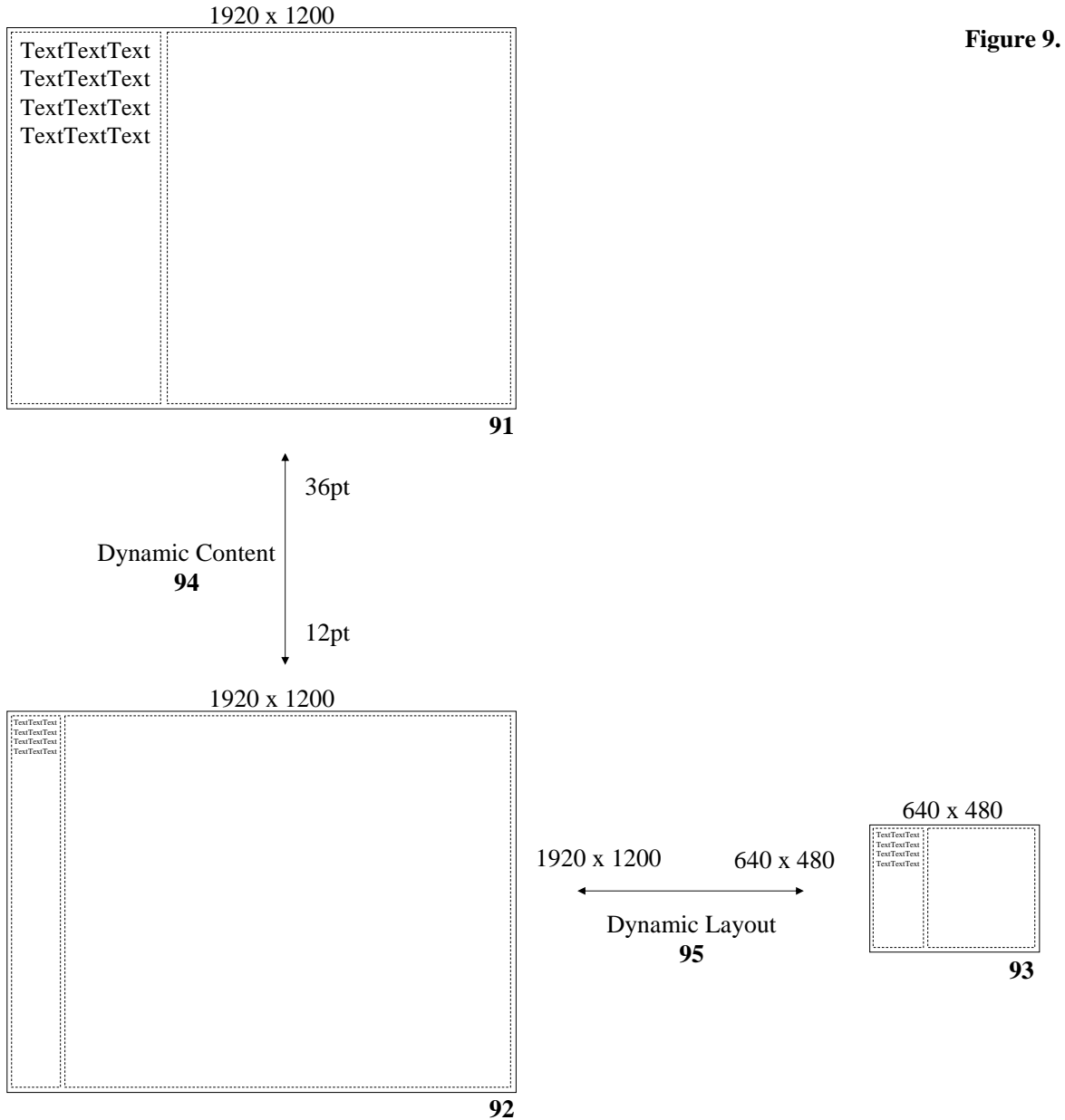
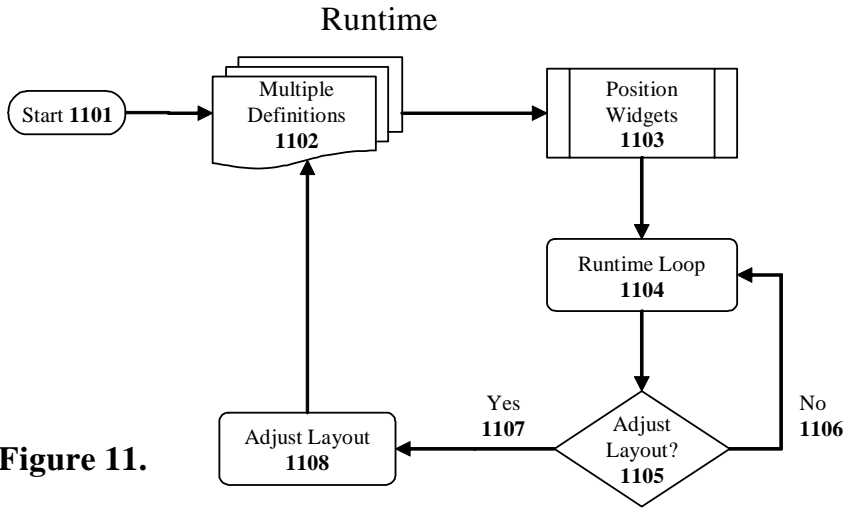


Fig. 9 shows how to achieve resolution independence by using three layout definitions 91, 92, and 93. Widget regions are delineated by dotted lines. Dynamic content 94 is achieved by providing two design layouts (having the same layout dimensions of 1920 x 1200 pixels) using two different content scales defined by a 36pt and 12pt font for layouts 91 and 92, respectively. Widget locations for the two designs are adjusted to accommodate the change in content scale as indicated by the vertical division between the two widgets. To achieve dynamic content 95, a third design is provided that holds constant content scale while changing layout size to 640 x 480 from 1920 x 1200 pixels for layouts 93 and 92, respectively.

**Figure 10.**



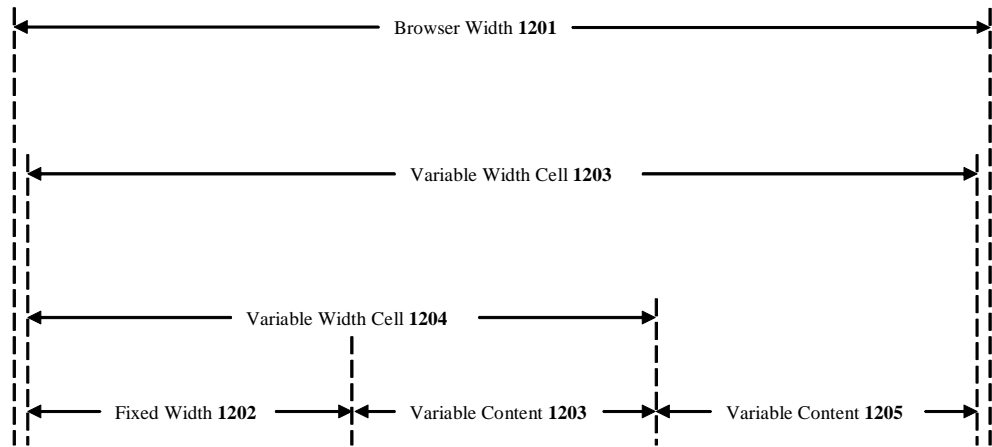
Fig. 10 shows a resolution independence parameter space using three layout definitions 101, 102, and 103. Dynamic content 104 is achieved by providing two design layouts (having the same layout dimensions of 1920 x 1200 pixels) using two different content scales defined by a 36pt and 12pt font for layouts 101 and 102, respectively. To achieve dynamic content 105, a third design is provided that holds constant content scale while changing layout size to 640 x 480 from 1920 x 1200 pixels for layouts 103 and 102, respectively.



**Figure 11.**

Fig. 11 shows an example process for runtime modification of layout definitions used to compute the runtime layout. The input layout definitions 1102 are first used to create an initial layout 1103. When a user-specified resize event occurs 1107, the user adjustment is inverted and applied 1108 to the layout definition 1102, which is then used to reposition the widgets 1103.

Conventional HTML Layout



**Figure 12.**

Fig. 12 shows bottom-up propagation of content width for an HTML layout (similar can apply to heights). The basic premise is that a depth-first computation of content size is employed in conventional HTML layout such that content regions 1202 and 1203 are computed first. This size is then propagated up to region 1204 allowing both 1204 and 1205 to determine their respective sizes. This information is then propagated upward to 1203 to complete the layout for the given browser width 1201. It is possible for table rows (or columns) to designate a fixed width (or height) in which case these values are propagated downward. Though, in general, the size propagation is upwards.

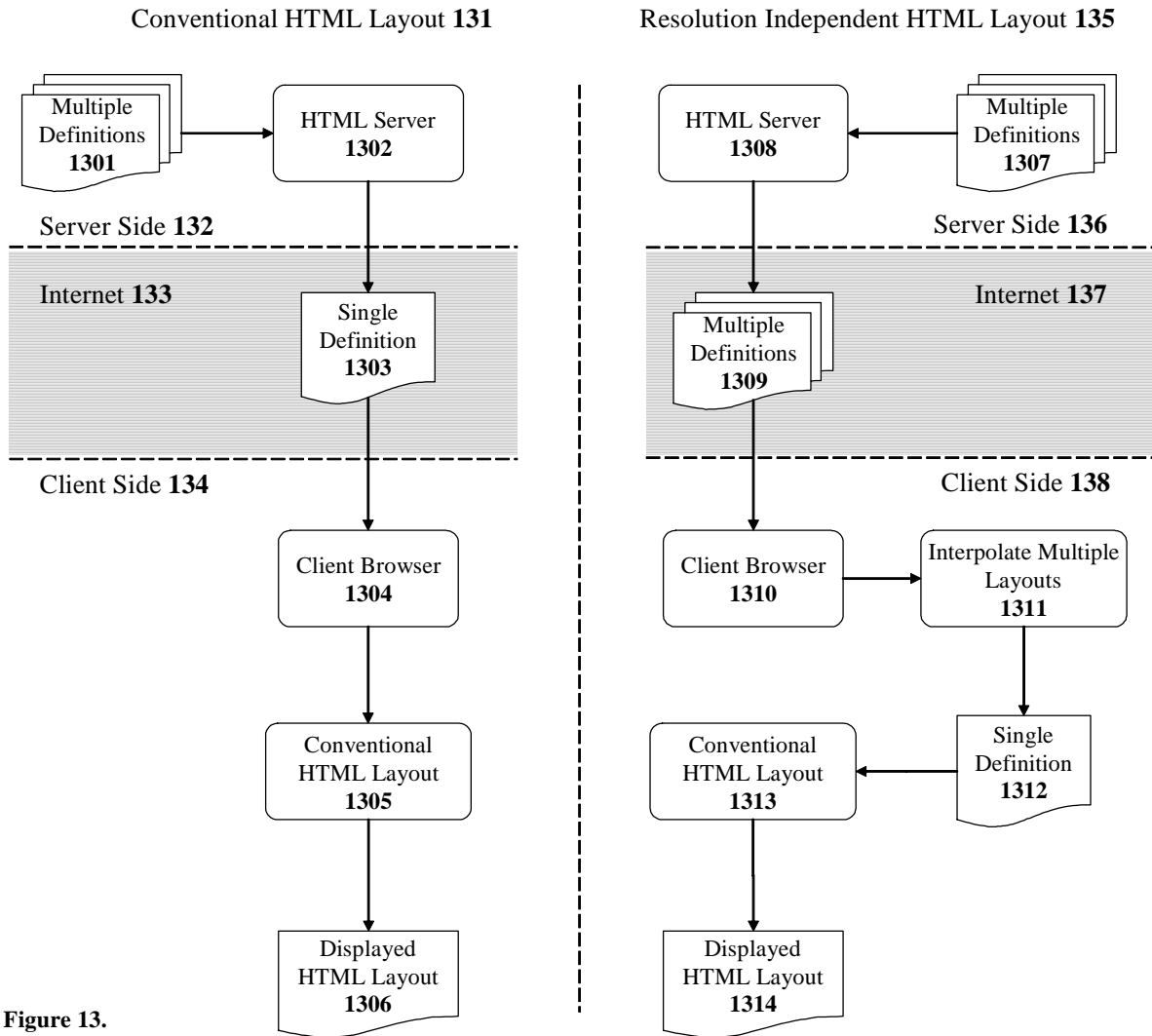
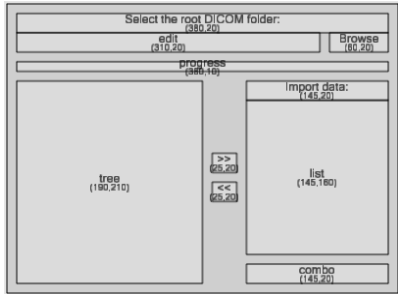


Figure 13.

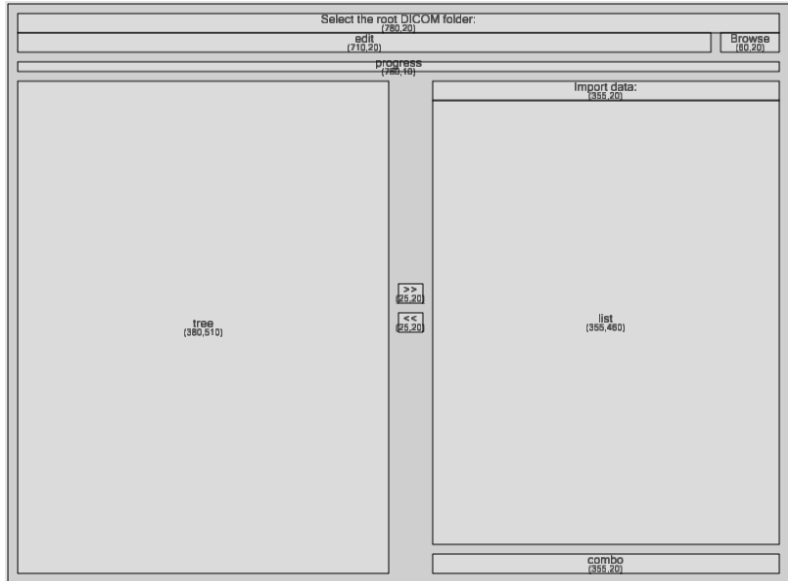
Fig. 13 shows conventional HTML layout 131 as compared to our resolution independent HTML layout 135. In both scenarios, there is a client side (134 and 138) that requests HTML data from the server side (132 and 136, respectively); the client and server sides communicate through the Internet (133 and 137, respectively). The conventional HTML layout method 131 can begin with either one or more layout definitions 1301; however, the HTML server 1302 provides only one layout definition 1303 to the client's browser 1304. This single layout then drives the conventional HTML layout process 1305 producing the result shown in the user's browser window 1306. To achieve resolution independence 135, the HTML server 1308 stores multiple layout definitions 137 and instead provides multiple layout definitions 1309 to the client's browser 1310. These definitions are then interpolated 1311 according to the client's browser size 1310 producing a single layout definition 1312. This single layout drives the conventional HTML layout process 1313 producing the result shown in the user's browser window 1314.

# Example User Interface Designs and Runtime Layouts

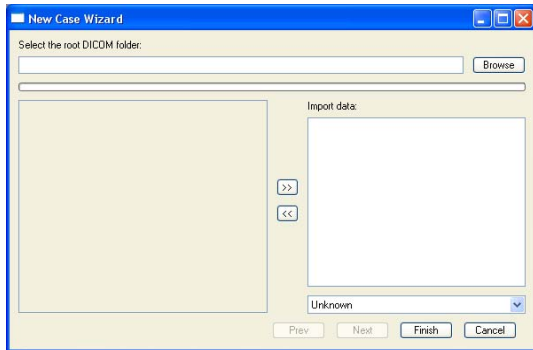
Layout Design Definition A  
(405 x 305) **1401**



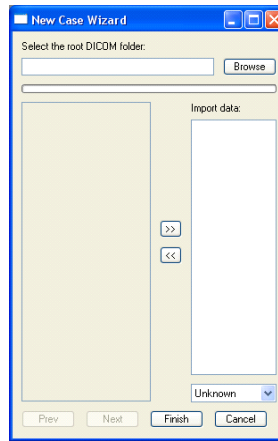
Layout Design Definition B  
(804 x 604) **1402**



Runtime Layout A  
(600 x 400) **1403**



Runtime Layout B  
(314 x 496) **1404**



Runtime Layout C  
(942 x 319) **1405**

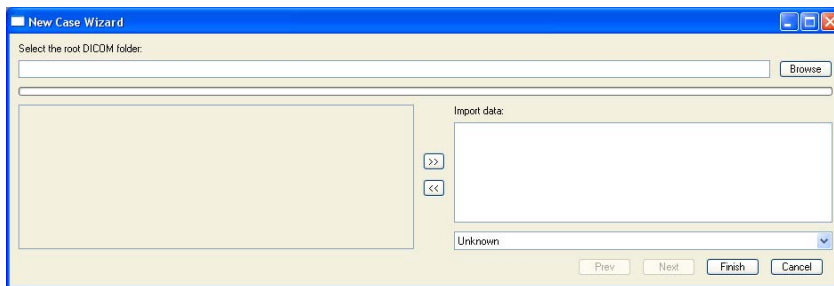


Figure 14.

Fig. 14 shows an example user interface layouts (1403 to 1405) interpolated from two layout design definitions (1401 and 1402). Definition 1401 and definition 1402 have several widget regions delineated in their respective layouts having each widget appear in both designs; however, the widgets are positioned in locations desirable for each layout size. These designs were used in an actual application and screenshots of that application are shown. Three runtime layouts 1403 to 1405 are interpolated from the two design definitions. The layout for this example is used on a wizard where navigation buttons “Prev,” “Next,” “Finish,” and “Cancel” have been added at the bottom using an unrelated mechanism and can be disregarded. The three runtime layouts have been fully determined by linearly interpolating between the two input definitions and were specified by the user simply resizing the wizard. These three runtime layouts represent the plurality of layouts that can be computed from the two input definitions.