Implementing Resolution Independence on top of the X Window System

Mark A. Linton

Abstract

A resolution-independent user interface system allows applications to support printing and different resolution displays without modifying any code. Because of the lack of direct support in the X Window System, we have implemented resolution independence on top of X in a client library. Our applications specify geometry and drawing requests in a real-world coordinate system, and the library translates the requests to pixel coordinates before passing them to X. We convert *positions* to pixel coordinates rather than *sizes*, avoiding unexpected gaps on the screen but allowing the size of an object in pixels to vary depending on where it is drawn. To support partial update, resolution-dependent information is available to application objects.

Our experience is that resolution-independence does not degrade interactive performance noticeably. Measurements indicate that our implementation of text drawing, which supports character positioning based on printer font metrics, is about 45% slower than drawing text using resolution-dependent calls to the X library.

Mark A. Linton (linton@sgi.com) is a Principal Scientist at Silicon Graphics, Inc.

1 Introduction

Resolution independence in a user interface means that the coordinate system used to specify sizes and positions is not tied to the coordinate system provided by the display. A document editor would ask for the width of a page object to be 6.5 inches, for example, instead of the corresponding number of pixels on a particular display.

The most important use of resolution independence is in WYSIWYG applications, which generate output to a printer that corresponds to the view presented on the display. Resolution independence also allows applications to run identically on screens with different resolutions.

The X Window System protocol, X library functions, and most X toolkits are resolution-dependent, defining graphics and window operations in terms of the pixels on the display. This characteristic of X has made some developers prefer to use other systems that offer resolution-independence, such as NeWS or Display PostScript.

Unlike these other systems, the InterViews toolkit[2] provides resolution independence on top of the X library. Our experience with InterViews is that resolution independence on top of X is convenient, practical, and efficient. In this paper, we describe how to implement resolution independence on top of X and analyze the effect of this approach on performance.

2 Programming model

The basic idea of resolution independence is that all coordinate values are in terms of a physical unit that is unrelated to the display. All graphics requests, window requests, and higher level toolkit operations (such as for geometry management) use coordinates in resolution-independent units. The implementation of these operations must translate the coordinates into pixel positions and sizes for the target device.

InterViews provides a data type called Coord for coordinate values. The unit coordinate is defined to be the size of a typographical point, about 1/72 of an inch. We chose points because they are a common unit in document processing and they are neutral with respect to locale. InterViews also provides scaling and rotation of coordinates, so an object can use other units when desired.

Ideally, Coord would be an abstract data type with the appropriate arithmetic operations. However, our implementation language, C++, does not provide adequate support for small objects. We therefore use a typedef of Coord to the built-in "float" type. We chose to use floating point instead of an integer representing very small units (Microsoft Windows[3] allows an application to use an integer representing a hundredth of a millimeter, for example) because floating point is more convenient for the programmer and the trend in recent CPUs has been to make the cost of a floating-point multiply close to the cost of an integer multiply.

The basic InterViews building block for defining a user interface is called a **glyph**. A **canvas** is a 2-dimensional surface to which a group of glyphs are attached. The glyphs negotiate for space on the canvas, draw on the canvas to refresh their appearance, and damage the canvas to cause an update traversal.

2.1 Canvases and Printers

The canvas class interface is similar to the PostScript drawing model. Operations are provided to specify a path as a list of points and to fill, stroke, or clip the current path. The canvas class also provides operations for drawing text and raster images. Printer is a subclass of Canvas that implements graphics operations by generating the appropriate PostScript text to a file. One can therefore write a single function for drawing to the screen and for generating PostScript suitable for printing. The InterViews document editor, for example, implements the print command by simply creating a printer object and passing it to the draw method for the root object of each page. The objects on the page use the identical code to draw to the printer that they use to draw to the display.

2.2 Fonts

Providing resolution-independent information for fonts introduces two problems. First, the resolution at which a font is designed does not necessarily match the resolution of the display. For example, suppose a user selects the 12-point Times Roman font provided for 75-dpi displays, but the user's display is 90-dpi. The metrics for this font specify a height of 12 points, but the actual height in pixels is only 10 points on the 90-dpi display ($12 * 75/90 = 10$).

Now suppose the user interactively drags a rectangle to be drawn around the 12-point text. If we compute the pixels per coordinate conversion factor assuming 90 dpi when drawing the rectangle, we will determine that the rectangle should be slightly larger than 10 points high. The result will look correct on the display, but on the printer the rectangle will be too small relative to the text.

The two possible solutions to this problem are to ignore the display resolution or to ignore the font metric information. The second solution may require more frequent font scaling, which may reduce the readability of text at small point sizes unless font outlines are available. InterViews implements the first solution, using the font resolution to compute the pixels per coordinate ratio. The conversion ratio also can be specified explicitly through the normal X defaults customization mechanism.

The second problem with fonts is that an application may wish to access resolution-dependent font metrics. For example, a document editor may provide an option to layout characters according to screen metrics, making the text look more regular on the display while sacrificing printed appearance. We solve this problem in InterViews by providing two kinds of font objects, one that returns display metrics and one that returns printer metrics.

2.3 Incremental update

InterViews glyphs may be shared and may overlap transparently (part of an object underneath can show through a transparent part of another object). Glyphs therefore are not allocated X subwindows and must cooperate to update the screen. A canvas manages a “damage” region that represents the update area. Logically, every update requires a full draw traversal over the objects in the window. In practice, some objects will short-circuit the traversal by determining that the damage region does not intersect their bounding boxes.

Unlike other coordinates, a damage region must be resolution-dependent to ensure correct update. For example, suppose object A draws at coordinate 10.5 and a object B draws after A at 10.7. Now suppose that A changes and generates damage for its update. If the destination display maps 10.5 and 10.7 to the same pixel, then the damage region must intersect the bounding box of B or else the overlapping pixel will not be updated correctly.

To support this need, InterViews defines a resolution-dependent region called an Extension. The coordinate units of an extension are points that correspond to pixels on the canvas’ device. Using 75-dpi conversion on the example above, 10.5 and 10.7 points are both 11 pixels and would be stored as 10.599 (11 * 72.27 / 75) in an extension.

Ideally, extensions should be computed automatically during drawing. Objects that needed to perform an update could retrieve their extension from the canvas after they were drawn. While this functionality simplifies update management, it complicates drawing and damage area management because glyphs need to be able to incur damage during drawing. Currently, a separate pass is made over the glyphs to compute extensions and incur damage.

3 Converting to pixels

The straightforward way to convert a coordinate to the corresponding number of pixels is to multiply by the pixels per coordinate ratio for the display and then round to the nearest integer. More sophisticated approaches to this problem use techniques such as anti-aliasing to reduce the visual effects of rounding. We have not explored trying such techniques on top of X; we first wanted experience with a simple approach. We therefore round to pixels coordinates and pass the pixel coordinates directly to X. Because the round operation must work for negative values to handle the case of drawing a polygon vertex outside the target window, a test is necessary in addition to an add and truncate. Scaling by the inverse factor is necessary to convert pixels in an X event back to coordinates, but no round operation is necessary.

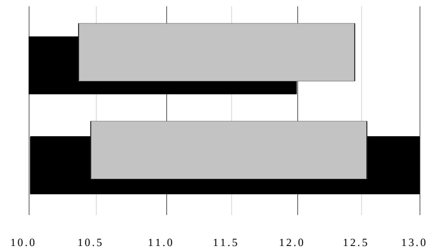

Figure 1: *Rounding positions can yield inconsistent sizes.*

3.1 *Placement versus size*

Rounding can cause anomalous results on low-resolution devices. Consider drawing a rectangle that is 2 points wide on a 75-dpi display. If the left edge of the rectangle is at 10.4 pixels, then the right edge is at 12.48 and rounding yields a 2-pixel wide rectangle. If instead the left edge were at 10.45 pixels, then the right edge is at 12.53 and rounding yields a 3-pixel wide rectangle. Figure 1 shows this problem visually. The light gray rectangles represent a 2-point wide rectangle and the dark gray rectangles underneath represent the rectangle in pixel coordinates. Rounding on the edges of the upper rectangle yields a 2-pixel wide rectangle; rounding on the lower rectangle yield a 3-pixel wide rectangle.

If we chose to round the size instead of the right edge of the rectangle, then we would always get the same number of pixels. However, we could also get a gap of one pixel between two rectangles that should abut. Figure 2 shows how this gap can occur. The two light gray rectangles at the top are specified in points. In the middle are the two rectangles with sizes rounded to pixel coordinates. At the bottom are the two rectangles with positions rounded.

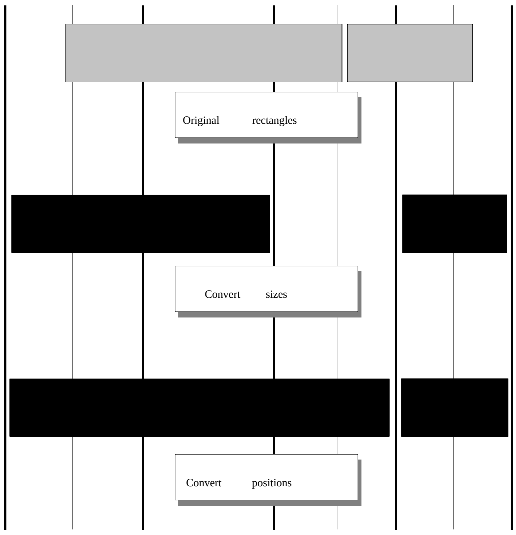

Figure 2: *Converting sizes versus positions.*

The InterViews implementation converts positions rather than sizes. Therefore, objects may not always be the same size depending on their position. Usually, a pixel difference in size is more desirable than a pixel gap between objects that are meant to abut. Some user interface components are not part of a printed representation, such as a text cursor in a document. In this case, it is desirable to specify the object size in pixel units. PostScript provides a “setstrokeadjust” operator to ensure consistent line widths[1]. InterViews simply makes available the display operations that convert between pixels and coordinates. For example, if a border object wishes to be 2 pixels wide on a display, it can ask the display for the size in coordinates of 2 pixels and pass that size to the canvas drawing operations.

3.2 Character positions

Document editors typically place each character at a coordinate position computed by a formatting algorithm. The positions are usually determined from printer font metrics, not display metrics. If the individual character positions are used for drawing, then it is possible that two adjacent characters may overlap on the display.

Our approach is to detect when the coordinate positions of two characters abut in the same line and accumulate the characters into a buffer to send in a single text item to X. When we find a space character, we begin a new text item at the correct position. The effect is that the white space absorbs the differences between display and printer font metrics. We also generate fewer X text items, which improves performance, though we also introduce the extra overhead of comparing coordinate positions for each character request.

Figure 3 shows part of a window dump of the InterViews document editor running on this paper. Figure 4 shows a window dump of a PostScript previewer on the PostScript for the same portion of the paper. The PostScript previewer uses the strategy of positioning each character individually. In the previewer, the word “translates” in the second line has the characters squeezed together. On the other hand, the document editor draws the words “translates the requests” so closely together that one might not realize there were spaces between the words.

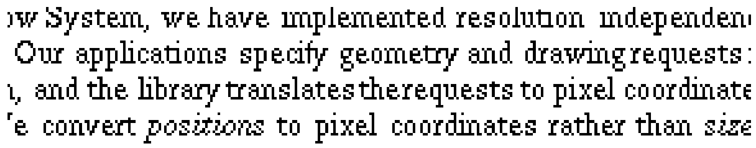

Figure 3: *Words as items.*

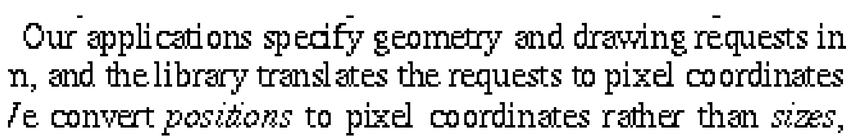

Figure 4: *Characters as items*

3.3 Window sizes

InterViews provides a window class for specifying top-level X windows and associated window manager information. The size and placement of a window is in resolution-independent coordinates, which must be converted to pixels before sending a request to X and must be converted back to coordinates in response to an X event such asConfigureNotify.

Windows have a desired size that most window managers will return in the absence of the user picking a different size. Because the size could have been rounded down in the conversion to coordinates, the objects in the window could get a smaller size than they requested. For example, suppose the natural size for a window is 148.5 points, which corresponds to 154.1 pixels. The window manager will return the requested 154 pixels, which is converted to 148.4 points. As a consequence, some object in the window will need to shrink by a small amount.

Our approach to this problem, though not currently implemented, is to store the round-off error in the window and add it to coordinates that correspond to window edges. This adjustment must be done on Expose events as well as ConfigureNotify events.

4 Performance

To evaluate the run-time cost of resolution-independence, we measured the cost of drawing text through InterViews to drawing text directly with X. Figure 5 shows the measurements of text drawing on an Indigo workstation. The first column is the drawing performance, measured in characters per millisecond. The XDrawText, XDrawString, and XDrawImageString measurements were collected using the x11perf program with the “-polytext”, “-ftext”, and “-fitext” options, respectively. The XDrawText case draws text from two different fonts, XDrawString draws a single, fixed-width font, and XDrawImageString draws both the text and the background behind the text. The second column contains the relative performance compared to the XDrawText case.

The InterViews benchmark computes the drawing times just like x11perf. The main loop simply calls the character operation on a canvas for each character to be drawn. Because the characters can be any font, this operation is closest in functionality to the XDrawTextbenchmark.

Benchmark	Characters/msec	Relative to XDrawText
InterViews	45.3	0.55
XDrawText	82.0	1.00
XDrawString	101.0	1.23
XDrawImageString	152.0	1.85

Figure 5: *Text drawing performance*.

Despite the overhead of a virtual function call for each character, copying the characters into a buffer, and testing the current position for each character, the InterViews performance benchmark is only about 45% slower than calling XDrawText directly. This ratio is a reflection of the ratio of CPU speed to rendering speed. If the text rendering were faster or the CPU slower, then the overhead would be higher.

Our experience building applications with InterViews is that the overhead of resolution independence is not noticeable for object such as buttons, menus, and dialog boxes. Objects for which drawing performance is important usually need resolution-independence anyway.

5 Conclusions

Resolution independence is an important capability of a user interface system that is missing from the standard X software. In the past, people have built resolution-independence at a low-level of the system.

Our work shows that it is practical and reasonably efficient to build resolution-independence on top of the existing X protocol. In doing so, we resolved issues involving font metrics, use of resolution-dependent bounding boxes for incremental update, coordinate roundoff for consistent placement, and positioning of characters. The implementation has been available as part of the InterViews distribution since January of 1990. The X-dependent portion of the code is about 6,000 lines of C++, of which about one quarter is in the implementation of the canvas class.

In an environment where the client machine is as fast as the server machine, implementing resolution-independence in the client will perform as well if not better than an implementation in the server. In the case where a server has support for resolution independence, we can simply retarget the InterViews canvas implementation to use the whatever mechanisms are available.

Acknowledgments

Paul Calder designed and implemented the resolution-independent mechanisms of InterViews. Dave Sternlicht wrote the InterViews benchmark program.

References

[1] Adobe Systems Incorporated. PostScript Language Reference Manual, Second Edition. Addison-Wesley, 1990.

[2] P. Calder and M. Linton. Glyphs: Flyweight objects for user interfaces. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 92-101, Snowbird, Utah, October 1990.

[3] C. Petzold. Programming Windows. Microsoft Press, 1990.