A Simple X11 Client Program

or

How hard can it really be to write "Hello, World"?

David S. H. Rosenthal

Sun Microsystems
2550 Garcia Ave.
Mountain View CA 94043

dshr@sun.com

ABSTRACT

The "Hello, World" program has achieved the status of a koan in the UNIX community. Various versions of this koan for Version 11 of the X Window System are examined, as a guide to writing correct programs, and as an illustration of the importance of toolkits in X11 programming.

'There, fourth graders discuss whether the machines prefer running simpler programs ("It's easier for them," "They hardly have to do any work.") or more complicated ones ("They feel proud," "It's like they are showing what they can do.").'

Sherry Turkle *The Second Self.*

1. Introduction

The "Hello, World" program has achieved the status of a koan in the UNIX[*] community. The spare elegance of:

```
#include <stdio.h>

main()
{
    printf("Hello, World\n");
}
```

has been much admired, and contrasted with the baroque complexity of other system's attempts to solve the problem.[2]

1. Permission to use, copy, modify and distribute this document for any purpose and without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies, and that the name of Sun Microsystems, Inc. not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. Sun Microsystems, Inc. makes no representations about the suitability of the software described herein for any purpose. It is provided "as is" without express or implied warranty.

* UNIX is a registered trademark of AT&T.

2 Of course, it is important to note that this implementation has bugs. A more correct, ANSI C version is shown in the appendix.

Various versions of the “Hello, World” koan for Version 11 of the X Window System* are examined. They provide a guide to writing correct programs for X11, insight into the complexity of the issues they have to deal with, and an illustration of the importance of toolkits in X11 programming.

2. Specification

The “Hello, World” problem needs to be restated slightly for the world of window systems. The program needs to:

- Create a window of an appropriate size, and position it on the screen if required.
- Paint the string “Hello, World” in a suitable font, centered in the window.
- Paint the string in a color which will contrast with the window background – it is not acceptable to paint an invisible string.
- Deal with its window being exposed – repainting the window if required.
- Deal with its window being resized – re-centering the text.
- Deal with its window being closed into an icon, and re-opened, providing suitable identification of the icon.

Note that this specification is much more complex than the canonical “Hello, World” program, so that it is likely that the resulting programs will be significantly more complex.

3. Vanilla X11

The first “Hello, World” example uses only the facilities of the basic X11 library.

3.1. Program Outline

- Open a connection to the X server. Exit gracefully if you cannot.
- Open the font to be used, and obtain the font data to allow string widths and heights to be calculated.
- Select pixel values for the window border, the window background, and the foreground that will be used to paint the characters.
- Compute the size and location of the window based on the text string and the font data, and set up the size hints structure.
- Create the window.
- Set up the standard properties for use by the window manager.
- Create a Graphics Context for use when painting the string.
- Select the types of input events that we are interested in receiving.
- Map the window to make it visible.
- Loop forever:
  - Obtain the next event.
  - If the event is the last event of a group of Expose events (that is, it has **count == 0**), repaint the window by:
    - Discovering the current size of the window.
    - Computing the position for the start of the string that will cause it to appear centered in the current window.
    - Clearing the window to the background color.
    - Drawing the string.

The X Window System is a trademark of the Massachusetts Institute of Technology.

3.2. Program Text

```
#include <stdio.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>

#define STRING  "Hello, world"
#define BORDER              1
#define FONT      "fixed"

/*
 * This structure forms the WM_HINTS property of the window,
 * letting the window manager know how to handle this window.
 * See Section 9.1 of the Xlib manual.
 */
XWMHints         xwmh = {
    (InputHint|StateHint),          /* flags */
    False,                                  /* input */
    NormalState,                    /* initial_state */
    0,                                              /* icon pixmap */
    0,                                              /* icon window */
    0, 0,                                   /* icon location */
    0,                                              /* icon mask */
    0,                                              /* Window group */
};

main(argc,argv)
    int argc;
    char **argv;
{
    Display    *dpy;                                /* X server connection */
    Window      win;                                /* Window ID */
    GC          gc;                         /* GC to draw with */
    XFontStruct *fontstruct;                /* Font descriptor */
    unsigned long fth, pad;                 /* Font size parameters */
    unsigned long fg, bg, bd;               /* Pixel values */
    unsigned long bw;                               /* Border width */
    XGCValues   gcv;                                /* Struct for creating GC */
    XEvent      event;                              /* Event received */
    XSizeHints  xsh;                                /* Size hints for window manager */
    char       *geomSpec;                   /* Window geometry string */
    XSetWindowAttributes xswa;      /* Temporary Set Window Attribute struct */

    /*
     * Open the display using the $DISPLAY environment variable to locate
     * the X server.  See Section 2.1.
     */
    if ((dpy = XOpenDisplay(NULL)) == NULL) {
                fprintf(stderr, "%s: can't open %s\n", argv[0], XDisplayName(NULL));
                exit(1);
    }

    /*
     * Load the font to use.  See Sections 10.2 & 6.5.1
     */
    if ((fontstruct = XLoadQueryFont(dpy, FONT)) == NULL) {
                fprintf(stderr, "%s: display %s doesn't know font %s\n",
                        argv[0], DisplayString(dpy), FONT);
                exit(1);
    }
    fth = fontstruct->max_bounds.ascent + fontstruct->max_bounds.descent;

    /*
     * Select colors for the border,  the window background,  and the
     * foreground.
     */
    bd = WhitePixel(dpy, DefaultScreen(dpy));
    bg = BlackPixel(dpy, DefaultScreen(dpy));
    fg = WhitePixel(dpy, DefaultScreen(dpy));

    /*
     * Set the border width of the window,  and the gap between the text
     * and the edge of the window, "pad".
     */
    pad = BORDER;
    bw = 1;

    /*
     * Deal with providing the window with an initial position & size.
     * Fill out the XSizeHints struct to inform the window manager. See
     * Sections 9.1.6 & 10.3.
     */
    xsh.flags = (PPosition | PSize);
    xsh.height = fth + pad * 2;
    xsh.width = XTextWidth(fontstruct, STRING, strlen(STRING)) + pad * 2;
    xsh.x = (DisplayWidth(dpy, DefaultScreen(dpy)) - xsh.width) / 2;
    xsh.y = (DisplayHeight(dpy, DefaultScreen(dpy)) - xsh.height) / 2;

    /*
     * Create the Window with the information in the XSizeHints, the
     * border width,  and the border & background pixels. See Section 3.3.
     */
    win = XCreateSimpleWindow(dpy, DefaultRootWindow(dpy),
                              xsh.x, xsh.y, xsh.width, xsh.height,
                              bw, bd, bg);

    /*
     * Set the standard properties for the window managers. See Section
     * 9.1.
     */
    XSetStandardProperties(dpy, win, STRING, STRING, None, argv, argc, &xsh);
    XSetWMHints(dpy, win, &xwmh);

    /*
     * Ensure that the window's colormap field points to the default
     * colormap,  so that the window manager knows the correct colormap to
     * use for the window.  See Section 3.2.9. Also,  set the window's Bit
```

```
         * Gravity to reduce Expose events.
         */
        xswa.colormap = DefaultColormap(dpy, DefaultScreen(dpy));
        xswa.bit_gravity = CenterGravity;
        XChangeWindowAttributes(dpy, win, (CWColormap | CWBitGravity), &xswa);

        /*
         * Create the GC for writing the text.  See Section 5.3.
         */
        gcv.font = fontstruct->fid;
        gcv.foreground = fg;
        gcv.background = bg;
        gc = XCreateGC(dpy, win, (GCFont | GCForeground | GCBackground), &gcv);

        /*
         * Specify the event types we're interested in - only Exposures.  See
         * Sections 8.5 & 8.4.5.1
         */
        XSelectInput(dpy, win, ExposureMask);

        /*
         * Map the window to make it visible.  See Section 3.5.
         */
        XMapWindow(dpy, win);

        /*
         * Loop forever,  examining each event.
         */
        while (1) {
                /*
                 * Get the next event
                 */
                XNextEvent(dpy, &event);

                /*
                 * On the last of each group of Expose events,  repaint the entire
                 * window.  See Section 8.4.5.1.
                 */
                if (event.type == Expose && event.xexpose.count == 0) {
                    XWindowAttributes xwa;          /* Temp Get Window Attribute struct */
                    int        x, y;

                    /*
                     * Remove any other pending Expose events from the queue to
                     * avoid multiple repaints. See Section 8.7.
                     */
                    while (XCheckTypedEvent(dpy, Expose, &event));

                    /*
                     * Find out how big the window is now,  so that we can center
                     * the text in it.
                     */
                    if (XGetWindowAttributes(dpy, win, &xwa) == 0)
                            break;
                    x = (xwa.width - XTextWidth(fontstruct, STRING, strlen(STRING))) / 2;
                    y = (xwa.height + fontstruct->max_bounds.ascent
                            - fontstruct->max_bounds.descent) / 2;

                    /*
                     * Fill the window with the background color,  and then paint
                     * the centered string.
                     */
                    XClearWindow(dpy, win);
                    XDrawString(dpy, win, gc, x, y, STRING, strlen(STRING));
                }
        }

        exit(1);
}
```

3.3. Does it meet the specification?

- It computes the size of the window from the string and the font, and positions it at the center of the screen.
- It paints the string in the color **WhitePixel( )** on a **BlackPixel( )** background. It ensures that the appropriate colormap will be used for the window, so that these colors (which may not actually be White and Black) will be distinguishable.
- Every time it gets the last of a group of Expose events, it enquires the size of the window, and paints the string centered in this space. In particular, it will get an Expose event initially as a consequence of its mapping the window, and will thus paint the window for the first time.
- The same mechanism copes with part or all of the window being exposed. The program will re-paint the entire window when any part is exposed; in this case the effort of only repainting the exposed parts is excessive.
- The fact that the string is re-centered every time the window is painted means that the program deals with re-sizing correctly. Subject to the caveats below, when the window is resized, an Expose event will be generated, and the window will be re-painted.

- The standard properties that the program sets include a specification of a string that a window manager can display in the icon. The programs sets this to be the string it is displaying, so it copes with being iconified. When it is opened, an Expose event will be generated, and the window will be re-painted.

3.4. Design Issues

Although this implementation of “Hello, World” is alarmingly long, it is structurally simple. Nevertheless, there are many detailed design issues that arise when writing it. This section covers them, in no particular order.

3.4.1. Repaint Strategy.

Every X11 application has the responsibility for re-painting its image whenever the server requests it to. It is possible to refresh only the parts requested, or to refresh the entire window. The “Hello, World” image is simple enough that refreshing the entire image is a sensible approach.

Exposing part or all the window results in the server painting the exposed areas in the window background color, and one or more Expose events. Each carries, in the **count** field, the number of events in the same group that follow it. After receiving the last of each group, identified by a zero **count**, the window is re-painted.

Re-painting on *every* Expose event would result in unnecessary multiple repaints. For example, consider a “Hello, World” that appears for the first time with one corner overlain by another window. The newly exposed area consists of two rectangles, so there will be two Expose events in the initial group.

We actually take even more rigorous measures to avoid multiple repaints. Every time we decide to repaint the window, we scan the event queue and remove all Expose events that have arrived at the client, but which have yet to arrive at the head of the queue.

3.4.2. Resize Strategy

The first time the window is painted, it seems as if enquiring the size of the window is unnecessary. We have, after all, just created the window and told it what size to be. But X11 does not allow us to assume that the window will actually get created at the requested size; we have to be prepared for a window manager to have intervened and overridden our choice of size. So it is necessary to enquire the window size on the initial Expose event.

When the window is resized, the client needs to re-compute the centering of the text. The implementation does this on the last of every group of Expose events. This raises two questions

- Does every resize of the window result in at least one Expose event?

  Consider a window, not obscured by others, that is resized to make it smaller. The X11 server actually has enough pixels to fill the new window size; there is no need to generate an Expose event to cause pixels to be repainted. This is the simplest example of what the X11 specification calls “Bit Gravity”. Clients may reduce the number of Expose events they receive by specifying an appropriate Bit Gravity. Even if the window is made larger, the Bit Gravity can tell the server how to re-locate the old pixels in the new window to avoid Expose events on parts of the window whose contents are not supposed to change.

  By default, X11 sets the Bit Gravity of windows to **ForgetGravity**. This ensures that the gravity mechanism is disabled, Expose events occur on all resizings of the window, and the “Hello, World” program operates correctly if the whole issue of Bit Gravity is ignored. In this default case, the answer to the question is “Yes, every resize results in at least one Expose event”.

  But we can exploit Bit Gravity to avoid unnecessary repaints by setting it to **CenterGravity**. This will preserve the centering of the text if the window is resized smaller without involving the client program. In this case, the answer to the question is “No, not every resize results in an Expose event”. But in the cases where no Expose event occurs, the window will still be correct.

- Can we avoid the overhead of enquiring the window size on every re-paint?

  As we have seen, Expose events have no direct relation to window re-sizing. In general, X11 clients should listen for both:

- Expose events, which tell them to re-paint some part of the window.
- ConfigureNotify events, which tell them that the window has been changed in some way which requires that the image be re-computed. These carry the new size of the window, there is no need for an explicit enquiry.

In principle, “Hello, World” should only re-compute the centering of the text when it gets a ConfigureNotify event. But the overhead of the extra round-trip to the server to enquire about the size of the window on every Expose event is not critical for this application, and the code in this case is much simpler.

3.4.3. Communicating with the Window Manager

Every X application must use some properties on its window to communicate with the window manager.

- The WM hints, containing information for the window manager about the input and icon behaviour of the window. In the case of “Hello, World”, this information is known at compile time and can be intialized statically.
- The size hints, containing information about the size and position of the window. These cannot be initialized statically since they depend on the font properties, and are only known at run-time.
- Other properties, including **WM_NAME**, **WM_ICON_NAME**, and **WM_COMMAND**, which are used to communicate strings such as the name of the program running behind the window.
- The colormap field of the window is not a property, but it may also be used to communicate with the window manager. The conventions about this have yet to be fully specified, and the topic is covered in the next section.

3.4.4. Pixel Values and Colormaps

To ensure “Hello, World” works on both monochrome and color displays, we use the colors Black and White[3]. To paint in a color using X11, you need to know the pixel value corresponding to it; the pixel value is the number you write into the pixel to cause that color to show on the screen.

Although X11 specifies that Graphics Contexts are by default created using 0 and 1 for the background and foreground pixel values, an application cannot predict the colors that these pixel values will resolve to. It cannot even predict that these two colors will be different, so every application must explicitly set the pixel values it will use.

Pixel values are determined relative to a Colormap; X11 supports an arbitrary number of colormaps, with one or more being installed in the hardware at any one time. X11 supports these colormaps even on monochrome displays. There is a default colormap, which applications with modest color requirements are urged to use, and “Hello, World” is as modest as you could wish. In fact, the colors Black and White have pre-defined pixel values in the default colormap, and we can use these directly.

However, using the pre-defined values means that “Hello, World” becomes dependent on having the default colormap installed. Unless it is, they may not be distinguishable. Unfortunately, when a window is created using **XCreateSimpleWindow( )** the colormap is inherited from the parent (in our case, the root window for the default screen), and it is possible that some client may have set the colormap of the root to something other than the default colormap. So, for now, we have to set the colormap field of the window explicitly to be the default colormap, although it is anticipated that when the conventions for window management are finally determined this code will be unnecessary.

Simply setting the colormap field of the window does not ensure that the correct colormap will actually be installed. The whole question of whether, and when, clients should install their colormaps is open to debate at present. There are two basic positions:

- Clients should explicitly install their own colormap when appropriate, for example when they obtain the input focus.

3 These colors may not actually be black and white, but they are guaranteed to contrast with each other.

This has two disadvantages, in that it makes every client much more complicated (it means, for example, that "Hello, World" has to worry about the input focus!), and that it means that every client will be doing the wrong thing eventually, when window managers start doing the right thing (whatever that is!). It does, however, mean that clients will work right now.

- Clients should never install their own colormaps, and should assume that some combination of the internals of the server, and the window manager, will do it for them.

  This has the disadvantage that it will not work at present, since existing window managers don't appear to do anything with colormaps.

Strictly speaking, therfore, "Hello, World" should deal with installing colormaps, since the policy has yet to be determined. But it would make the code so complex as to be out of the question for a paper such as this.

3.4.5. Error Handling

It appears that this "Hello, World" implementation follows the canonical "Hello, World" implementation in the great UNIX tradition of optimism, by ignoring the possibility of errors. Not so, the question of error handling has been fully considered:

- On the **XOpenDisplay( )** call, we check for the error return, and exit gracefully.
- When opening the font, we cannot be sure that the server will map that name into a font. So we check the error return, and exit gracefully, if the server objects to the name.
- For all other errors, we depend on the default error handling mechanism. When an X11 client gets an Error event from the server, the library code invokes an error handler. The client is free to override the default one, which prints an informative message and exits, but its behaviour is fine for "Hello, World".

Of course, one might ask why we need to explicitly check for errors on opening the font. Surely, the default error handler does what we want? It is an (alas, undocumented) feature of Xlib that not all errors cause the error handler to be invoked. Some errors, such as failure to open a font, are regarded as failure status returns and are signalled by Status return values – in general any routine that returns Status will need its return tested, because it will have bypassed the error handling mechanism.

3.4.6. Finding a Server.

The particular server to use is identified by the **$DISPLAY** environment variable[4], so it does not need to be specified explicitly. It is a convention among X11 clients that a command-line argument containing a colon is a specification of the server to use, but this version of "Hello, World" does not support the convention (or any other command-line arguments).

3.4.7. Looping for events.

It is natural to assume that you can write the event wait loop:

```
while (XNextEvent(dpy, &event) {
   ......
}
```

but this is not the case. **XNextEvent( )** is defined to be void; it only ever returns when there is an event to return, and errors are handled through the error handling mechanism, rather than being indicated by a return value. So it is necessary to write the event loop:

```
while (1) {
   XNextEvent(dpy, &event);
   ......
}
```

4 Non-UNIX systems will use some other technique.

3.4.8. Centering the Text

There are a number of ways to compute the size of the string, in order to center it in the window. The most correct method is to use **XTextExtents( )**, which computes not merely the width of the string, but also the maximum and minimum Y valuse for the characters in the string. The example uses **XTextWidth( )** to compute the width of the string, and uses the maximum and minimum Y values over all characters in the font. It is to some extent a matter of taste which looks better, the string "Hello, World" has no descenders so the example will tend to locate it somewhat higher in the window than is visually correct.

3.5. Protocol Usage

Benchmarking X11 applications is an interesting problem. The performance as seen by the user is affected both by the client, and by the server. To measure the performance of clients independently of any server, I have instrumented the X11 library to gather statistics on the usage of the protocol. The results for the vanilla X11 "Hello, World" program are shown in Tables 1 (aggregate statistics) and 2 (usage of individual requests).

Table 1: Aggregate Statistics - Vanilla

Statistic	Value
number of writes	4
number of reads	12
bytes written	392
bytes read	1736
number of requests	16
number of errors	0
number of events	1
number of replies	3

Table 2: Usage of Requests - Vanilla

Request	Count
CreateWindow	1
ChangeWindowAttributes	2
GetWindowAttributes	1
MapWindow	1
GetGeometry	1
ChangeProperty	5
OpenFont	1
QueryFont	1
CreateGC	1
ClearArea	1
PolyText8	1

In interpreting these tables, the important thing to remember is that a round-trip to the server (a request that needs a reply) is relatively expensive. For example, the "replies" entry in Table 1 shows that there were 3 round-trips, and in Table 2 they can be identified as being the GetWindowAttributes, GetGeometry, and QueryFont requests.

This brings out an interesting point. Where did the GetGeometry request come from? The answer is that the **XGetWindowAttributes( )** X11 library call uses both the GetWindowAttributes and GetGeometry protocol request. It is easy to assume that there is a one-to-one mapping between X11 library calls and protocol requests, but this is not the case. The use of **XGetWindowAttributes( )** in this "Hello, World" program is inefficient, **XGetGeometry( )** should be used instead.

4. Defaults

This program wires-in a large number of parameters, through the following statements:

```
if ((fontstruct = XLoadQueryFont(dpy, FONT)) == NULL) {
bd = WhitePixel(dpy, DefaultScreen(dpy));
bg = BlackPixel(dpy, DefaultScreen(dpy));
fg = WhitePixel(dpy, DefaultScreen(dpy));
pad = BORDER;
bw = 1;

xsh.height = fth + pad * 2;
xsh.width = XTextWidth(fontstruct, STRING, strlen(STRING)) + pad * 2;
xsh.x = (DisplayWidth(dpy, DefaultScreen(dpy)) - xsh.width) / 2;
xsh.y = (DisplayHeight(dpy, DefaultScreen(dpy)) - xsh.height) / 2;
```

Ideally, and certainly for any real application, the user should be able to override these wired-in defaults. X11 supplies a default database mechanism to address this problem.

4.1. Program

Here is the first example, modified to use the X11 default database mechanism to allow the user to specify values for all these defaults. For each of the first group, it uses **XGetDefault( )** to obtain a string value, and then parses it (using **XParseColor( )**, or **atoi( )**) to the required value. In the case of the window geometry values in the second group, X11 provides a single mechanism (**XGeometry( )**) to parse a string into some or all of the parameters specifying the geometry of a window. To save space, and because the changes to deal with defaults are restricted to a small part of the code, they are presented as a context diff.

```
*** xhw0.c      Sun Dec  6 08:25:11 1987
--- xhw1.c      Sun Dec  6 08:25:12 1987
***************
*** 5,10 ****
--- 5,17 ----
  #define STRING                  "Hello, world"
  #define BORDER                  1
  #define FONT    "fixed"
+ #define       ARG_FONT                        "font"
+ #define       ARG_BORDER_COLOR                "bordercolor"
+ #define       ARG_FOREGROUND                                  "foreground"
+ #define       ARG_BACKGROUND                                  "background"
+ #define ARG_BORDER                            "border"
+ #define       ARG_GEOMETRY                                    "geometry"
+ #define DEFAULT_GEOMETRY""

  /*
   * This structure forms the WM_HINTS property of the window,
***************
*** 29,38 ****
      Display    *dpy;                          /* X server connection */
      Window     win;                           /* Window ID */
      GC         gc;                  /* GC to draw with */
      XFontStruct *fontstruct;        /* Font descriptor */
!     unsigned long fth, pad;         /* Font size parameters */
      unsigned long fg, bg, bd;       /* Pixel values */
      unsigned long bw;                         /* Border width */
      XGCValues   gcv;                          /* Struct for creating GC */
      XEvent     event;                         /* Event received */
      XSizeHints  xsh;                          /* Size hints for window manager */
--- 36,49 ----
      Display    *dpy;                          /* X server connection */
      Window     win;                           /* Window ID */
      GC         gc;                  /* GC to draw with */
+     char       *fontName;           /* Name of font for string */
      XFontStruct *fontstruct;        /* Font descriptor */
!     unsigned long ftw, fth, pad;/* Font size parameters */
      unsigned long fg, bg, bd;       /* Pixel values */
      unsigned long bw;                         /* Border width */
+     char       *tempstr;            /* Temporary string */
+     XColor     color;                         /* Temporary color */
+     Colormap   cmap;                          /* Color map to use */
      XGCValues   gcv;                          /* Struct for creating GC */
      XEvent     event;                         /* Event received */
      XSizeHints  xsh;                          /* Size hints for window manager */
***************
*** 51,77 ****
      /*
       * Load the font to use.  See Sections 10.2 & 6.5.1
       */
!     if ((fontstruct = XLoadQueryFont(dpy, FONT)) == NULL) {
              fprintf(stderr, "%s: display %s doesn't know font %s\n",
!                     argv[0], DisplayString(dpy), FONT);
              exit(1);
      }
      fth = fontstruct->max_bounds.ascent + fontstruct->max_bounds.descent;

      /*
       * Select colors for the border,  the window background,  and the
!      * foreground.
       */
!     bd = WhitePixel(dpy, DefaultScreen(dpy));
!     bg = BlackPixel(dpy, DefaultScreen(dpy));
!     fg = WhitePixel(dpy, DefaultScreen(dpy));
!
      /*
       * Set the border width of the window,  and the gap between the text
       * and the edge of the window, "pad".
       */
```

```
	pad = BORDER;
!	bw = 1;

	/*
	 * Deal with providing the window with an initial position & size.
--- 62,117 ----
	/*
	 * Load the font to use.  See Sections 10.2 & 6.5.1
	 */
!	if ((fontName = XGetDefault(dpy, argv[0], ARG_FONT)) == NULL) {
!		fontName = FONT;
!	}
!	if ((fontstruct = XLoadQueryFont(dpy, fontName)) == NULL) {
		fprintf(stderr, "%s: display %s doesn't know font %s\n",
!				argv[0], DisplayString(dpy), fontName);
		exit(1);
	}
	fth = fontstruct->max_bounds.ascent + fontstruct->max_bounds.descent;
+	ftw = fontstruct->max_bounds.width;

	/*
	 * Select colors for the border,  the window background,  and the
!	 * foreground.  We use the default colormap to allocate the colors in.
!	 * See Sections 2.2.1, 5.1.2, & 10.4.
	 */
!	cmap = DefaultColormap(dpy, DefaultScreen(dpy));
!	if ((tempstr = XGetDefault(dpy, argv[0], ARG_BORDER_COLOR)) == NULL ||
!		XParseColor(dpy, cmap, tempstr, &color) == 0 ||
!		XAllocColor(dpy, cmap, &color) == 0) {
!		bd = WhitePixel(dpy, DefaultScreen(dpy));
!	}
!	else {
!		bd = color.pixel;
!	}
!	if ((tempstr = XGetDefault(dpy, argv[0], ARG_BACKGROUND)) == NULL ||
!		XParseColor(dpy, cmap, tempstr, &color) == 0 ||
!		XAllocColor(dpy, cmap, &color) == 0) {
!		bg = BlackPixel(dpy, DefaultScreen(dpy));
!	}
!	else {
!		bg = color.pixel;
!	}
!	if ((tempstr = XGetDefault(dpy, argv[0], ARG_FOREGROUND)) == NULL ||
!		XParseColor(dpy, cmap, tempstr, &color) == 0 ||
!		XAllocColor(dpy, cmap, &color) == 0) {
!		fg = WhitePixel(dpy, DefaultScreen(dpy));
!	}
!	else {
!		fg = color.pixel;
!	}
	/*
	 * Set the border width of the window,  and the gap between the text
	 * and the edge of the window, "pad".
	 */
	pad = BORDER;
!	if ((tempstr = XGetDefault(dpy, argv[0], ARG_BORDER)) == NULL)
!		bw = 1;
!	else
!		bw = atoi(tempstr);

	/*
	 * Deal with providing the window with an initial position & size.
***************
*** 78,88 ****
	 * Fill out the XSizeHints struct to inform the window manager. See
	 * Sections 9.1.6 & 10.3.
	 */
!	xsh.flags = (PPosition | PSize);
!	xsh.height = fth + pad * 2;
!	xsh.width = XTextWidth(fontstruct, STRING, strlen(STRING)) + pad * 2;
!	xsh.x = (DisplayWidth(dpy, DefaultScreen(dpy)) - xsh.width) / 2;
!	xsh.y = (DisplayHeight(dpy, DefaultScreen(dpy)) - xsh.height) / 2;

	/*
	 * Create the Window with the information in the XSizeHints, the
--- 118,150 ----
	 * Fill out the XSizeHints struct to inform the window manager. See
	 * Sections 9.1.6 & 10.3.
	 */
!	geomSpec = XGetDefault(dpy, argv[0], ARG_GEOMETRY);
!	if (geomSpec == NULL) {
!		/*
!		 * The defaults database doesn't contain a specification of the
!		 * initial size & position - fit the window to the text and locate
!		 * it in the center of the screen.
!		 */
!		xsh.flags = (PPosition | PSize);
!		xsh.height = fth + pad * 2;
!		xsh.width = XTextWidth(fontstruct, STRING, strlen(STRING)) + pad * 2;
!		xsh.x = (DisplayWidth(dpy, DefaultScreen(dpy)) - xsh.width) / 2;
!		xsh.y = (DisplayHeight(dpy, DefaultScreen(dpy)) - xsh.height) / 2;
!	}
!	else {
!		int	bitmask;
!
!		bzero(&xsh, sizeof(xsh));
!		bitmask = XGeometry(dpy, DefaultScreen(dpy), geomSpec, DEFAULT_GEOMETRY,
!					bw, ftw, fth, pad, pad, &(xsh.x), &(xsh.y),
!					&(xsh.width), &(xsh.height));
!		if (bitmask & (XValue | YValue)) {
!		    xsh.flags |= USPosition;
!		}
!		if (bitmask & (WidthValue | HeightValue)) {
!		    xsh.flags |= USSize;
!		}
!	}

	/*
```

```
* Create the Window with the information in the XSizeHints, the
```

4.2. Protocol Usage

Turning statistics gathering on for this version, we get Tables 3 and 4. The extra replies come from the mappings between string names and colors for the foreground, background, and border. These mappings could have been done with a single AllocNamedColor request each, instead of a LookupColor/AllocColor pair, but this would not have supported the convention that colors can be specified by strings like “#3a7”.

Table 3: Aggregate Statistics - Defaults

Statistic	Value
number of writes	10
number of reads	24
bytes written	504
bytes read	1928
number of requests	22
number of errors	0
number of events	1
number of replies	9

Table 4: Usage of Requests - Defaults

Request	Count
CreateWindow	1
ChangeWindowAttributes	2
GetWindowAttributes	1
MapWindow	1
GetGeometry	1
ChangeProperty	5
OpenFont	1
QueryFont	1
CreateGC	1
CleaarArea	1
PolyText8	1
AllocColor	3
LookupColor	3

5. The Toolkit

Examining the preceding examples, anyone would admit that the basic X11 library fails the “Hello, World” test. Even the simplest “Hello, World” client takes 40 executable statements, and 25 calls through the X11 library interface.

All is not lost, however. It was never intended that normal applications programmers would use the basic X11 library interface. An analogy is that very few UNIX programmers use the raw system call interface, they almost all use the higher-level “Standard I/O Library” interface. The canonical “Hello, World” program is an example.

The X11 distribution includes a user interface toolkit, intended to provide a more congenial environment for applications development in exactly the same way that *stdio* does for vanilla UNIX. Using this toolkit, the following example shows that X11 can pass the “Hello, World” test with ease.

5.1. Program Outline

The outline of the program is:

- Create the top level Widget that represents the toolkit’s view of the (top-level) window.

- Create a Label Widget to display the string, over-riding the defaults database to set the Label's value to the string to display.
- Tell the top level Widget to display the label, by adding it to the top level Widget's managed list.
- Realize the top level Widget (and therefore its sub-Widgets). This process creates an X11 window for each Widget, setting its attributes from the data in the Widget.
- Loop forever, processing the events that appear.

5.2. Program Text

```
#include <stdio.h>
#include <X11/Xlib.h>
#include <X11/Intrinsic.h>
#include <X11/Atoms.h>
#include <X11/Label.h>

#define         STRING          "Hello, World"

Arg wargs[] = {
    XtNlabel,       (XtArgVal) STRING,
};

main(argc, argv)
    int argc;
    char **argv;
{
    Widget      toplevel, label;

    /*
     * Create the Widget that represents the window.
     * See Section 14 of the Toolkit manual.
     */
    toplevel = XtInitialize(argv[0], "XLabel", NULL, 0, &argc, argv);

    /*
     * Create a Widget to display the string, using wargs to set
     * the string as its value. See Section 9.1.
     */
    label = XtCreateWidget(argv[0], labelWidgetClass,
                                        toplevel, wargs, XtNumber(wargs));

    /*
     * Tell the toplevel widget to display the label. See Section 13.5.2.
     */
    XtManageChild(label);

    /*
     * Create the windows, and set their attributes according
     * to the Widget data. See Section 9.2.
     */
    XtRealizeWidget(toplevel);

    /*
     * Now process the events. See Section 16.6.2.
     */
    XtMainLoop();
}
```

5.3. Does it meet the specification?

This implementation of ‘‘Hello, World’’ fulfills the specification:

- The window is sized as a result of the geometry negotiation between the top level Widget and its sub-Widgets (in this case the label), so that by default the window is sized to fit the text.
- The default attributes for the Label Widget specify that the text is centered, and the default mechanism supplies a suitable font.
- In the same way, the default mechanism supplies background and foreground colors for the Widget.
- The toolkit manages all Expose events, routing them to appropriate Widgets. Thus, the program behaves correctly for exposure.
- The Label Widget recomputes the centering of the text whenever it is being painted, so that resizing is handled correctly.
- The toolkit handles communicating with the window manager about icon properties, so that iconification is handled correctly.

5.4. Design Issues

Note that this implementation isn't merely much shorter than the earlier examples. It has an additional useful feature, in that any or all the values from the default database used by the program can be overridden by command line arguments. **XtInitialize( )** parses the command line and merges any specifiers it finds there with the defaults database.

The toolkit also provides peace of mind by organizing the error handling correctly. Although the documentation of error handling in the toolkit manual is sparse, experiments seem to show that the implementation is satisfactory, providing intelligible messages and sensible behaviour.

5.5. Protocol Usage

Turning statistics gathering on for the toolkit version gives Tables 5 and 6.

Table 5: Aggregate Statistics - Toolkit

Statistic	Value
number of writes	11
number of reads	25
bytes written	832
bytes read	2012
number of requests	29
number of errors	0
number of events	4
number of replies	9

Table 6: Usage of Requests - Toolkit

Request	Count
CreateWindow	2
MapWindow	1
MapSubwindows	1
ConfigureWindow	1
InternAtom	2
ChangeProperty	5
OpenFont	1
QueryFont	1
CreatePixmap	3
CreateGC	3
FreeGC	1
PutImage	1
PolyText8	1
AllocColor	3
LookupColor	3

These tables reveal that:

- Use of the toolkit does not result in significantly greater protocol traffic.
- The toolkit does not use GetWindowAttributes or Get Geometry. Its repaint and resize strategies use the information in Expose and ConfigureNotify events, and don't require round-trips.

To compare the performance of the toolkit and vanilla versions, we can contrast the cost of being resized by the *uwm* window manager. This is more meaningful than the total cost since startup, since in general window system clients are well advised to invest extra effort in startup code to improve response to interactions.

Table 7 shows that the simplistic repaint and resize strategies of the Vanilla version cost significantly more in terms of the number of round-trips (4 vs. 0) and the amount of data transferred (344 vs. 272)[5].

5 The reasons why resizing with *uwm* results in several events are too complex to go into here (i.e. both uwm and the server have bugs).

Table 7: Cost of resize with *uwm*

Item	Vanilla	Toolkit
number of writes	6	4
number of reads	15	4
bytes written	128	112
bytes read	248	160
number of requests	8	4
number of events	3	5
number of replies	4	0
GetWindowAttributes	2	0
GetGeometry	2	0
ConfigureWindow	0	1
ClearArea	2	0
PolyText8	2	3

6. Conclusions

These examples demonstrate that programming applications using only the basic X library interface is even more difficult and unrewarding than programming UNIX applications using only the system call interface.

Observing the usage of X11 protocol requests gives a server-independent measure of X11 client performance. This can be a useful tool in debugging and performance-tuning X11 clients, the more so in that the performance of interactive clients is likely to be dominated by the number of round-trips per interaction.

Just as anyone considering developing UNIX applications should use *stdio*, anyone considering developing X11 applications should use a toolkit. There is (at least) one in the distribution, and others are available from other sources. Using a toolkit, you can expect:

- isolation from complex and, as yet, undecided design issues about the interaction between X11 clients and their environment,
- competitive performance, at least in terms of protocol usage,
- and much less verbose and more maintainable source code.

In the case of ‘‘Hello, World’’, a client that took 40 executable statements to program using the basic X11 library took 5 statements to program using the X11 toolkit. And the toolkit version had more functionality and better repaint performance than a library version with 60 statements.

Acknowledgements

Richard Johnson posted the first attempt at a ‘‘Hello, World’’ program for X11 to the *xpert* mail list. This, and Ellis Cohen’s praiseworthy attempts to write up the conventions needed for communicating between applications and window managers, inspired me to try to write the ‘‘Hello, World’’ program right.

Jim Gettys, Bob Scheifler, & Mark Opperman all identified bugs and suggested fixes in the Vanilla version. The toolkit version is derived from *lib/Xtk/clients/xlabel*.

Appendix: ANSI C Hello World

```
#include <stdio.h>

main()
{
    (void) printf("Hello, World\n");
    exit (0);
}
```

- ANSI C specifies that printf() returns the number of characters printed.
- It is necessary to exit() or return() a value from main().