Basic Graphics Programming With The XCB Library

January 11, 2023

This tutorial is based on the Xlib Tutorial written by Guy Keren. The author allowed me to take some parts of his text, mainly the text which deals with the X Windows generality.

This tutorial is intended for people who want to start to program with the XCB library. keep in mind that XCB, like the Xlib library, isn't what most programmers wanting to write X applications are looking for. They should use a much higher level GUI toolkit like Motif, LessTiff, GTK, QT, EWL, ETK, or use Cairo. However, we need to start somewhere. More than this, knowing how things work down below is never a bad idea.

After reading this tutorial, one should be able to write very simple graphical programs, but not programs with decent user interfaces. For such programs, one of the previously mentioned libraries should be used.

But what is XCB? Xlib has been the standard C binding for the X Window System protocol for many years now. It is an excellent piece of work, but there are applications for which it is not ideal, for example:

- **Small platforms**: Xlib is a large piece of code, and it's difficult to make it smaller
- **Latency hiding**: Xlib requests requiring a reply are effectively synchronous: they block until the reply appears, whether the result is needed immediately or not.
- **Direct access to the protocol**: Xlib does quite a bit of caching, layering, and similar optimizations. While this is normally a feature, it makes it difficult to simply emit specified X protocol requests and process specific responses.
- **Threaded applications**: While Xlib does attempt to support multithreading, the API makes this difficult and error-prone.
- **New extensions**: The Xlib infrastructure provides limited support for the new creation of X extension client side code.

For these reasons, among others, XCB, an X C binding, has been designed to solve the above problems and thus provide a base for

- Toolkit implementation.
- Direct protocol-level programming.
- Lightweight emulation of commonly used portions of the Xlib API.

2. The client and server model of the X window system

The X Window System was developed with one major goal: flexibility. The idea was that the way things look is one thing, but the way things work is another matter. Thus, the lower levels provide the tools required to draw windows, handle user input, allow drawing graphics using colors (or black and white screens), etc. To this point, a decision was made to separate the system into two parts. A client that decides what to do, and a server that actually draws on the screen and reads user input in order to send it to the client for processing.

This model is the complete opposite of what is used to when dealing with clients and servers. In our case, the user sits near the machine controlled by the server, while the client might be running on a remote machine. The server controls the screens, mouse and keyboard. A

client may connect to the server, request that it draws a window (or several windows), and ask the server to send it any input the user sends to these windows. Thus, several clients may connect to a single X server (one might be running mail software, one running a WWW browser, etc). When input is sent by the user to some window, the server sends a message to the client controlling this window for processing. The client decides what to do with this input, and sends the server requests for drawing in the window.

The whole session is carried out using the X message protocol. This protocol was originally carried over the TCP/IP protocol suite, allowing the client to run on any machine connected to the same network that the server is. Later on, the X servers were extended to allow clients running on the local machine with more optimized access to the server (note that an X protocol message may be several hundreds of KB in size), such as using shared memory, or using Unix domain sockets (a method for creating a logical channel on a Unix system between two processes).

3. GUI programming: the asynchronous model

   Unlike conventional computer programs, that carry some serial nature, a GUI program usually uses an asynchronous programming model, also known as "event-driven programming". This means that that program mostly sits idle, waiting for events sent by the X server, and then acts upon these events. An event may say "The user pressed the 1st button mouse in spot (x,y)", or "The window you control needs to be redrawn". In order for the program to be responsive to the user input, as well as to refresh requests, it needs to handle each event in a rather short period of time (e.g. less that 200 milliseconds, as a rule of thumb).

   This also implies that the program may not perform operations that might take a long time while handling an event (such as opening a network connection to some remote server, or connecting to a database server, or even performing a long file copy operation). Instead, it needs to perform all these operations in an asynchronous manner. This may be done by using various asynchronous models to perform the longish operations, or by performing them in a different process or thread.

   So the way a GUI program looks is something like that:

   (a) Perform initialization routines.
   (b) Connect to the X server.
   (c) Perform X-related initialization.
   (d) While not finished:
      i. Receive the next event from the X server.
      ii. Handle the event, possibly sending various drawing requests to the X server.
      iii. If the event was a quit message, exit the loop.
   (e) Close down the connection to the X server.
   (f) Perform cleanup operations.

4. Basic XCB notions

   XCB has been created to eliminate the need for programs to actually implement the X protocol layer. This library gives a program a very low-level access to any X server. Since the protocol is standardized, a client using any implementation of XCB may talk with any X server (the same occurs for Xlib, of course). We now give a brief description of the basic XCB notions. They will be detailed later.

   (a) The X Connection

The major notion of using XCB is the X Connection. This is a structure representing the connection we have open with a given X server. It hides a queue of messages coming from the server, and a queue of pending requests that our client intends to send to the server. In XCB, this structure is named 'xcb_connection_t'. It is analogous to the Xlib Display. When we open a connection to an X server, the library returns a pointer to such a structure. Later, we supply this pointer to any XCB function that should send messages to the X server or receive messages from this server.

(b) Requests and replies: the Xlib killers

To ask for information from the X server, we have to make a request and ask for a reply. With Xlib, these two tasks are automatically done: Xlib locks the system, sends a request, waits for a reply from the X server and unlocks. This is annoying, especially if one makes a lot of requests to the X server. Indeed, Xlib has to wait for the end of a reply before asking for the next request (because of the locks that Xlib sends). For example, here is a time-line of N=4 requests/replies with Xlib, with a round-trip latency **T_round_trip** that is 5 times long as the time required to write or read a request/reply (**T_write/T_read**):

```
W-----RW-----RW-----RW-----R
```

- W: Writing request
- -: Stalled, waiting for data
- R: Reading reply

The total time is N * (T_write + T_round_trip + T_read).

With XCB, we can suppress most of the round-trips as the requests and the replies are not locked. We usually send a request, then XCB returns to us a **cookie**, which is an identifier. Then, later, we ask for a reply using this **cookie** and XCB returns a pointer to that reply. Hence, with XCB, we can send a lot of requests, and later in the program, ask for all the replies when we need them. Here is the time-line for 4 requests/replies when we use this property of XCB:

```
WWWW--RRRR
```

The total time is N * T_write + max (0, T_round_trip - (N-1) * T_write) + N * T_read. Which can be considerably faster than all those Xlib round-trips.

Here is a program that computes the time to create 500 atoms with Xlib and XCB. It shows the Xlib way, the bad XCB way (which is similar to Xlib) and the good XCB way. On my computer, XCB is 25 times faster than Xlib.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>

#include <xcb/xcb.h>

#include <X11/Xlib.h>

double
get_time(void)
{
  struct timeval timev;

  gettimeofday(&timev, NULL);

```

```
17    return (double)timev.tv_sec + (((double)timev.tv_usec) / 1000000);
18  }
19
20  int
21  main ()
22  {
23    xcb_connection_t         *c;
24    xcb_atom_t               *atoms;
25    xcb_intern_atom_cookie_t *cs;
26    char                     **names;
27    int                      count;
28    int                      i;
29    double                   start;
30    double                   end;
31    double                   diff;
32
33    /* Xlib */
34    Display *disp;
35    Atom    *atoms_x;
36    double   diff_x;
37
38    c = xcb_connect (NULL, NULL);
39
40    count = 500;
41    atoms = (xcb_atom_t *)malloc (count * sizeof (atoms));
42    names = (char **)malloc (count * sizeof (char *));
43
44    /* init names */
45    for (i = 0; i < count; ++i) {
46      char buf[100];
47
48      sprintf (buf, "NAME%d", i);
49      names[i] = strdup (buf);
50    }
51
52    /* bad use */
53    start = get_time ();
54
55    for (i = 0; i < count; ++i)
56      atoms[i] = xcb_intern_atom_reply (c,
57                                        xcb_intern_atom (c,
58                                                         0,
59                                                         strlen(names[i]),
60                                                         names[i]),
61                                        NULL)->atom;
62
63    end = get_time ();
64    diff = end - start;
65    printf ("bad use time  : %f\n", diff);
66
67    /* good use */
68    start = get_time ();
69
70    cs = (xcb_intern_atom_cookie_t *) malloc (count *
      ↪  sizeof(xcb_intern_atom_cookie_t));
```

```
71    for(i = 0; i < count; ++i)
72      cs[i] = xcb_intern_atom (c, 0, strlen(names[i]), names[i]);
73
74    for(i = 0; i < count; ++i) {
75      xcb_intern_atom_reply_t *r;
76
77      r = xcb_intern_atom_reply(c, cs[i], 0);
78      if(r)
79        atoms[i] = r->atom;
80      free(r);
81    }
82
83    end = get_time ();
84    printf ("good use time : %f\n", end - start);
85    printf ("ratio         : %f\n", diff / (end - start));
86    diff = end - start;
87
88    /* free var */
89    free (atoms);
90    free (cs);
91
92    xcb_disconnect (c);
93
94    /* Xlib */
95    disp = XOpenDisplay (getenv("DISPLAY"));
96
97    atoms_x = (Atom *)malloc (count * sizeof (atoms_x));
98
99    start = get_time ();
100
101   for (i = 0; i < count; ++i)
102     atoms_x[i] = XInternAtom(disp, names[i], 0);
103
104   end = get_time ();
105   diff_x = end - start;
106   printf ("Xlib use time : %f\n", diff_x);
107   printf ("ratio         : %f\n", diff_x / diff);
108
109   free (atoms_x);
110   for (i = 0; i < count; ++i)
111     free (names[i]);
112   free (names);
113
114   XCloseDisplay (disp);
115
116   return 0;
117 }
```

(c) The Graphic Context

When we perform various drawing operations (graphics, text, etc), we may specify various options for controlling how the data will be drawn (what foreground and background colors to use, how line edges will be connected, what font to use when drawing some text, etc). In order to avoid the need to supply hundreds of parameters to each drawing function, a graphical context structure is used. We set the various drawing options in this structure, and then we pass a pointer to this structure to any drawing routines. This is rather handy,

as we often need to perform several drawing requests with the same options. Thus, we would initialize a graphical context, set the desired options, and pass this structure to all drawing functions.

Note that graphic contexts have no client-side structure in XCB, they're just XIDs. Xlib has a client-side structure because it caches the GC contents so it can avoid making redundant requests, but of course XCB doesn't do that.

(d) Events

A structure is used to pass events received from the X server. XCB supports exactly the events specified in the protocol (33 events). This structure contains the type of event received (including a bit for whether it came from the server or another client), as well as the data associated with the event (e.g. position on the screen where the event was generated, mouse button associated with the event, region of the screen associated with a "redraw" event, etc). The way to read the event's data depends on the event type.

5. Using XCB-based programs

(a) Installation of XCB

TODO: These instructions are out of date. Just reference the main XCB page so we don't have to maintain these instructions in more than one place.

To build XCB from source, you need to have installed at least:

- pkgconfig 0.15.0
- automake 1.7
- autoconf 2.50
- check
- xsltproc
- gperf 3.0.1

You have to checkout in the git repository the following modules:

- Xau from xlibs
- xcb-proto
- xcb

Note that xcb-proto exists only to install header files, so typing 'make' or 'make all' will produce the message "Nothing to be done for 'all'". That's normal.

(b) Compiling XCB-based programs

Compiling XCB-based programs requires linking them with the XCB library. This is easily done thanks to pkgconfig:

```
1 gcc -Wall prog.c -o prog `pkg-config --cflags --libs xcb`
```

6. Opening and closing the connection to an X server

An X program first needs to open the connection to the X server. There is a function that opens a connection. It requires the display name, or NULL. In the latter case, the display name will be the one in the environment variable DISPLAY.

```
1 xcb_connection_t *xcb_connect (const char *displayname,
2                                int        *screenp);
```

The second parameter returns the screen number used for the connection. The returned structure describes an XCB connection and is opaque. Here is how the connection can be opened:

```
#include <xcb/xcb.h>

int
main ()
{
  xcb_connection_t *c;

  /* Open the connection to the X server. Use the DISPLAY environment variable as
   ↪  the default display name */
  c = xcb_connect (NULL, NULL);

  return 0;
}
```

To close a connection, it suffices to use:

```
void xcb_disconnect (xcb_connection_t *c);
```

Comparison Xlib/XCB

- XOpenDisplay ()
- xcb_connect ()
- XCloseDisplay ()
- xcb_disconnect ()

7. Checking basic information about a connection

Once we have opened a connection to an X server, we should check some basic information about it: what screens it has, what is the size (width and height) of the screen, how many colors it supports (black and white ? grey scale ?, 256 colors ? more ?), and so on. We get such information from the xcb_screen_t structure:

```
typedef struct {
    xcb_window_t   root;
    xcb_colormap_t default_colormap;
    uint32_t       white_pixel;
    uint32_t       black_pixel;
    uint32_t       current_input_masks;
    uint16_t       width_in_pixels;
    uint16_t       height_in_pixels;
    uint16_t       width_in_millimeters;
    uint16_t       height_in_millimeters;
    uint16_t       min_installed_maps;
    uint16_t       max_installed_maps;
    xcb_visualid_t root_visual;
    uint8_t        backing_stores;
    uint8_t        save_unders;
    uint8_t        root_depth;
    uint8_t        allowed_depths_len;
} xcb_screen_t;
```

We could retrieve the first screen of the connection by using the following function:

```
xcb_screen_iterator_t xcb_setup_roots_iterator (xcb_setup_t *R);
```

Here is a small program that shows how to use this function:

```
#include <stdio.h>

#include <xcb/xcb.h>

int
main ()
{
  xcb_connection_t     *c;
  xcb_screen_t         *screen;
  int                   screen_nbr;
  xcb_screen_iterator_t iter;

  /* Open the connection to the X server. Use the DISPLAY environment variable */
  c = xcb_connect (NULL, &screen_nbr);

  /* Get the screen #screen_nbr */
  iter = xcb_setup_roots_iterator (xcb_get_setup (c));
  for (; iter.rem; --screen_nbr, xcb_screen_next (&iter))
    if (screen_nbr == 0) {
      screen = iter.data;
      break;
    }

  printf ("\n");
  printf ("Informations of screen %ld:\n", screen->root);
  printf ("  width.........: %d\n", screen->width_in_pixels);
  printf ("  height........: %d\n", screen->height_in_pixels);
  printf ("  white pixel...: %ld\n", screen->white_pixel);
  printf ("  black pixel...: %ld\n", screen->black_pixel);
  printf ("\n");

  return 0;
}
```

8. Creating a basic window - the ”hello world” program

   After we got some basic information about our screen, we can create our first window. In the X Window System, a window is characterized by an Id. So, in XCB, a window is of type:

```
typedef uint32_t xcb_window_t;
```

We first ask for a new Id for our window, with this function:

```
xcb_window_t xcb_generate_id(xcb_connection_t *c);
```

Then, XCB supplies the following function to create new windows:

```
xcb_void_cookie_t xcb_create_window (xcb_connection_t *c,             /* Pointer to
 ↪  the xcb_connection_t structure */
                                     uint8_t           depth,         /* Depth of
 ↪  the screen */
                                     xcb_window_t      wid,           /* Id of the
 ↪  window */
                                     xcb_window_t      parent,        /* Id of an
 ↪  existing window that should be the parent of the new window */
```

```
                                    int16_t           x,             /* X position
    ↪ of the top-left corner of the window (in pixels) */
                                    int16_t           y,             /* Y position
    ↪ of the top-left corner of the window (in pixels) */
                                    uint16_t          width,         /* Width of
    ↪ the window (in pixels) */
                                    uint16_t          height,        /* Height of
    ↪ the window (in pixels) */
                                    uint16_t          border_width,  /* Width of
    ↪ the window's border (in pixels) */
                                    uint16_t          _class,
                                    xcb_visualid_t    visual,
                                    uint32_t          value_mask,
                                    const uint32_t   *value_list);
```

The fact that we created the window does not mean that it will be drawn on screen. By default, newly created windows are not mapped on the screen (they are invisible). In order to make our window visible, we use the function xcb_map_window(), whose prototype is

```
xcb_void_cookie_t xcb_map_window (xcb_connection_t *c,
                                  xcb_window_t      window);
```

Finally, here is a small program to create a window of size 150x150 pixels, positioned at the top-left corner of the screen:

```
#include <unistd.h>      /* pause() */

#include <xcb/xcb.h>

int
main ()
{
  xcb_connection_t *c;
  xcb_screen_t     *screen;
  xcb_window_t      win;

  /* Open the connection to the X server */
  c = xcb_connect (NULL, NULL);

  /* Get the first screen */
  screen = xcb_setup_roots_iterator (xcb_get_setup (c)).data;

  /* Ask for our window's Id */
  win = xcb_generate_id(c);

  /* Create the window */
  xcb_create_window (c,                             /* Connection          */
                     XCB_COPY_FROM_PARENT,          /* depth (same as root)*/
                     win,                           /* window Id           */
                     screen->root,                  /* parent window       */
                     0, 0,                          /* x, y                */
                     150, 150,                      /* width, height       */
                     10,                            /* border_width        */
                     XCB_WINDOW_CLASS_INPUT_OUTPUT, /* class               */
                     screen->root_visual,           /* visual              */
                     0, NULL);                      /* masks, not used yet */

```

```
33    /* Map the window on the screen */
34    xcb_map_window (c, win);
35
36    /* Make sure commands are sent before we pause, so window is shown */
37    xcb_flush (c);
38
39    pause ();    /* hold client until Ctrl-C */
40
41    return 0;
42  }
```

In this code, you see one more function - xcb_flush(), not explained yet. It is used to flush all the pending requests. More precisely, there are 2 functions that do such things. The first one is xcb_flush():

```
1  int xcb_flush (xcb_connection_t *c);
```

This function flushes all pending requests to the X server (much like the fflush() function is used to flush standard output). The second function is xcb_aux_sync():

```
1  int xcb_aux_sync (xcb_connection_t *c);
```

This functions also flushes all pending requests to the X server, and then waits until the X server finishing processing these requests. In a normal program, this will not be necessary (we'll see why when we get to write a normal X program), but for now, we put it there.

The window that is created by the above code has a non defined background. This one can be set to a specific color, thanks to the two last parameters of xcb_create_window(), which are not described yet. See the subsections Configuring a window or Registering for event types using event masks for examples on how to use these parameters. In addition, as no events are handled, you have to make a Ctrl-C to interrupt the program.

TODO: one should tell what these functions return and about the generic error

Comparison Xlib/XCB

- XCreateWindow ()
- xcb_generate_id ()
- xcb_create_window ()

9. Drawing in a window

Drawing in a window can be done using various graphical functions (drawing pixels, lines, rectangles, etc). In order to draw in a window, we first need to define various general drawing parameters (what line width to use, which color to draw with, etc). This is done using a graphical context.

(a) Allocating a Graphics Context

As we said, a graphical context defines several attributes to be used with the various drawing functions. For this, we define a graphical context. We can use more than one graphical context with a single window, in order to draw in multiple styles (different colors, different line widths, etc). In XCB, a Graphics Context is, as a window, characterized by an Id:

```
1  typedef uint32_t xcb_gcontext_t;
```

We first ask the X server to attribute an Id to our graphic context with this function:

```
xcb_gcontext_t xcb_generate_id (xcb_connection_t *c);
```

Then, we set the attributes of the graphic context with this function:

```
xcb_void_cookie_t xcb_create_gc (xcb_connection_t *c,
                                 xcb_gcontext_t    cid,
                                 xcb_drawable_t    drawable,
                                 uint32_t          value_mask,
                                 const uint32_t   *value_list);
```

We give now an example on how to allocate a graphic context that specifies that each drawing function that uses it will draw in foreground with a black color.

```
#include <xcb/xcb.h>

int
main ()
{
  xcb_connection_t *c;
  xcb_screen_t     *screen;
  xcb_drawable_t    win;
  xcb_gcontext_t    black;
  uint32_t          mask;
  uint32_t          value[1];

  /* Open the connection to the X server and get the first screen */
  c = xcb_connect (NULL, NULL);
  screen = xcb_setup_roots_iterator (xcb_get_setup (c)).data;

  /* Create a black graphic context for drawing in the foreground */
  win = screen->root;
  black = xcb_generate_id (c);
  mask = XCB_GC_FOREGROUND;
  value[0] = screen->black_pixel;
  xcb_create_gc (c, black, win, mask, value);

  return 0;
}
```

Note should be taken regarding the role of ”value_mask” and ”value_list” in the prototype of xcb_create_gc(). Since a graphic context has many attributes, and since we often just want to define a few of them, we need to be able to tell the xcb_create_gc() which attributes we want to set. This is what the ”value_mask” parameter is for. We then use the ”value_list” parameter to specify actual values for the attribute we defined in ”value_mask”. Thus, for each constant used in ”value_list”, we will use the matching constant in ”value_mask”. In this case, we define a graphic context with one attribute: when drawing (a point, a line, etc), the foreground color will be black. The rest of the attributes of this graphic context will be set to their default values.

See the next Subsection for more details.

Comparison Xlib/XCB

- XCreateGC ()
- xcb_generate_id ()
- xcb_create_gc ()

(b) Changing the attributes of a Graphics Context

Once we have allocated a Graphic Context, we may need to change its attributes (for example, changing the foreground color we use to draw a line, or changing the attributes of the font we use to display strings. See Subsections Drawing with a color and Assigning a Font to a Graphic Context). This is done by using this function:

```
1 xcb_void_cookie_t xcb_change_gc (xcb_connection_t *c,           /* The XCB
  ↪ Connection */
2                                  xcb_gcontext_t    gc,          /* The Graphic
  ↪ Context */
3                                  uint32_t          value_mask,  /* Components
  ↪ of the Graphic Context that have to be set */
4                                  const uint32_t   *value_list); /* Value as
  ↪ specified by value_mask */
```

The value_mask parameter could take any combination of these masks from the xcb_gc_t enumeration:

- XCB_GC_FUNCTION
- XCB_GC_PLANE_MASK
- XCB_GC_FOREGROUND
- XCB_GC_BACKGROUND
- XCB_GC_LINE_WIDTH
- XCB_GC_LINE_STYLE
- XCB_GC_CAP_STYLE
- XCB_GC_JOIN_STYLE
- XCB_GC_FILL_STYLE
- XCB_GC_FILL_RULE
- XCB_GC_TILE
- XCB_GC_STIPPLE
- XCB_GC_TILE_STIPPLE_ORIGIN_X
- XCB_GC_TILE_STIPPLE_ORIGIN_Y
- XCB_GC_FONT
- XCB_GC_SUBWINDOW_MODE
- XCB_GC_GRAPHICS_EXPOSURES
- XCB_GC_CLIP_ORIGIN_X
- XCB_GC_CLIP_ORIGIN_Y
- XCB_GC_CLIP_MASK
- XCB_GC_DASH_OFFSET
- XCB_GC_DASH_LIST
- XCB_GC_ARC_MODE

It is possible to set several attributes at the same time (for example setting the attributes of a font and the color which will be used to display a string), by OR'ing these values in value_mask. Then value_list has to be an array which lists the value for the respective attributes. **These values must be in the same order as masks listed above.** See Subsection Drawing with a color to have an example.

TODO: set the links of the 3 subsections, once they will be written :)

TODO: give an example which sets several attributes.

(c) Drawing primitives: point, line, box, circle,...

After we have created a Graphic Context, we can draw on a window using this Graphic Context, with a set of XCB functions, collectively called "drawing primitives". Let see how they are used.

To draw a point, or several points, we use

```
xcb_void_cookie_t xcb_poly_point (xcb_connection_t  *c,              /* The connection to the X server */
                                  uint8_t            coordinate_mode, /* Coordinate mode, usually set to XCB_COORD_MODE_ORIGIN */
                                  xcb_drawable_t     drawable,        /* The drawable on which we want to draw the point(s) */
                                  xcb_gcontext_t     gc,              /* The Graphic Context we use to draw the point(s) */
                                  uint32_t           points_len,      /* The number of points */
                                  const xcb_point_t *points);         /* An array of points */
```

The coordinate_mode parameter specifies the coordinate mode. Available values are

- XCB_COORD_MODE_ORIGIN
- XCB_COORD_MODE_PREVIOUS

If XCB_COORD_MODE_PREVIOUS is used, then all points but the first one are relative to the immediately previous point.

The xcb_point_t type is just a structure with two fields (the coordinates of the point):

```
typedef struct {
    int16_t x;
    int16_t y;
} xcb_point_t;
```

You could see an example in xpoints.c. **TODO** Set the link.

To draw a line, or a polygonal line, we use

```
xcb_void_cookie_t xcb_poly_line (xcb_connection_t  *c,              /* The connection to the X server */
                                 uint8_t            coordinate_mode, /* Coordinate mode, usually set to XCB_COORD_MODE_ORIGIN */
                                 xcb_drawable_t     drawable,        /* The drawable on which we want to draw the line(s) */
                                 xcb_gcontext_t     gc,              /* The Graphic Context we use to draw the line(s) */
                                 uint32_t           points_len,      /* The number of points in the polygonal line */
                                 const xcb_point_t *points);         /* An array of points */
```

This function will draw the line between the first and the second points, then the line between the second and the third points, and so on.

To draw a segment, or several segments, we use

```
xcb_void_cookie_t xcb_poly_segment (xcb_connection_t    *c,              /* The connection to the X server */
                                    xcb_drawable_t       drawable,       /* The drawable on which we want to draw the segment(s) */
```

```
3                                       xcb_gcontext_t       gc,              /* The
   ↪ Graphic Context we use to draw the segment(s) */
4                                       uint32_t             segments_len,    /* The
   ↪ number of segments */
5                                       const xcb_segment_t *segments);       /* An
   ↪ array of segments */
```

The xcb_segment_t type is just a structure with four fields (the coordinates of the two points that define the segment):

```
typedef struct {
    int16_t x1;
    int16_t y1;
    int16_t x2;
    int16_t y2;
} xcb_segment_t;
```

To draw a rectangle, or several rectangles, we use

```
xcb_void_cookie_t xcb_poly_rectangle (xcb_connection_t      *c,              /*
   ↪ The connection to the X server */
                                      xcb_drawable_t         drawable,       /*
   ↪ The drawable on which we want to draw the rectangle(s) */
                                      xcb_gcontext_t         gc,             /*
   ↪ The Graphic Context we use to draw the rectangle(s) */
                                      uint32_t               rectangles_len, /*
   ↪ The number of rectangles */
                                      const xcb_rectangle_t *rectangles);    /*
   ↪ An array of rectangles */
```

The xcb_rectangle_t type is just a structure with four fields (the coordinates of the top-left corner of the rectangle, and its width and height):

```
typedef struct {
    int16_t  x;
    int16_t  y;
    uint16_t width;
    uint16_t height;
} xcb_rectangle_t;
```

To draw an elliptical arc, or several elliptical arcs, we use

```
xcb_void_cookie_t xcb_poly_arc (xcb_connection_t *c,          /* The connection
   ↪ to the X server */
                                xcb_drawable_t    drawable,   /* The drawable
   ↪ on which we want to draw the arc(s) */
                                xcb_gcontext_t    gc,         /* The Graphic
   ↪ Context we use to draw the arc(s) */
                                uint32_t          arcs_len,   /* The number of
   ↪ arcs */
                                const xcb_arc_t  *arcs);      /* An array of
   ↪ arcs */
```

The xcb_arc_t type is a structure with six fields:

```
typedef struct {
    int16_t  x;       /* Top left x coordinate of the rectangle surrounding the
   ↪ ellipse */
    int16_t  y;       /* Top left y coordinate of the rectangle surrounding the
   ↪ ellipse */
```

```
    uint16_t width;   /* Width of the rectangle surrounding the ellipse */
    uint16_t height;  /* Height of the rectangle surrounding the ellipse */
    int16_t  angle1;  /* Angle at which the arc begins */
    int16_t  angle2;  /* Angle at which the arc ends */
} xcb_arc_t;
```

Note: the angles are expressed in units of 1/64 of a degree, so to have an angle of 90 degrees, starting at 0, angle1 = 0 and angle2 = 90 « 6. Positive angles indicate counterclockwise motion, while negative angles indicate clockwise motion.

The corresponding function which fill inside the geometrical object are listed below, without further explanation, as they are used as the above functions.

To Fill a polygon defined by the points given as arguments , we use

```
xcb_void_cookie_t xcb_fill_poly (xcb_connection_t  *c,
                                 xcb_drawable_t     drawable,
                                 xcb_gcontext_t     gc,
                                 uint8_t            shape,
                                 uint8_t            coordinate_mode,
                                 uint32_t           points_len,
                                 const xcb_point_t *points);
```

The shape parameter specifies a shape that helps the server to improve performance. Available values are

- XCB_POLY_SHAPE_COMPLEX
- XCB_POLY_SHAPE_NONCONVEX
- XCB_POLY_SHAPE_CONVEX

To fill one or several rectangles, we use

```
xcb_void_cookie_t xcb_poly_fill_rectangle (xcb_connection_t      *c,
                                           xcb_drawable_t         drawable,
                                           xcb_gcontext_t         gc,
                                           uint32_t
  ↪ rectangles_len,
                                           const xcb_rectangle_t *rectangles);
```

To fill one or several arcs, we use

```
xcb_void_cookie_t xcb_poly_fill_arc (xcb_connection_t *c,
                                     xcb_drawable_t    drawable,
                                     xcb_gcontext_t    gc,
                                     uint32_t          arcs_len,
                                     const xcb_arc_t  *arcs);
```

To illustrate these functions, here is an example that draws four points, a polygonal line, two segments, two rectangles and two arcs. Remark that we use events for the first time, as an introduction to the next section.

TODO: Use screen->root_depth for depth parameter.

```
#include <stdlib.h>
#include <stdio.h>

#include <xcb/xcb.h>

```

```c
int
main ()
{
  xcb_connection_t    *c;
  xcb_screen_t        *screen;
  xcb_drawable_t       win;
  xcb_gcontext_t       foreground;
  xcb_generic_event_t *e;
  uint32_t             mask = 0;
  uint32_t             values[2];

  /* geometric objects */
  xcb_point_t          points[] = {
    {10, 10},
    {10, 20},
    {20, 10},
    {20, 20}};

  xcb_point_t          polyline[] = {
    {50, 10},
    { 5, 20},     /* rest of points are relative */
    {25,-20},
    {10, 10}};

  xcb_segment_t        segments[] = {
    {100, 10, 140, 30},
    {110, 25, 130, 60}};

  xcb_rectangle_t      rectangles[] = {
    { 10, 50, 40, 20},
    { 80, 50, 10, 40}};

  xcb_arc_t            arcs[] = {
    {10, 100, 60, 40, 0, 90 << 6},
    {90, 100, 55, 40, 0, 270 << 6}};

  /* Open the connection to the X server */
  c = xcb_connect (NULL, NULL);

  /* Get the first screen */
  screen = xcb_setup_roots_iterator (xcb_get_setup (c)).data;

  /* Create black (foreground) graphic context */
  win = screen->root;

  foreground = xcb_generate_id (c);
  mask = XCB_GC_FOREGROUND | XCB_GC_GRAPHICS_EXPOSURES;
  values[0] = screen->black_pixel;
  values[1] = 0;
  xcb_create_gc (c, foreground, win, mask, values);

  /* Ask for our window's Id */
  win = xcb_generate_id(c);

  /* Create the window */
```

```
61    mask = XCB_CW_BACK_PIXEL | XCB_CW_EVENT_MASK;
62    values[0] = screen->white_pixel;
63    values[1] = XCB_EVENT_MASK_EXPOSURE;
64    xcb_create_window (c,                             /* Connection          */
65                       XCB_COPY_FROM_PARENT,          /* depth               */
66                       win,                           /* window Id           */
67                       screen->root,                  /* parent window       */
68                       0, 0,                          /* x, y                */
69                       150, 150,                      /* width, height       */
70                       10,                            /* border_width        */
71                       XCB_WINDOW_CLASS_INPUT_OUTPUT, /* class               */
72                       screen->root_visual,           /* visual              */
73                       mask, values);                 /* masks */
74
75    /* Map the window on the screen */
76    xcb_map_window (c, win);
77
78
79    /* We flush the request */
80    xcb_flush (c);
81
82    while ((e = xcb_wait_for_event (c))) {
83      switch (e->response_type & ~0x80) {
84      case XCB_EXPOSE: {
85        /* We draw the points */
86        xcb_poly_point (c, XCB_COORD_MODE_ORIGIN, win, foreground, 4, points);
87
88        /* We draw the polygonal line */
89        xcb_poly_line (c, XCB_COORD_MODE_PREVIOUS, win, foreground, 4, polyline);
90
91        /* We draw the segements */
92        xcb_poly_segment (c, win, foreground, 2, segments);
93
94        /* We draw the rectangles */
95        xcb_poly_rectangle (c, win, foreground, 2, rectangles);
96
97        /* We draw the arcs */
98        xcb_poly_arc (c, win, foreground, 2, arcs);
99
100       /* We flush the request */
101       xcb_flush (c);
102
103       break;
104     }
105     default: {
106       /* Unknown event type, ignore it */
107       break;
108     }
109     }
110     /* Free the Generic Event */
111     free (e);
112   }
113
114   return 0;
115 }
```

10. X Events

In an X program, everything is driven by events. Event painting on the screen is sometimes done as a response to an event (an Expose event). If part of a program's window that was hidden, gets exposed (e.g. the window was raised above other widows), the X server will send an "expose" event to let the program know it should repaint that part of the window. User input (key presses, mouse movement, etc) is also received as a set of events.

(a) Registering for event types using event masks

During the creation of a window, you should give it what kind of events it wishes to receive. Thus, you may register for various mouse (also called pointer) events, keyboard events, expose events, and so on. This is done for optimizing the server-to-client connection (i.e. why send a program (that might even be running at the other side of the globe) an event it is not interested in ?)

In XCB, you use the "value_mask" and "value_list" data in the xcb_create_window() function to register for events. Here is how we register for Expose event when creating a window:

```
1   mask = XCB_CW_EVENT_MASK;
2   valwin[0] = XCB_EVENT_MASK_EXPOSURE;
3   win = xcb_generate_id (c);
4   xcb_create_window (c, depth, win, root->root,
5                      0, 0, 150, 150, 10,
6                      XCB_WINDOW_CLASS_INPUT_OUTPUT, root->root_visual,
7                      mask, valwin);
```

XCB_EVENT_MASK_EXPOSURE is a constant defined in the xcb_event_mask_t enumeration in the "xproto.h" header file. If we wanted to register for several event types, we can logically "or" them, as follows:

```
1   mask = XCB_CW_EVENT_MASK;
2   valwin[0] = XCB_EVENT_MASK_EXPOSURE | XCB_EVENT_MASK_BUTTON_PRESS;
3   win = xcb_generate_id (c);
4   xcb_create_window (c, depth, win, root->root,
5                      0, 0, 150, 150, 10,
6                      XCB_WINDOW_CLASS_INPUT_OUTPUT, root->root_visual,
7                      mask, valwin);
```

This registers for Expose events as well as for mouse button presses inside the created window. You should note that a mask may represent several event sub-types.

The values that a mask could take are given by the xcb_cw_t enumeration:

```
1  typedef enum {
2      XCB_CW_BACK_PIXMAP       = 1L<<0,
3      XCB_CW_BACK_PIXEL        = 1L<<1,
4      XCB_CW_BORDER_PIXMAP     = 1L<<2,
5      XCB_CW_BORDER_PIXEL      = 1L<<3,
6      XCB_CW_BIT_GRAVITY       = 1L<<4,
7      XCB_CW_WIN_GRAVITY       = 1L<<5,
8      XCB_CW_BACKING_STORE     = 1L<<6,
9      XCB_CW_BACKING_PLANES    = 1L<<7,
10     XCB_CW_BACKING_PIXEL     = 1L<<8,
11     XCB_CW_OVERRIDE_REDIRECT = 1L<<9,
12     XCB_CW_SAVE_UNDER        = 1L<<10,
13     XCB_CW_EVENT_MASK        = 1L<<11,
14     XCB_CW_DONT_PROPAGATE    = 1L<<12,
```

```
15      XCB_CW_COLORMAP          = 1L<<13,
16      XCB_CW_CURSOR            = 1L<<14
17  } xcb_cw_t;
```

Note: we must be careful when setting the values of the valwin parameter, as they have to follow the order the xcb_cw_t enumeration. Here is an example:

```
1   mask = XCB_CW_EVENT_MASK | XCB_CW_BACK_PIXMAP;
2   valwin[0] = XCB_NONE;                                          /* for
    ↪ XCB_CW_BACK_PIXMAP (whose value is 1)     */
3   valwin[1] = XCB_EVENT_MASK_EXPOSURE | XCB_EVENT_MASK_BUTTON_PRESS; /* for
    ↪ XCB_CW_EVENT_MASK, whose value (2048)     */
4                                                                  /* is
    ↪ greater than the one of XCB_CW_BACK_PIXMAP */
```

If the window has already been created, we can use the xcb_change_window_attributes() function to set the events that the window will receive. The subsection Configuring a window shows its prototype. As an example, here is a piece of code that configures the window to receive the Expose and ButtonPress events:

```
1  const static uint32_t values[] = { XCB_EVENT_MASK_EXPOSURE |
   ↪ XCB_EVENT_MASK_BUTTON_PRESS };
2
3  /* The connection c and the window win are supposed to be defined */
4
5  xcb_change_window_attributes (c, win, XCB_CW_EVENT_MASK, values);
```

Note: A common bug programmers do is adding code to handle new event types in their program, while forgetting to add the masks for these events in the creation of the window. Such a programmer then should sit down for hours debugging his program, wondering "Why doesn't my program notice that I released the button?", only to find that they registered for button press events but not for button release events.

(b) Receiving events: writing the events loop

After we have registered for the event types we are interested in, we need to enter a loop of receiving events and handling them. There are two ways to receive events: a blocking way and a non-blocking way:

- xcb_wait_for_event (xcb_connection_t *c) is the blocking way. It waits (so blocks...) until an event is queued in the X server. Then it retrieves it into a newly allocated structure (it dequeues it from the queue) and returns it. This structure has to be freed. The function returns NULL if an error occurs.

- xcb_poll_for_event (xcb_connection_t *c, int *error) is the non-blocking way. It looks at the event queue and returns (and dequeues too) an existing event into a newly allocated structure. This structure has to be freed. It returns NULL if there is no event. If an error occurs, the parameter error will be filled with the error status.

There are various ways to write such a loop. We present two ways to write such a loop, with the two functions above. The first one uses xcb_wait_for_event_t, which is similar to an event Xlib loop using only XNextEvent:

```
1   xcb_generic_event_t *e;
2
3   while ((e = xcb_wait_for_event (c))) {
4     switch (e->response_type & ~0x80) {
5     case XCB_EXPOSE: {
```

```
      /* Handle the Expose event type */
      xcb_expose_event_t *ev = (xcb_expose_event_t *)e;

      /* ... */

      break;
    }
    case XCB_BUTTON_PRESS: {
      /* Handle the ButtonPress event type */
      xcb_button_press_event_t *ev = (xcb_button_press_event_t *)e;

      /* ... */

      break;
    }
    default: {
      /* Unknown event type, ignore it */
      break;
    }
    }
    /* Free the Generic Event */
    free (e);
  }
```

You will certainly want to use xcb_poll_for_event(xcb_connection_t *c, int *error) if, in Xlib, you use XPending or XCheckMaskEvent:

```
  while (XPending (display)) {
    XEvent ev;

    XNextEvent(d, &ev);

    /* Manage your event */
  }
```

Such a loop in XCB looks like:

```
  xcb_generic_event_t *ev;

  while ((ev = xcb_poll_for_event (conn, 0))) {
    /* Manage your event */
  }
```

The events are managed in the same way as with xcb_wait_for_event_t. Obviously, we will need to give the user some way of terminating the program. This is usually done by handling a special ”quit” event, as we will soon see.

Comparison Xlib/XCB

- XNextEvent ()
- xcb_wait_for_event ()
- XPending ()
- XCheckMaskEvent ()
- xcb_poll_for_event ()

(c) Expose events

The Expose event is one of the most basic (and most used) events an application may receive. It will be sent to us in one of several cases:

- A window that covered part of our window has moved away, exposing part (or all) of our window.
- Our window was raised above other windows.
- Our window mapped for the first time.
- Our window was de-iconified.

You should note the implicit assumption hidden here: the contents of our window is lost when it is being obscured (covered) by either windows. One may wonder why the X server does not save this contents. The answer is: to save memory. After all, the number of windows on a display at a given time may be very large, and storing the contents of all of them might require a lot of memory. Actually, there is a way to tell the X server to store the contents of a window in special cases, as we will see later.

When we get an Expose event, we should take the event's data from the members of the following structure:

```
1  typedef struct {
2      uint8_t      response_type; /* The type of the event, here it is XCB_EXPOSE
       ↪ */
3      uint8_t      pad0;
4      uint16_t     sequence;
5      xcb_window_t window;        /* The Id of the window that receives the event
       ↪ (in case */
6                                  /* our application registered for events on
       ↪ several windows */
7      uint16_t     x;             /* The x coordinate of the top-left part of the
       ↪ window that needs to be redrawn */
8      uint16_t     y;             /* The y coordinate of the top-left part of the
       ↪ window that needs to be redrawn */
9      uint16_t     width;         /* The width of the part of the window that
       ↪ needs to be redrawn */
10     uint16_t     height;        /* The height of the part of the window that
       ↪ needs to be redrawn */
11     uint16_t     count;
12 } xcb_expose_event_t;
```

(d) Getting user input

User input traditionally comes from two sources: the mouse and the keyboard. Various event types exist to notify us of user input (a key being presses on the keyboard, a key being released on the keyboard, the mouse moving over our window, the mouse entering (or leaving) our window, and so on.

i. Mouse button press and release events

The first event type we will deal with is a mouse button-press (or button-release) event in our window. In order to register to such an event type, we should add one (or more) of the following masks when we create our window:

- XCB_EVENT_MASK_BUTTON_PRESS: notify us of any button that was pressed in one of our windows.
- XCB_EVENT_MASK_BUTTON_RELEASE: notify us of any button that was released in one of our windows.

The structure to be checked for in our events loop is the same for these two events, and is the following:

```
typedef struct {
    uint8_t         response_type; /* The type of the event, here it is
    ↪ xcb_button_press_event_t or xcb_button_release_event_t */
    xcb_button_t    detail;
    uint16_t        sequence;
    xcb_timestamp_t time;          /* Time, in milliseconds the event took
    ↪ place in */
    xcb_window_t    root;
    xcb_window_t    event;
    xcb_window_t    child;
    int16_t         root_x;
    int16_t         root_y;
    int16_t         event_x;       /* The x coordinate where the mouse has
    ↪ been pressed in the window */
    int16_t         event_y;       /* The y coordinate where the mouse has
    ↪ been pressed in the window */
    uint16_t        state;         /* A mask of the buttons (or keys)
    ↪ during the event */
    uint8_t         same_screen;
} xcb_button_press_event_t;

typedef xcb_button_press_event_t xcb_button_release_event_t;
```

The time field may be used to calculate ”double-click” situations by an application (e.g. if the mouse button was clicked two times in a duration shorter than a given amount of time, assume this was a double click).

The state field is a mask of the buttons held down during the event. It is a bitwise OR of any of the following (from the xcb_button_mask_t and xcb_mod_mask_t enumerations):

- XCB_BUTTON_MASK_1
- XCB_BUTTON_MASK_2
- XCB_BUTTON_MASK_3
- XCB_BUTTON_MASK_4
- XCB_BUTTON_MASK_5
- XCB_MOD_MASK_SHIFT
- XCB_MOD_MASK_LOCK
- XCB_MOD_MASK_CONTROL
- XCB_MOD_MASK_1
- XCB_MOD_MASK_2
- XCB_MOD_MASK_3
- XCB_MOD_MASK_4
- XCB_MOD_MASK_5

Their names are self explanatory, where the first 5 refer to the mouse buttons that are being pressed, while the rest refer to various ”special keys” that are being pressed (Mod1 is usually the ’Alt’ key or the ’Meta’ key).

TODO: Problem: it seems that the state does not change when clicking with various buttons.

ii. Mouse movement events

Similar to mouse button press and release events, we also can be notified of various mouse movement events. These can be split into two families. One is of mouse pointer movement while no buttons are pressed, and the second is a mouse pointer motion while one (or more) of the buttons are pressed (this is sometimes called "a mouse drag operation", or just "dragging"). The following event masks may be added during the creation of our window:

- XCB_EVENT_MASK_POINTER_MOTION: events of the pointer moving in one of the windows controlled by our application, while no mouse button is held pressed.
- XCB_EVENT_MASK_BUTTON_MOTION: Events of the pointer moving while one or more of the mouse buttons is held pressed.
- XCB_EVENT_MASK_BUTTON_1_MOTION: same as XCB_EVENT_MASK_BUT but only when the 1st mouse button is held pressed.
- XCB_EVENT_MASK_BUTTON_2_MOTION, XCB_EVENT_MASK_BUTTON_3 XCB_EVENT_MASK_BUTTON_4_MOTION, XCB_EVENT_MASK_BUTTON_5 same as XCB_EVENT_MASK_BUTTON_1_MOTION, but respectively for 2nd, 3rd, 4th and 5th mouse button.

The structure to be checked for in our events loop is the same for these events, and is the following:

```
typedef struct {
    uint8_t         response_type; /* The type of the event */
    uint8_t         detail;
    uint16_t        sequence;
    xcb_timestamp_t time;          /* Time, in milliseconds the event took
    ↪ place in */
    xcb_window_t    root;
    xcb_window_t    event;
    xcb_window_t    child;
    int16_t         root_x;
    int16_t         root_y;
    int16_t         event_x;       /* The x coordinate of the mouse when
    ↪ the  event was generated */
    int16_t         event_y;       /* The y coordinate of the mouse when
    ↪ the  event was generated */
    uint16_t        state;         /* A mask of the buttons (or keys)
    ↪ during the event */
    uint8_t         same_screen;
} xcb_motion_notify_event_t;
```

iii. Mouse pointer enter and leave events

Another type of event that applications might be interested in, is a mouse pointer entering a window the program controls, or leaving such a window. Some programs use these events to show the user that the application is now in focus. In order to register for such an event type, we should add one (or more) of the following masks when we create our window:

- xcb_event_enter_window_t: notify us when the mouse pointer enters any of our controlled windows.
- xcb_event_leave_window_t: notify us when the mouse pointer leaves any of our controlled windows.

The structure to be checked for in our events loop is the same for these two events, and is the following:

```
typedef struct {
    uint8_t         response_type; /* The type of the event */
    uint8_t         detail;
    uint16_t        sequence;
    xcb_timestamp_t time;          /* Time, in milliseconds the event took
    ↪ place in */
    xcb_window_t    root;
    xcb_window_t    event;
    xcb_window_t    child;
    int16_t         root_x;
    int16_t         root_y;
    int16_t         event_x;       /* The x coordinate of the mouse when
    ↪ the  event was generated */
    int16_t         event_y;       /* The y coordinate of the mouse when
    ↪ the  event was generated */
    uint16_t        state;         /* A mask of the buttons (or keys)
    ↪ during the event */
    uint8_t         mode;          /* The number of mouse button that was
    ↪ clicked */
    uint8_t         same_screen_focus;
} xcb_enter_notify_event_t;

typedef xcb_enter_notify_event_t xcb_leave_notify_event_t;
```

iv. The keyboard focus

There may be many windows on a screen, but only a single keyboard attached to them. How does the X server then know which window should be sent a given keyboard input ? This is done using the keyboard focus. Only a single window on the screen may have the keyboard focus at a given time. There is a XCB function that allows a program to set the keyboard focus to a given window. The user can usually set the keyboard focus using the window manager (often by clicking on the title bar of the desired window). Once our window has the keyboard focus, every key press or key release will cause an event to be sent to our program (if it regsitered for these event types...).

v. Keyboard press and release events

If a window controlled by our program currently holds the keyboard focus, it can receive key press and key release events. So, we should add one (or more) of the following masks when we create our window:

- XCB_EVENT_MASK_KEY_PRESS: notify us when a key was pressed while any of our controlled windows had the keyboard focus.
- XCB_EVENT_MASK_KEY_RELEASE: notify us when a key was released while any of our controlled windows had the keyboard focus.

The structure to be checked for in our events loop is the same for these two events, and is the following:

```
typedef struct {
    uint8_t         response_type; /* The type of the event */
    xcb_keycode_t   detail;
    uint16_t        sequence;
    xcb_timestamp_t time;          /* Time, in milliseconds the event took
    ↪ place in */
    xcb_window_t    root;
```

```
    xcb_window_t    event;
    xcb_window_t    child;
    int16_t         root_x;
    int16_t         root_y;
    int16_t         event_x;
    int16_t         event_y;
    uint16_t        state;
    uint8_t         same_screen;
} xcb_key_press_event_t;

typedef xcb_key_press_event_t xcb_key_release_event_t;
```

The detail field refers to the physical key on the keyboard.

TODO: Talk about getting the ASCII code from the key code.

(e) X events: a complete example

As an example for handling events, we show a program that creates a window, enters an events loop and checks for all the events described above, and writes on the terminal the relevant characteristics of the event. With this code, it should be easy to add drawing operations, like those which have been described above.

```
#include <stdlib.h>
#include <stdio.h>

#include <xcb/xcb.h>

void
print_modifiers (uint32_t mask)
{
  const char **mod, *mods[] = {
    "Shift", "Lock", "Ctrl", "Alt",
    "Mod2", "Mod3", "Mod4", "Mod5",
    "Button1", "Button2", "Button3", "Button4", "Button5"
  };
  printf ("Modifier mask: ");
  for (mod = mods ; mask; mask >>= 1, mod++)
    if (mask & 1)
      printf(*mod);
  putchar ('\n');
}

int
main ()
{
  xcb_connection_t    *c;
  xcb_screen_t        *screen;
  xcb_window_t         win;
  xcb_generic_event_t *e;
  uint32_t             mask = 0;
  uint32_t             values[2];

  /* Open the connection to the X server */
  c = xcb_connect (NULL, NULL);

  /* Get the first screen */
```

```
35    screen = xcb_setup_roots_iterator (xcb_get_setup (c)).data;
36
37    /* Ask for our window's Id */
38    win = xcb_generate_id (c);
39
40    /* Create the window */
41    mask = XCB_CW_BACK_PIXEL | XCB_CW_EVENT_MASK;
42    values[0] = screen->white_pixel;
43    values[1] = XCB_EVENT_MASK_EXPOSURE       | XCB_EVENT_MASK_BUTTON_PRESS   |
44                XCB_EVENT_MASK_BUTTON_RELEASE | XCB_EVENT_MASK_POINTER_MOTION |
45                XCB_EVENT_MASK_ENTER_WINDOW   | XCB_EVENT_MASK_LEAVE_WINDOW   |
46                XCB_EVENT_MASK_KEY_PRESS      | XCB_EVENT_MASK_KEY_RELEASE;
47    xcb_create_window (c,                             /* Connection          */
48                       0,                             /* depth               */
49                       win,                           /* window Id           */
50                       screen->root,                  /* parent window       */
51                       0, 0,                          /* x, y                */
52                       150, 150,                      /* width, height       */
53                       10,                            /* border_width        */
54                       XCB_WINDOW_CLASS_INPUT_OUTPUT, /* class               */
55                       screen->root_visual,           /* visual              */
56                       mask, values);                 /* masks */
57
58    /* Map the window on the screen */
59    xcb_map_window (c, win);
60
61    xcb_flush (c);
62
63    while ((e = xcb_wait_for_event (c))) {
64      switch (e->response_type & ~0x80) {
65      case XCB_EXPOSE: {
66        xcb_expose_event_t *ev = (xcb_expose_event_t *)e;
67
68        printf ("Window %ld exposed. Region to be redrawn at location (%d,%d),
    ↪  with dimension (%d,%d)\n",
69                ev->window, ev->x, ev->y, ev->width, ev->height);
70        break;
71      }
72      case XCB_BUTTON_PRESS: {
73        xcb_button_press_event_t *ev = (xcb_button_press_event_t *)e;
74        print_modifiers(ev->state);
75
76        switch (ev->detail) {
77        case 4:
78          printf ("Wheel Button up in window %ld, at coordinates (%d,%d)\n",
79                  ev->event, ev->event_x, ev->event_y);
80          break;
81        case 5:
82          printf ("Wheel Button down in window %ld, at coordinates (%d,%d)\n",
83                  ev->event, ev->event_x, ev->event_y);
84          break;
85        default:
86          printf ("Button %d pressed in window %ld, at coordinates (%d,%d)\n",
87                  ev->detail, ev->event, ev->event_x, ev->event_y);
88        }
```

```
89        break;
90      }
91      case XCB_BUTTON_RELEASE: {
92        xcb_button_release_event_t *ev = (xcb_button_release_event_t *)e;
93        print_modifiers(ev->state);
94
95        printf ("Button %d released in window %ld, at coordinates (%d,%d)\n",
96                ev->detail, ev->event, ev->event_x, ev->event_y);
97        break;
98      }
99      case XCB_MOTION_NOTIFY: {
100       xcb_motion_notify_event_t *ev = (xcb_motion_notify_event_t *)e;
101
102       printf ("Mouse moved in window %ld, at coordinates (%d,%d)\n",
103               ev->event, ev->event_x, ev->event_y);
104       break;
105     }
106     case XCB_ENTER_NOTIFY: {
107       xcb_enter_notify_event_t *ev = (xcb_enter_notify_event_t *)e;
108
109       printf ("Mouse entered window %ld, at coordinates (%d,%d)\n",
110               ev->event, ev->event_x, ev->event_y);
111       break;
112     }
113     case XCB_LEAVE_NOTIFY: {
114       xcb_leave_notify_event_t *ev = (xcb_leave_notify_event_t *)e;
115
116       printf ("Mouse left window %ld, at coordinates (%d,%d)\n",
117               ev->event, ev->event_x, ev->event_y);
118       break;
119     }
120     case XCB_KEY_PRESS: {
121       xcb_key_press_event_t *ev = (xcb_key_press_event_t *)e;
122       print_modifiers(ev->state);
123
124       printf ("Key pressed in window %ld\n",
125               ev->event);
126       break;
127     }
128     case XCB_KEY_RELEASE: {
129       xcb_key_release_event_t *ev = (xcb_key_release_event_t *)e;
130       print_modifiers(ev->state);
131
132       printf ("Key released in window %ld\n",
133               ev->event);
134       break;
135     }
136     default:
137       /* Unknown event type, ignore it */
138       printf("Unknown event: %d\n", e->response_type);
139       break;
140     }
141     /* Free the Generic Event */
142     free (e);
143   }
```

```
144
145   return 0;
146 }
```

11. Handling text and fonts

Besides drawing graphics on a window, we often want to draw text. Text strings have two major properties: the characters to be drawn and the font with which they are drawn. In order to draw text, we need to first request the X server to load a font. We then assign a font to a Graphic Context, and finally, we draw the text in a window, using the Graphic Context.

(a) The Font structure

In order to support flexible fonts, a font type is defined. You know what ? It's an Id:

```
1 typedef uint32_t xcb_font_t;
```

It is used to contain information about a font, and is passed to several functions that handle fonts selection and text drawing. We ask the X server to attribute an Id to our font with the function:

```
1 xcb_font_t xcb_generate_id (xcb_connection_t *c);
```

(b) Opening a Font

To open a font, we use the following function:

```
1 xcb_void_cookie_t xcb_open_font (xcb_connection_t *c,
2                                  xcb_font_t        fid,
3                                  uint16_t          name_len,
4                                  const char       *name);
```

The fid parameter is the font Id defined by xcb_generate_id() (see above). The name parameter is the name of the font you want to open. Use the command xlsfonts in a terminal to know which are the fonts available on your computer. The parameter name_len is the length of the name of the font (given by strlen()).

(c) Assigning a Font to a Graphic Context

Once a font is opened, you have to create a Graphic Context that will contain the informations about the color of the foreground and the background used when you draw a text in a Drawable. Here is an exemple of a Graphic Context that will allow us to draw an opened font with a black foreground and a white background:

```
1   /*
2    * c is the connection
3    * screen is the screen where the window is displayed
4    * window is the window in which we will draw the text
5    * font is the opened font
6    */
7
8   uint32_t             value_list[3];
9   xcb_gcontext_t       gc;
10  uint32_t             mask;
11
12  gc = xcb_generate_id (c);
13  mask = XCB_GC_FOREGROUND | XCB_GC_BACKGROUND | XCB_GC_FONT;
14  value_list[0] = screen->black_pixel;
```

```
  value_list[1] = screen->white_pixel;
  value_list[2] = font;
  xcb_create_gc (c, gc, window, mask, value_list);

  /* The font is not needed anymore, so we close it */
  xcb_close_font (c, font);
```

(d) Drawing text in a drawable

To draw a text in a drawable, we use the following function:

```
xcb_void_cookie_t xcb_image_text_8 (xcb_connection_t *c,
                                    uint8_t           string_len,
                                    xcb_drawable_t    drawable,
                                    xcb_gcontext_t    gc,
                                    int16_t           x,
                                    int16_t           y,
                                    const char       *string);
```

The string parameter is the text to draw. The location of the drawing is given by the parameters x and y. The base line of the text is exactly the parameter y.

(e) Complete example

This example draw a text at 10 pixels (for the base line) of the bottom of a window. Pressing the Esc key exits the program.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <xcb/xcb.h>

#define WIDTH 300
#define HEIGHT 100



static xcb_gc_t gc_font_get (xcb_connection_t *c,
                             xcb_screen_t     *screen,
                             xcb_window_t      window,
                             const char       *font_name);

static void text_draw (xcb_connection_t *c,
                       xcb_screen_t     *screen,
                       xcb_window_t      window,
                       int16_t           x1,
                       int16_t           y1,
                       const char       *label);

static void
text_draw (xcb_connection_t *c,
           xcb_screen_t     *screen,
           xcb_window_t      window,
           int16_t           x1,
           int16_t           y1,
           const char       *label)
{
```

```
32    xcb_void_cookie_t    cookie_gc;
33    xcb_void_cookie_t    cookie_text;
34    xcb_generic_error_t *error;
35    xcb_gcontext_t       gc;
36    uint8_t              length;
37
38    length = strlen (label);
39
40    gc = gc_font_get(c, screen, window, "7x13");
41
42    cookie_text = xcb_image_text_8_checked (c, length, window, gc,
43                                            x1,
44                                            y1, label);
45    error = xcb_request_check (c, cookie_text);
46    if (error) {
47      fprintf (stderr, "ERROR: can't paste text : %d\n", error->error_code);
48      xcb_disconnect (c);
49      exit (-1);
50    }
51
52    cookie_gc = xcb_free_gc (c, gc);
53    error = xcb_request_check (c, cookie_gc);
54    if (error) {
55      fprintf (stderr, "ERROR: can't free gc : %d\n", error->error_code);
56      xcb_disconnect (c);
57      exit (-1);
58    }
59  }
60
61  static xcb_gc_t
62  gc_font_get (xcb_connection_t *c,
63               xcb_screen_t     *screen,
64               xcb_window_t      window,
65               const char       *font_name)
66  {
67    uint32_t             value_list[3];
68    xcb_void_cookie_t    cookie_font;
69    xcb_void_cookie_t    cookie_gc;
70    xcb_generic_error_t *error;
71    xcb_font_t           font;
72    xcb_gcontext_t       gc;
73    uint32_t             mask;
74
75    font = xcb_generate_id (c);
76    cookie_font = xcb_open_font_checked (c, font,
77                                         strlen (font_name),
78                                         font_name);
79
80    error = xcb_request_check (c, cookie_font);
81    if (error) {
82      fprintf (stderr, "ERROR: can't open font : %d\n", error->error_code);
83      xcb_disconnect (c);
84      return -1;
85    }
86
```

```
87    gc = xcb_generate_id (c);
88    mask = XCB_GC_FOREGROUND | XCB_GC_BACKGROUND | XCB_GC_FONT;
89    value_list[0] = screen->black_pixel;
90    value_list[1] = screen->white_pixel;
91    value_list[2] = font;
92    cookie_gc = xcb_create_gc_checked (c, gc, window, mask, value_list);
93    error = xcb_request_check (c, cookie_gc);
94    if (error) {
95      fprintf (stderr, "ERROR: can't create gc : %d\n", error->error_code);
96      xcb_disconnect (c);
97      exit (-1);
98    }
99
100   cookie_font = xcb_close_font_checked (c, font);
101   error = xcb_request_check (c, cookie_font);
102   if (error) {
103     fprintf (stderr, "ERROR: can't close font : %d\n", error->error_code);
104     xcb_disconnect (c);
105     exit (-1);
106   }
107
108   return gc;
109 }
110
111 int main ()
112 {
113   xcb_screen_iterator_t screen_iter;
114   xcb_connection_t     *c;
115   const xcb_setup_t    *setup;
116   xcb_screen_t         *screen;
117   xcb_generic_event_t  *e;
118   xcb_generic_error_t  *error;
119   xcb_void_cookie_t     cookie_window;
120   xcb_void_cookie_t     cookie_map;
121   xcb_window_t          window;
122   uint32_t              mask;
123   uint32_t              values[2];
124   int                   screen_number;
125
126   /* getting the connection */
127   c = xcb_connect (NULL, &screen_number);
128   if (!c) {
129     fprintf (stderr, "ERROR: can't connect to an X server\n");
130     return -1;
131   }
132
133   /* getting the current screen */
134   setup = xcb_get_setup (c);
135
136   screen = NULL;
137   screen_iter = xcb_setup_roots_iterator (setup);
138   for (; screen_iter.rem != 0; --screen_number, xcb_screen_next (&screen_iter))
139     if (screen_number == 0)
140       {
141         screen = screen_iter.data;
```

```
142           break;
143         }
144     if (!screen) {
145       fprintf (stderr, "ERROR: can't get the current screen\n");
146       xcb_disconnect (c);
147       return -1;
148     }
149
150     /* creating the window */
151     window = xcb_generate_id (c);
152     mask = XCB_CW_BACK_PIXEL | XCB_CW_EVENT_MASK;
153     values[0] = screen->white_pixel;
154     values[1] =
155       XCB_EVENT_MASK_KEY_RELEASE |
156       XCB_EVENT_MASK_BUTTON_PRESS |
157       XCB_EVENT_MASK_EXPOSURE |
158       XCB_EVENT_MASK_POINTER_MOTION;
159     cookie_window = xcb_create_window_checked (c,
160                                                screen->root_depth,
161                                                window, screen->root,
162                                                20, 200, WIDTH, HEIGHT,
163                                                0, XCB_WINDOW_CLASS_INPUT_OUTPUT,
164                                                screen->root_visual,
165                                                mask, values);
166     cookie_map = xcb_map_window_checked (c, window);
167
168     /* error managing */
169     error = xcb_request_check (c, cookie_window);
170     if (error) {
171       fprintf (stderr, "ERROR: can't create window : %d\n", error->error_code);
172       xcb_disconnect (c);
173       return -1;
174     }
175     error = xcb_request_check (c, cookie_map);
176     if (error) {
177       fprintf (stderr, "ERROR: can't map window : %d\n", error->error_code);
178       xcb_disconnect (c);
179       return -1;
180     }
181
182     xcb_flush(c);
183
184     while (1) {
185       e = xcb_poll_for_event(c);
186       if (e) {
187         switch (e->response_type & ~0x80) {
188         case XCB_EXPOSE: {
189           char *text;
190
191           text = "Press ESC key to exit...";
192           text_draw (c, screen, window, 10, HEIGHT - 10, text);
193           break;
194         }
195         case XCB_KEY_RELEASE: {
196           xcb_key_release_event_t *ev;
```

```
197
198         ev = (xcb_key_release_event_t *)e;
199
200         switch (ev->detail) {
201           /* ESC */
202         case 9:
203           free (e);
204           xcb_disconnect (c);
205           return 0;
206         }
207       }
208       }
209       free (e);
210     }
211   }
212
213   return 0;
214 }
```

12. Interacting with the window manager

After we have seen how to create windows and draw on them, we take one step back, and look at how our windows are interacting with their environment (the full screen and the other windows). First of all, our application needs to interact with the window manager. The window manager is responsible to decorating drawn windows (i.e. adding a frame, an iconify button, a system menu, a title bar, etc), as well as handling icons shown when windows are being iconified. It also handles ordering of windows on the screen, and other administrative tasks. We need to give it various hints as to how we want it to treat our application's windows.

(a) Window properties

Many of the parameters communicated to the window manager are passed using data called "properties". These properties are attached by the X server to different windows, and are stored in a format that makes it possible to read them from different machines that may use different architectures (remember that an X client program may run on a remote machine).

The property and its type (a string, an integer, etc) are Id. Their type are xcb_atom_t:

```
1 typedef uint32_t xcb_atom_t;
```

To change the property of a window, we use the following function:

```
1 xcb_void_cookie_t xcb_change_property (xcb_connection_t *c,        /* Connection
   ↪ to the X server */
2                                        uint8_t          mode,     /* Property
   ↪ mode */
3                                        xcb_window_t     window,   /* Window */
4                                        xcb_atom_t       property, /* Property
   ↪ to change */
5                                        xcb_atom_t       type,     /* Type of
   ↪ the property */
6                                        uint8_t          format,   /* Format of
   ↪ the property (8, 16, 32) */
7                                        uint32_t         data_len, /* Length of
   ↪ the data parameter */
8                                        const void      *data);    /* Data */
```

The mode parameter coud be one of the following values (defined in enumeration xcb_prop_mode_t in the xproto.h header file):

- XCB_PROP_MODE_REPLACE
- XCB_PROP_MODE_PREPEND
- XCB_PROP_MODE_APPEND

(b) Setting the window name and icon name

The first thing we want to do would be to set the name for our window. This is done using the xcb_change_property() function. This name may be used by the window manager as the title of the window (in the title bar), in a task list, etc. The property atom to use to set the name of a window is WM_NAME (and WM_ICON_NAME for the iconified window) and its type is STRING. Here is an example of utilization:

```
1  #include <string.h>
2
3  #include <xcb/xcb.h>
4  #include <xcb/xcb_atom.h>
5
6  int
7  main ()
8  {
9    xcb_connection_t *c;
10   xcb_screen_t     *screen;
11   xcb_window_t      win;
12   char             *title = "Hello World !";
13   char             *title_icon = "Hello World ! (iconified)";
14
15
16
17   /* Open the connection to the X server */
18   c = xcb_connect (NULL, NULL);
19
20   /* Get the first screen */
21   screen = xcb_setup_roots_iterator (xcb_get_setup (c)).data;
22
23   /* Ask for our window's Id */
24   win = xcb_generate_id (c);
25
26   /* Create the window */
27   xcb_create_window (c,                             /* Connection          */
28                      0,                             /* depth               */
29                      win,                           /* window Id           */
30                      screen->root,                  /* parent window       */
31                      0, 0,                          /* x, y                */
32                      250, 150,                      /* width, height       */
33                      10,                            /* border_width        */
34                      XCB_WINDOW_CLASS_INPUT_OUTPUT, /* class               */
35                      screen->root_visual,           /* visual              */
36                      0, NULL);                      /* masks, not used     */
37
38   /* Set the title of the window */
39   xcb_change_property (c, XCB_PROP_MODE_REPLACE, win,
```

```
                        WM_NAME, STRING, 8,
                        strlen (title), title);

  /* Set the title of the window icon */
  xcb_change_property (c, XCB_PROP_MODE_REPLACE, win,
                        WM_ICON_NAME, STRING, 8,
                        strlen(title_icon), title_icon);

  /* Map the window on the screen */
  xcb_map_window (c, win);

  xcb_flush (c);

  while (1) {}

  return 0;
}
```

Note: the use of the atoms needs our program to be compiled and linked against xcb_atom, so that we have to use

```
gcc prog.c -o prog `pkg-config --cflags --libs xcb_atom`
```

for the program to compile fine.

13. Simple window operations

One more thing we can do to our window is manipulate them on the screen (resize them, move them, raise or lower them, iconify them, and so on). Some window operations functions are supplied by XCB for this purpose.

(a) Mapping and un-mapping a window

The first pair of operations we can apply on a window is mapping it, or un-mapping it. Mapping a window causes the window to appear on the screen, as we have seen in our simple window program example. Un-mapping it causes it to be removed from the screen (although the window as a logical entity still exists). This gives the effect of making a window hidden (unmapped) and shown again (mapped). For example, if we have a dialog box window in our program, instead of creating it every time the user asks to open it, we can create the window once, in an un-mapped mode, and when the user asks to open it, we simply map the window on the screen. When the user clicked the 'OK' or 'Cancel' button, we simply un-map the window. This is much faster than creating and destroying the window, however, the cost is wasted resources, both on the client side, and on the X server side.

To map a window, you use the following function:

```
xcb_void_cookie_t xcb_map_window (xcb_connection_t *c,
                                  xcb_window_t      window);
```

To have a simple example, see the example above. The mapping operation will cause an Expose event to be sent to our application, unless the window is completely covered by other windows.

Un-mapping a window is also simple. You use the function

```
xcb_void_cookie_t xcb_unmap_window (xcb_connection_t *c,
                                    xcb_window_t      window);
```

The utilization of this function is the same as xcb_map_window().

(b) Configuring a window

As we have seen when we have created our first window, in the X Events subsection, we can set some attributes for the window (that is, the position, the size, the events the window will receive, etc). If we want to modify them, but the window is already created, we can change them by using the following function:

```
xcb_void_cookie_t xcb_configure_window (xcb_connection_t *c,             /* The
 ↪  connection to the X server*/
                                        xcb_window_t      window,        /* The
 ↪  window to configure */
                                        uint16_t          value_mask,    /* The
 ↪  mask */
                                        const uint32_t   *value_list);   /* The
 ↪  values to set */
```

We set the value_mask to one or several mask values that are in the xcb_config_window_t enumeration in the xproto.h header:

- XCB_CONFIG_WINDOW_X: new x coordinate of the window's top left corner
- XCB_CONFIG_WINDOW_Y: new y coordinate of the window's top left corner
- XCB_CONFIG_WINDOW_WIDTH: new width of the window
- XCB_CONFIG_WINDOW_HEIGHT: new height of the window
- XCB_CONFIG_WINDOW_BORDER_WIDTH: new width of the border of the window
- XCB_CONFIG_WINDOW_SIBLING
- XCB_CONFIG_WINDOW_STACK_MODE: the new stacking order

We then give to value_mask the new value. We now describe how to use xcb_configure_window_t in some useful situations.

(c) Moving a window around the screen

An operation we might want to do with windows is to move them to a different location. This can be done like this:

```
const static uint32_t values[] = { 10, 20 };

/* The connection c and the window win are supposed to be defined */

/* Move the window to coordinates x = 10 and y = 20 */
xcb_configure_window (c, win, XCB_CONFIG_WINDOW_X | XCB_CONFIG_WINDOW_Y,
 ↪  values);
```

Note that when the window is moved, it might get partially exposed or partially hidden by other windows, and thus we might get Expose events due to this operation.

(d) Resizing a window

Yet another operation we can do is to change the size of a window. This is done using the following code:

```
const static uint32_t values[] = { 200, 300 };

/* The connection c and the window win are supposed to be defined */

/* Resize the window to width = 10 and height = 20 */
xcb_configure_window (c, win, XCB_CONFIG_WINDOW_WIDTH |
 ↪  XCB_CONFIG_WINDOW_HEIGHT, values);
```

We can also combine the move and resize operations using one single call to xcb_configure_window_t:

```
const static uint32_t values[] = { 10, 20, 200, 300 };

/* The connection c and the window win are supposed to be defined */

/* Move the window to coordinates x = 10 and y = 20 */
/* and resize the window to width = 10 and height = 20 */
xcb_configure_window (c, win, XCB_CONFIG_WINDOW_X | XCB_CONFIG_WINDOW_Y |
 ↪  XCB_CONFIG_WINDOW_WIDTH | XCB_CONFIG_WINDOW_HEIGHT, values);
```

(e) Changing windows stacking order: raise and lower

Until now, we changed properties of a single window. We'll see that there are properties that relate to the window and other windows. One of them is the stacking order. That is, the order in which the windows are layered on top of each other. The front-most window is said to be on the top of the stack, while the back-most window is at the bottom of the stack. Here is how to manipulate our windows stack order:

```
const static uint32_t values[] = { XCB_STACK_MODE_ABOVE };

/* The connection c and the window win are supposed to be defined */

/* Move the window on the top of the stack */
xcb_configure_window (c, win, XCB_CONFIG_WINDOW_STACK_MODE, values);
```

```
const static uint32_t values[] = { XCB_STACK_MODE_BELOW };

/* The connection c and the window win are supposed to be defined */

/* Move the window on the bottom of the stack */
xcb_configure_window (c, win, XCB_CONFIG_WINDOW_STACK_MODE, values);
```

(f) Getting information about a window

Just like we can set various attributes of our windows, we can also ask the X server supply the current values of these attributes. For example, we can check where a window is located on the screen, what is its current size, whether it is mapped or not, etc. The structure that contains some of this information is

```
typedef struct {
    uint8_t      response_type;
    uint8_t      depth;         /* depth of the window */
    uint16_t     sequence;
    uint32_t     length;
    xcb_window_t root;          /* Id of the root window *>
    int16_t      x;             /* X coordinate of the window's location */
    int16_t      y;             /* Y coordinate of the window's location */
    uint16_t     width;         /* Width of the window */
    uint16_t     height;        /* Height of the window */
    uint16_t     border_width;  /* Width of the window's border */
} xcb_get_geometry_reply_t;
```

XCB fill this structure with two functions:

```
xcb_get_geometry_cookie_t  xcb_get_geometry       (xcb_connection_t         *c,
                                                   xcb_drawable_t
 ↪  drawable);
```

```
xcb_get_geometry_reply_t  *xcb_get_geometry_reply (xcb_connection_t         *c,
                                                   xcb_get_geometry_cookie_t
 ↪  cookie,
                                                   xcb_generic_error_t
 ↪  **e);
```

You use them as follows:

```
  xcb_connection_t         *c;
  xcb_drawable_t            win;
  xcb_get_geometry_reply_t *geom;

  /* You initialize c and win */

  geom = xcb_get_geometry_reply (c, xcb_get_geometry (c, win), NULL);

  /* Do something with the fields of geom */

  free (geom);
```

Remark that you have to free the structure, as xcb_get_geometry_reply_t allocates a newly one.

One problem is that the returned location of the window is relative to its parent window. This makes these coordinates rather useless for any window manipulation functions, like moving it on the screen. In order to overcome this problem, we need to take a two-step operation. First, we find out the Id of the parent window of our window. We then translate the above relative coordinates to the screen coordinates.

To get the Id of the parent window, we need this structure:

```
typedef struct {
    uint8_t      response_type;
    uint8_t      pad0;
    uint16_t     sequence;
    uint32_t     length;
    xcb_window_t root;
    xcb_window_t parent;       /* Id of the parent window */
    uint16_t     children_len;
    uint8_t      pad1[14];
} xcb_query_tree_reply_t;
```

To fill this structure, we use these two functions:

```
xcb_query_tree_cookie_t xcb_query_tree       (xcb_connection_t        *c,
                                              xcb_window_t             window);
xcb_query_tree_reply_t *xcb_query_tree_reply (xcb_connection_t        *c,
                                              xcb_query_tree_cookie_t  cookie,
                                              xcb_generic_error_t    **e);
```

The translated coordinates will be found in this structure:

```
typedef struct {
    uint8_t      response_type;
    uint8_t      same_screen;
    uint16_t     sequence;
    uint32_t     length;
    xcb_window_t child;
    uint16_t     dst_x;        /* Translated x coordinate */
```

```
    uint16_t     dst_y;        /* Translated y coordinate */
} xcb_translate_coordinates_reply_t;
```

As usual, we need two functions to fill this structure:

```
xcb_translate_coordinates_cookie_t xcb_translate_coordinates
 ↪ (xcb_connection_t                  *c,

 ↪ xcb_window_t                       src_window,

 ↪ xcb_window_t                       dst_window,
                                                                int16_t
 ↪ src_x,
                                                                int16_t
 ↪ src_y);
xcb_translate_coordinates_reply_t *xcb_translate_coordinates_reply
 ↪ (xcb_connection_t                  *c,

 ↪ xcb_translate_coordinates_cookie_t cookie,

 ↪ xcb_generic_error_t               **e);
```

We use them as follows:

```
  xcb_connection_t                 *c;
  xcb_drawable_t                    win;
  xcb_get_geometry_reply_t         *geom;
  xcb_query_tree_reply_t           *tree;
  xcb_translate_coordinates_reply_t *trans;

  /* You initialize c and win */

  geom  = xcb_get_geometry_reply (c, xcb_get_geometry (c, win), NULL);
  if (!geom)
    return 0;

  tree  = xcb_query_tree_reply (c, xcb_query_tree (c, win), NULL);
  if (!tree)
    return 0;

  trans = xcb_translate_coordinates_reply (c,
                                           xcb_translate_coordinates (c,
                                                                      win,

 ↪ tree->parent,
                                                                      geom->x,
 ↪ geom->y),
                                           NULL);
  if (!trans)
    return 0;

  /* the translated coordinates are in trans->dst_x and trans->dst_y */

  free (trans);
  free (tree);
  free (geom);
```

Of course, as for geom, tree and trans have to be freed.

The work is a bit hard, but XCB is a very low-level library.

TODO: the utilization of these functions should be a prog, which displays the coordinates of the window.

There is another structure that gives informations about our window:

```
typedef struct {
    uint8_t        response_type;
    uint8_t        backing_store;
    uint16_t       sequence;
    uint32_t       length;
    xcb_visualid_t visual;               /* Visual of the window */
    uint16_t       _class;
    uint8_t        bit_gravity;
    uint8_t        win_gravity;
    uint32_t       backing_planes;
    uint32_t       backing_pixel;
    uint8_t        save_under;
    uint8_t        map_is_installed;
    uint8_t        map_state;            /* Map state of the window */
    uint8_t        override_redirect;
    xcb_colormap_t colormap;             /* Colormap of the window */
    uint32_t       all_event_masks;
    uint32_t       your_event_mask;
    uint16_t       do_not_propagate_mask;
} xcb_get_window_attributes_reply_t;
```

XCB supplies these two functions to fill it:

```
xcb_get_window_attributes_cookie_t xcb_get_window_attributes
 ↪  (xcb_connection_t                  *c,

 ↪  xcb_window_t                       window);
xcb_get_window_attributes_reply_t *xcb_get_window_attributes_reply
 ↪  (xcb_connection_t                  *c,

 ↪  xcb_get_window_attributes_cookie_t cookie,

 ↪  xcb_generic_error_t              **e);
```

You use them as follows:

```
  xcb_connection_t                 *c;
  xcb_drawable_t                    win;
  xcb_get_window_attributes_reply_t *attr;

  /* You initialize c and win */

  attr = xcb_get_window_attributes_reply (c, xcb_get_window_attributes (c,
 ↪  win), NULL);

  if (!attr)
    return 0;

  /* Do something with the fields of attr */

  free (attr);
```

As for geom, attr has to be freed.

14. Using colors to paint the rainbow

Up until now, all our painting operation were done using black and white. We will (finally) see now how to draw using colors.

(a) Color maps

In the beginning, there were not enough colors. Screen controllers could only support a limited number of colors simultaneously (initially 2, then 4, 16 and 256). Because of this, an application could not just ask to draw in a "light purple-red" color, and expect that color to be available. Each application allocated the colors it needed, and when all the color entries (4, 16, 256 colors) were in use, the next color allocation would fail.

Thus, the notion of "a color map" was introduced. A color map is a table whose size is the same as the number of simultaneous colors a given screen controller. Each entry contained the RGB (Red, Green and Blue) values of a different color (all colors can be drawn using some combination of red, green and blue). When an application wants to draw on the screen, it does not specify which color to use. Rather, it specifies which color entry of some color map to be used during this drawing. Change the value in this color map entry and the drawing will use a different color.

In order to be able to draw using colors that got something to do with what the programmer intended, color map allocation functions are supplied. You could ask to allocate entry for a color with a set of RGB values. If one already existed, you would get its index in the table. If none existed, and the table was not full, a new cell would be allocated to contain the given RGB values, and its index returned. If the table was full, the procedure would fail. You could then ask to get a color map entry with a color that is closest to the one you were asking for. This would mean that the actual drawing on the screen would be done using colors similar to what you wanted, but not the same.

On today's more modern screens where one runs an X server with support for 16 million colors, this limitation looks a little silly, but remember that there are still older computers with older graphics cards out there. Using color map, support for these screen becomes transparent to you. On a display supporting 16 million colors, any color entry allocation request would succeed. On a display supporting a limited number of colors, some color allocation requests would return similar colors. It won't look as good, but your application would still work.

(b) Allocating and freeing Color Maps

When you draw using XCB, you can choose to use the standard color map of the screen your window is displayed on, or you can allocate a new color map and apply it to a window. In the latter case, each time the mouse moves onto your window, the screen color map will be replaced by your window's color map, and you'll see all the other windows on screen change their colors into something quite bizzare. In fact, this is the effect you get with X applications that use the "-install" command line option.

In XCB, a color map is (as often in X) an Id:

```
1 typedef uint32_t xcb_colormap_t;
```

In order to access the screen's default color map, you just have to retrieve the default_colormap field of the xcb_screen_t structure (see Section Checking basic information about a connection):

```
1 #include <stdio.h>
2
```

```c
#include <xcb/xcb.h>

int
main ()
{
  xcb_connection_t *c;
  xcb_screen_t     *screen;
  xcb_colormap_t    colormap;

  /* Open the connection to the X server and get the first screen */
  c = xcb_connect (NULL, NULL);
  screen = xcb_setup_roots_iterator (xcb_get_setup (c)).data;

  colormap = screen->default_colormap;

  return 0;
}
```

This will return the color map used by default on the first screen (again, remember that an X server may support several different screens, each of which might have its own resources).

The other option, that of allocating a new colormap, works as follows. We first ask the X server to give an Id to our color map, with this function:

```c
xcb_colormap_t xcb_generate_id (xcb_connection_t *c);
```

Then, we create the color map with

```c
xcb_void_cookie_t xcb_create_colormap (xcb_connection_t *c,       /* Pointer to the xcb_connection_t structure */
                                       uint8_t           alloc,   /* Colormap entries to be allocated (AllocNone or AllocAll) */
                                       xcb_colormap_t    mid,     /* Id of the color map */
                                       xcb_window_t      window,  /* Window on whose screen the colormap will be created */
                                       xcb_visualid_t    visual); /* Id of the visual supported by the screen */
```

Here is an example of creation of a new color map:

```c
#include <xcb/xcb.h>

int
main ()
{
  xcb_connection_t *c;
  xcb_screen_t     *screen;
  xcb_window_t      win;
  xcb_colormap_t    cmap

  /* Open the connection to the X server and get the first screen */
  c = xcb_connect (NULL, NULL);
  screen = xcb_setup_roots_iterator (xcb_get_setup (c)).data;

  /* We create the window win here*/

```

```
17    cmap = xcb_generate_id (c);
18    xcb_create_colormap (c, XCB_COLORMAP_ALLOC_NONE, cmap, win,
      ↪  screen->root_visual);
19
20    return 0;
21  }
```

Note that the window parameter is only used to allow the X server to create the color map for the given screen. We can then use this color map for any window drawn on the same screen.

To free a color map, it suffices to use this function:

```
1  xcb_void_cookie_t xcb_free_colormap (xcb_connection_t *c,   /* The connection
   ↪  */
2                                       xcb_colormap_t cmap);  /* The color map */
```

Comparison Xlib/XCB

- XCreateColormap ()
- xcb_generate_id ()
- xcb_create_colormap ()
- XFreeColormap ()
- xcb_free_colormap ()

(c) Allocating and freeing a color entry

Once we got access to some color map, we can start allocating colors. The informations related to a color are stored in the following structure:

```
1  typedef struct {
2      uint8_t  response_type;
3      uint8_t  pad0;
4      uint16_t sequence;
5      uint32_t length;
6      uint16_t red;          /* The red component   */
7      uint16_t green;        /* The green component */
8      uint16_t blue;         /* The blue component  */
9      uint8_t  pad1[2];
10     uint32_t pixel;        /* The entry in the color map, supplied by the X
       ↪  server */
11 } xcb_alloc_color_reply_t;
```

XCB supplies these two functions to fill it:

```
1  xcb_alloc_color_cookie_t xcb_alloc_color       (xcb_connection_t        *c,
2                                                  xcb_colormap_t           cmap,
3                                                  uint16_t                 red,
4                                                  uint16_t                 green,
5                                                  uint16_t                 blue);
6  xcb_alloc_color_reply_t *xcb_alloc_color_reply (xcb_connection_t        *c,
7                                                  xcb_alloc_color_cookie_t
   ↪  cookie,
8                                                  xcb_generic_error_t    **e);
```

The fuction xcb_alloc_color() takes the 3 RGB components as parameters (red, green and blue). Here is an example of using these functions:

```
#include <malloc.h>

#include <xcb/xcb.h>

int
main ()
{
  xcb_connection_t        *c;
  xcb_screen_t            *screen;
  xcb_window_t             win;
  xcb_colormap_t           cmap;
  xcb_alloc_color_reply_t *rep;

  /* Open the connection to the X server and get the first screen */
  c = xcb_connect (NULL, NULL);
  screen = xcb_setup_roots_iterator (xcb_get_setup (c)).data;

  /* We create the window win here*/

  cmap = xcb_generate_id (c);
  xcb_create_colormap (c, XCB_COLORMAP_ALLOC_NONE, cmap, win,
   ↪  screen->root_visual);

  rep = xcb_alloc_color_reply (c, xcb_alloc_color (c, cmap, 65535, 0, 0),
   ↪  NULL);

  if (!rep)
    return 0;

  /* Do something with r->pixel or the components */

  free (rep);

  return 0;
}
```

As xcb_alloc_color_reply() allocates memory, you have to free rep.

TODO: Talk about freeing colors.

15. X Bitmaps and Pixmaps

One thing many so-called ”Multi-Media” applications need to do, is display images. In the X world, this is done using bitmaps and pixmaps. We have already seen some usage of them when setting an icon for our application. Lets study them further, and see how to draw these images inside a window, along side the simple graphics and text we have seen so far.

One thing to note before delving further, is that XCB (nor Xlib) supplies no means of manipulating popular image formats, such as gif, png, jpeg or tiff. It is up to the programmer (or to higher level graphics libraries) to translate these image formats into formats that the X server is familiar with (x bitmaps and x pixmaps).

(a) What is a X Bitmap? An X Pixmap?

An X bitmap is a two-color image stored in a format specific to the X window system.

When stored in a file, the bitmap data looks like a C source file. It contains variables defining the width and the height of the bitmap, an array containing the bit values of the bitmap (the size of the array is (width+7)/8*height and the bit and byte order are LSB), and an optional hot-spot location (that will be explained later, when discussing mouse cursors).

An X pixmap is a format used to stored images in the memory of an X server. This format can store both black and white images (such as x bitmaps) as well as color images. It is the only image format supported by the X protocol, and any image to be drawn on screen, should be first translated into this format.

In actuality, an X pixmap can be thought of as a window that does not appear on the screen. Many graphics operations that work on windows, will also work on pixmaps. Indeed, the type of X pixmap in XCB is an Id like a window:

```
typedef uint32_t xcb_pixmap_t;
```

Like Xlib, there is no difference between a Drawable, a Window or a Pixmap:

```
typedef uint32_t xcb_drawable_t;
```

in order to avoid confusion between a window and a pixmap. The operations that will work the same on a window or a pixmap will require a xcb_drawable_t

Remark: In Xlib, there is no specific difference between a Drawable, a Pixmap or a Window: all are 32 bit long integer. XCB wraps all these different IDs in structures to provide some measure of type-safety.

(b) Creating a pixmap

Sometimes we want to create an un-initialized pixmap, so we can later draw into it. This is useful for image drawing programs (creating a new empty canvas will cause the creation of a new pixmap on which the drawing can be stored). It is also useful when reading various image formats: we load the image data into memory, create a pixmap on the server, and then draw the decoded image data onto that pixmap.

To create a new pixmap, we first ask the X server to give an Id to our pixmap, with this function:

```
xcb_pixmap_t xcb_generate_id (xcb_connection_t *c);
```

Then, XCB supplies the following function to create new pixmaps:

```
xcb_void_cookie_t xcb_create_pixmap (xcb_connection_t *c,          /* Pointer to
    the xcb_connection_t structure */
                                     uint8_t           depth,      /* Depth of
    the screen */
                                     xcb_pixmap_t      pid,        /* Id of the
    pixmap */
                                     xcb_drawable_t    drawable,
                                     uint16_t          width,      /* Width of
    the window (in pixels) */
                                     uint16_t          height);    /* Height of
    the window (in pixels) */
```

TODO: Explain the drawable parameter, and give an example (like xpoints.c)

(c) Drawing a pixmap in a window

Once we got a handle to a pixmap, we can draw it on some window, using the following function:

```
xcb_void_cookie_t xcb_copy_area (xcb_connection_t *c,             /* Pointer to
    ↪ the xcb_connection_t structure */
                                 xcb_drawable_t    src_drawable,  /* The
    ↪ Drawable we want to paste */
                                 xcb_drawable_t    dst_drawable,  /* The
    ↪ Drawable on which we copy the previous Drawable */
                                 xcb_gcontext_t    gc,            /* A Graphic
    ↪ Context */
                                 int16_t           src_x,         /* Top left x
    ↪ coordinate of the region we want to copy */
                                 int16_t           src_y,         /* Top left y
    ↪ coordinate of the region we want to copy */
                                 int16_t           dst_x,         /* Top left x
    ↪ coordinate of the region where we want to copy */
                                 int16_t           dst_y,         /* Top left y
    ↪ coordinate of the region where we want to copy */
                                 uint16_t          width,         /* Width of
    ↪ the region we want to copy */
                                 uint16_t          height);       /* Height of
    ↪ the region we want to copy */
```

As you can see, we could copy the whole pixmap, as well as only a given rectangle of the pixmap. This is useful to optimize the drawing speed: we could copy only what we have modified in the pixmap.

One important note should be made: it is possible to create pixmaps with different depths on the same screen. When we perform copy operations (a pixmap onto a window, etc), we should make sure that both source and target have the same depth. If they have a different depth, the operation would fail. The exception to this is if we copy a specific bit plane of the source pixmap using the xcb_copy_plane_t function. In such an event, we can copy a specific plane to the target window (in actuality, setting a specific bit in the color of each pixel copied). This can be used to generate strange graphic effects in a window, but that is beyond the scope of this tutorial.

(d) Freeing a pixmap

Finally, when we are done using a given pixmap, we should free it, in order to free resources of the X server. This is done using this function:

```
xcb_void_cookie_t xcb_free_pixmap (xcb_connection_t *c,       /* Pointer to
    ↪ the xcb_connection_t structure */
                                   xcb_pixmap_t      pixmap); /* A given
    ↪ pixmap */
```

Of course, after having freed it, we must not try accessing the pixmap again.

TODO: Give an example, or a link to xpoints.c

16. Messing with the mouse cursor

It it possible to modify the shape of the mouse pointer (also called the X pointer) when in certain states, as we otfen see in programs. For example, a busy application would often display the sand clock over its main window, to give the user a visual hint that he should wait. Let's see how we can change the mouse cursor of our windows.

(a) Creating and destroying a mouse cursor

There are two methods for creating cursors. One of them is by using a set of predefined cursors, that are supplied by the X server, the other is by using a user-supplied bitmap.

In the first method, we use a special font named ”cursor”, and the function xcb_create_glyph_cursor:

```
xcb_void_cookie_t xcb_create_glyph_cursor (xcb_connection_t *c,
                                           xcb_cursor_t      cid,
                                           xcb_font_t        source_font, /*
    ↪ font for the source glyph */
                                           xcb_font_t        mask_font,   /*
    ↪ font for the mask glyph or XCB_NONE */
                                           uint16_t          source_char, /*
    ↪ character glyph for the source */
                                           uint16_t          mask_char,   /*
    ↪ character glyph for the mask */
                                           uint16_t          fore_red,    /*
    ↪ red value for the foreground of the source */
                                           uint16_t          fore_green,  /*
    ↪ green value for the foreground of the source */
                                           uint16_t          fore_blue,   /*
    ↪ blue value for the foreground of the source */
                                           uint16_t          back_red,    /*
    ↪ red value for the background of the source */
                                           uint16_t          back_green,  /*
    ↪ green value for the background of the source */
                                           uint16_t          back_blue)   /*
    ↪ blue value for the background of the source */
```

TODO: Describe source_char and mask_char, for example by giving an example on how to get the values. There is a list there: X Font Cursors

So we first open that font (see Loading a Font) and create the new cursor. As for every X ressource, we have to ask for an X id with xcb_generate_id first:

```
xcb_font_t           font;
xcb_cursor_t         cursor;

/* The connection is set */

font = xcb_generate_id (conn);
xcb_open_font (conn, font, strlen ("cursor"), "cursor");

cursor = xcb_generate_id (conn);
xcb_create_glyph_cursor (conn, cursor, font, font,
                         58, 58 + 1,
                         0, 0, 0,
                         0, 0, 0);
```

We have created the cursor ”right hand” by specifying 58 to the source_font argument and 58 + 1 to the mask_font.

The cursor is destroyed by using the function

```
xcb_void_cookie_t xcb_free_cursor (xcb_connection_t *c,
                                   xcb_cursor_t      cursor);
```

In the second method, we create a new cursor by using a pair of pixmaps, with depth of one (that is, two colors pixmaps). One pixmap defines the shape of the cursor, while the other works as a mask, specifying which pixels of the cursor will be actually drawn. The rest of the pixels will be transparent.

TODO: give an example.

(b) Setting a window's mouse cursor

Once the cursor is created, we can modify the cursor of our window by using xcb_change_window_attributes and using the XCB_CWCURSOR attribute:

```
uint32_t mask;
uint32_t value_list;

/* The connection and window are set */
/* The cursor is already created */

mask = XCB_CWCURSOR;
value_list = cursor;
xcb_change_window_attributes (conn, window, mask, &value_list);
```

Of course, the cursor and the font must be freed.

(c) Complete example

The following example displays a window with a button. When entering the window, the window cursor is changed to an arrow. When clicking once on the button, the cursor is changed to a hand. When clicking again on the button, the cursor window gets back to the arrow. The Esc key exits the application.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <xcb/xcb.h>

#define WIDTH 300
#define HEIGHT 150



static xcb_gc_t gc_font_get (xcb_connection_t *c,
                             xcb_screen_t     *screen,
                             xcb_window_t      window,
                             const char       *font_name);

static void button_draw (xcb_connection_t *c,
                         xcb_screen_t     *screen,
                         xcb_window_t      window,
                         int16_t           x1,
                         int16_t           y1,
                         const char       *label);

static void text_draw (xcb_connection_t *c,
                       xcb_screen_t     *screen,
                       xcb_window_t      window,
                       int16_t           x1,
                       int16_t           y1,
                       const char       *label);

static void cursor_set (xcb_connection_t *c,
                        xcb_screen_t     *screen,
```

```
33                          xcb_window_t      window,
34                          int               cursor_id);
35
36
37  static void
38  button_draw (xcb_connection_t *c,
39               xcb_screen_t     *screen,
40               xcb_window_t      window,
41               int16_t           x1,
42               int16_t           y1,
43               const char       *label)
44  {
45    xcb_point_t          points[5];
46    xcb_void_cookie_t    cookie_gc;
47    xcb_void_cookie_t    cookie_line;
48    xcb_void_cookie_t    cookie_text;
49    xcb_generic_error_t *error;
50    xcb_gcontext_t       gc;
51    int16_t              width;
52    int16_t              height;
53    uint8_t              length;
54    int16_t              inset;
55
56    length = strlen (label);
57    inset = 2;
58
59    gc = gc_font_get(c, screen, window, "7x13");
60
61    width = 7 * length + 2 * (inset + 1);
62    height = 13 + 2 * (inset + 1);
63    points[0].x = x1;
64    points[0].y = y1;
65    points[1].x = x1 + width;
66    points[1].y = y1;
67    points[2].x = x1 + width;
68    points[2].y = y1 - height;
69    points[3].x = x1;
70    points[3].y = y1 - height;
71    points[4].x = x1;
72    points[4].y = y1;
73    cookie_line = xcb_poly_line_checked (c, XCB_COORD_MODE_ORIGIN,
74                                         window, gc, 5, points);
75
76    error = xcb_request_check (c, cookie_line);
77    if (error) {
78      fprintf (stderr, "ERROR: can't draw lines : %d\n", error->error_code);
79      xcb_disconnect (c);
80      exit (-1);
81    }
82
83    cookie_text = xcb_image_text_8_checked (c, length, window, gc,
84                                            x1 + inset + 1,
85                                            y1 - inset - 1, label);
86    error = xcb_request_check (c, cookie_text);
87    if (error) {
```

```
88      fprintf (stderr, "ERROR: can't paste text : %d\n", error->error_code);
89      xcb_disconnect (c);
90      exit (-1);
91    }
92
93    cookie_gc = xcb_free_gc (c, gc);
94    error = xcb_request_check (c, cookie_gc);
95    if (error) {
96      fprintf (stderr, "ERROR: can't free gc : %d\n", error->error_code);
97      xcb_disconnect (c);
98      exit (-1);
99    }
100 }
101
102 static void
103 text_draw (xcb_connection_t *c,
104            xcb_screen_t     *screen,
105            xcb_window_t      window,
106            int16_t           x1,
107            int16_t           y1,
108            const char       *label)
109 {
110   xcb_void_cookie_t    cookie_gc;
111   xcb_void_cookie_t    cookie_text;
112   xcb_generic_error_t *error;
113   xcb_gcontext_t       gc;
114   uint8_t              length;
115
116   length = strlen (label);
117
118   gc = gc_font_get(c, screen, window, "7x13");
119
120   cookie_text = xcb_image_text_8_checked (c, length, window, gc,
121                                           x1,
122                                           y1, label);
123   error = xcb_request_check (c, cookie_text);
124   if (error) {
125     fprintf (stderr, "ERROR: can't paste text : %d\n", error->error_code);
126     xcb_disconnect (c);
127     exit (-1);
128   }
129
130   cookie_gc = xcb_free_gc (c, gc);
131   error = xcb_request_check (c, cookie_gc);
132   if (error) {
133     fprintf (stderr, "ERROR: can't free gc : %d\n", error->error_code);
134     xcb_disconnect (c);
135     exit (-1);
136   }
137 }
138
139 static xcb_gc_t
140 gc_font_get (xcb_connection_t *c,
141              xcb_screen_t     *screen,
142              xcb_window_t      window,
```

```
143             const char       *font_name)
144 {
145   uint32_t             value_list[3];
146   xcb_void_cookie_t    cookie_font;
147   xcb_void_cookie_t    cookie_gc;
148   xcb_generic_error_t *error;
149   xcb_font_t           font;
150   xcb_gcontext_t       gc;
151   uint32_t             mask;
152 
153   font = xcb_generate_id (c);
154   cookie_font = xcb_open_font_checked (c, font,
155                                        strlen (font_name),
156                                        font_name);
157 
158   error = xcb_request_check (c, cookie_font);
159   if (error) {
160     fprintf (stderr, "ERROR: can't open font : %d\n", error->error_code);
161     xcb_disconnect (c);
162     return -1;
163   }
164 
165   gc = xcb_generate_id (c);
166   mask = XCB_GC_FOREGROUND | XCB_GC_BACKGROUND | XCB_GC_FONT;
167   value_list[0] = screen->black_pixel;
168   value_list[1] = screen->white_pixel;
169   value_list[2] = font;
170   cookie_gc = xcb_create_gc_checked (c, gc, window, mask, value_list);
171   error = xcb_request_check (c, cookie_gc);
172   if (error) {
173     fprintf (stderr, "ERROR: can't create gc : %d\n", error->error_code);
174     xcb_disconnect (c);
175     exit (-1);
176   }
177 
178   cookie_font = xcb_close_font_checked (c, font);
179   error = xcb_request_check (c, cookie_font);
180   if (error) {
181     fprintf (stderr, "ERROR: can't close font : %d\n", error->error_code);
182     xcb_disconnect (c);
183     exit (-1);
184   }
185 
186   return gc;
187 }
188 
189 static void
190 cursor_set (xcb_connection_t *c,
191             xcb_screen_t     *screen,
192             xcb_window_t      window,
193             int               cursor_id)
194 {
195   uint32_t             values_list[3];
196   xcb_void_cookie_t    cookie_font;
197   xcb_void_cookie_t    cookie_gc;
```

```
198   xcb_generic_error_t *error;
199   xcb_font_t           font;
200   xcb_cursor_t         cursor;
201   xcb_gcontext_t       gc;
202   uint32_t             mask;
203   uint32_t             value_list;
204
205   font = xcb_generate_id (c);
206   cookie_font = xcb_open_font_checked (c, font,
207                                        strlen ("cursor"),
208                                        "cursor");
209   error = xcb_request_check (c, cookie_font);
210   if (error) {
211     fprintf (stderr, "ERROR: can't open font : %d\n", error->error_code);
212     xcb_disconnect (c);
213     exit (-1);
214   }
215
216   cursor = xcb_generate_id (c);
217   xcb_create_glyph_cursor (c, cursor, font, font,
218                            cursor_id, cursor_id + 1,
219                            0, 0, 0,
220                            0, 0, 0);
221
222   gc = xcb_generate_id (c);
223   mask = XCB_GC_FOREGROUND | XCB_GC_BACKGROUND | XCB_GC_FONT;
224   values_list[0] = screen->black_pixel;
225   values_list[1] = screen->white_pixel;
226   values_list[2] = font;
227   cookie_gc = xcb_create_gc_checked (c, gc, window, mask, values_list);
228   error = xcb_request_check (c, cookie_gc);
229   if (error) {
230     fprintf (stderr, "ERROR: can't create gc : %d\n", error->error_code);
231     xcb_disconnect (c);
232     exit (-1);
233   }
234
235   mask = XCB_CW_CURSOR;
236   value_list = cursor;
237   xcb_change_window_attributes (c, window, mask, &value_list);
238
239   xcb_free_cursor (c, cursor);
240
241   cookie_font = xcb_close_font_checked (c, font);
242   error = xcb_request_check (c, cookie_font);
243   if (error) {
244     fprintf (stderr, "ERROR: can't close font : %d\n", error->error_code);
245     xcb_disconnect (c);
246     exit (-1);
247   }
248 }
249
250 int main ()
251 {
252   xcb_screen_iterator_t screen_iter;
```

```
253   xcb_connection_t     *c;
254   const xcb_setup_t    *setup;
255   xcb_screen_t         *screen;
256   xcb_generic_event_t  *e;
257   xcb_generic_error_t  *error;
258   xcb_void_cookie_t     cookie_window;
259   xcb_void_cookie_t     cookie_map;
260   xcb_window_t          window;
261   uint32_t              mask;
262   uint32_t              values[2];
263   int                   screen_number;
264   uint8_t               is_hand = 0;
265
266   /* getting the connection */
267   c = xcb_connect (NULL, &screen_number);
268   if (!c) {
269     fprintf (stderr, "ERROR: can't connect to an X server\n");
270     return -1;
271   }
272
273   /* getting the current screen */
274   setup = xcb_get_setup (c);
275
276   screen = NULL;
277   screen_iter = xcb_setup_roots_iterator (setup);
278   for (; screen_iter.rem != 0; --screen_number, xcb_screen_next (&screen_iter))
279     if (screen_number == 0)
280       {
281         screen = screen_iter.data;
282         break;
283       }
284   if (!screen) {
285     fprintf (stderr, "ERROR: can't get the current screen\n");
286     xcb_disconnect (c);
287     return -1;
288   }
289
290   /* creating the window */
291   window = xcb_generate_id (c);
292   mask = XCB_CW_BACK_PIXEL | XCB_CW_EVENT_MASK;
293   values[0] = screen->white_pixel;
294   values[1] =
295     XCB_EVENT_MASK_KEY_RELEASE |
296     XCB_EVENT_MASK_BUTTON_PRESS |
297     XCB_EVENT_MASK_EXPOSURE |
298     XCB_EVENT_MASK_POINTER_MOTION;
299   cookie_window = xcb_create_window_checked (c,
300                                              screen->root_depth,
301                                              window, screen->root,
302                                              20, 200, WIDTH, HEIGHT,
303                                              0, XCB_WINDOW_CLASS_INPUT_OUTPUT,
304                                              screen->root_visual,
305                                              mask, values);
306   cookie_map = xcb_map_window_checked (c, window);
307
```

```
308   /* error managing */
309   error = xcb_request_check (c, cookie_window);
310   if (error) {
311     fprintf (stderr, "ERROR: can't create window : %d\n", error->error_code);
312     xcb_disconnect (c);
313     return -1;
314   }
315   error = xcb_request_check (c, cookie_map);
316   if (error) {
317     fprintf (stderr, "ERROR: can't map window : %d\n", error->error_code);
318     xcb_disconnect (c);
319     return -1;
320   }
321
322   cursor_set (c, screen, window, 68);
323
324   xcb_flush(c);
325
326   while (1) {
327     e = xcb_poll_for_event(c);
328     if (e) {
329       switch (e->response_type & ~0x80) {
330       case XCB_EXPOSE: {
331         char *text;
332
333         text = "click here to change cursor";
334         button_draw (c, screen, window,
335                      (WIDTH - 7 * strlen(text)) / 2,
336                      (HEIGHT - 16) / 2, text);
337
338         text = "Press ESC key to exit...";
339         text_draw (c, screen, window, 10, HEIGHT - 10, text);
340         break;
341       }
342       case XCB_BUTTON_PRESS: {
343         xcb_button_press_event_t *ev;
344         int                       length;
345
346         ev = (xcb_button_press_event_t *)e;
347         length = strlen ("click here to change cursor");
348
349         if ((ev->event_x >= (WIDTH - 7 * length) / 2) &&
350             (ev->event_x <= ((WIDTH - 7 * length) / 2 + 7 * length + 6)) &&
351             (ev->event_y >= (HEIGHT - 16) / 2 - 19) &&
352             (ev->event_y <= ((HEIGHT - 16) / 2)))
353           is_hand = 1 - is_hand;
354
355         is_hand ? cursor_set (c, screen, window, 58) : cursor_set (c, screen,
    ↪  window, 68);
356       }
357       case XCB_KEY_RELEASE: {
358         xcb_key_release_event_t *ev;
359
360         ev = (xcb_key_release_event_t *)e;
361
```

```
362           switch (ev->detail) {
363             /* ESC */
364           case 9:
365             free (e);
366             xcb_disconnect (c);
367             return 0;
368           }
369         }
370         }
371         free (e);
372       }
373     }
374
375     return 0;
376   }
```

17. Translation of basic Xlib functions and macros

    The problem when you want to port an Xlib program to XCB is that you don't know if the Xlib function that you want to "translate" is a X Window one or an Xlib macro. In that section, we describe a way to translate the usual functions or macros that Xlib provides. It's usually just a member of a structure.

    (a) Members of the Display structure

        In this section, we look at how to translate the macros that return some members of the Display structure. They are obtained by using a function that requires a xcb_connection_t * or a member of the xcb_setup_t structure (via the function xcb_get_setup), or a function that requires that structure.

        i. ConnectionNumber

            This number is the file descriptor that connects the client to the server. You just have to use that function:

```
1  int xcb_get_file_descriptor (xcb_connection_t *c);
```

        ii. DefaultScreen

            That number is not stored by XCB. It is returned in the second parameter of the function xcb_connect. Hence, you have to store it yourself if you want to use it. Then, to get the xcb_screen_t structure, you have to iterate on the screens. The equivalent function of the Xlib's ScreenOfDisplay function can be found below. This is also provided in the xcb_aux_t library as xcb_aux_get_screen(). OK, here is the small piece of code to get that number:

```
1  xcb_connection_t *c;
2  int               screen_default_nbr;
3
4  /* you pass the name of the display you want to xcb_connect_t */
5
6  c = xcb_connect (display_name, &screen_default_nbr);
7
8  /* screen_default_nbr contains now the number of the default screen */
```

        iii. QLength

            Not documented yet.

            However, this points out a basic difference in philosophy between Xlib and XCB.

Xlib has several functions for filtering and manipulating the incoming and outgoing X message queues. XCB wishes to hide this as much as possible from the user, which allows for more freedom in implementation strategies.

iv. ScreenCount

You get the count of screens with the functions xcb_get_setup and xcb_setup_roots_iterator (if you need to iterate):

```
xcb_connection_t *c;
int               screen_count;

/* you init the connection */

screen_count = xcb_setup_roots_iterator (xcb_get_setup (c)).rem;

/* screen_count contains now the count of screens */
```

If you don't want to iterate over the screens, a better way to get that number is to use xcb_setup_roots_length_t:

```
xcb_connection_t *c;
int               screen_count;

/* you init the connection */

screen_count = xcb_setup_roots_length (xcb_get_setup (c));

/* screen_count contains now the count of screens */
```

v. ServerVendor

You get the name of the vendor of the server hardware with the functions xcb_get_setup and xcb_setup_vendor. Beware that, unlike Xlib, the string returned by XCB is not necessarily null-terminaled:

```
xcb_connection_t *c;
char             *vendor = NULL;
int               length;

/* you init the connection */
length = xcb_setup_vendor_length (xcb_get_setup (c));
vendor = (char *)malloc (length + 1);
if (vendor)
memcpy (vendor, xcb_setup_vendor (xcb_get_setup (c)), length);
vendor[length] = '\0';

/* vendor contains now the name of the vendor. Must be freed when not used
 ↪  anymore */
```

vi. ProtocolVersion

You get the major version of the protocol in the xcb_setup_t structure, with the function xcb_get_setup:

```
xcb_connection_t *c;
uint16_t          protocol_major_version;

/* you init the connection */

```

```
protocol_major_version = xcb_get_setup (c)->protocol_major_version;

/* protocol_major_version contains now the major version of the protocol */
```

vii. ProtocolRevision

You get the minor version of the protocol in the xcb_setup_t structure, with the function xcb_get_setup:

```
xcb_connection_t *c;
uint16_t          protocol_minor_version;

/* you init the connection */

protocol_minor_version = xcb_get_setup (c)->protocol_minor_version;

/* protocol_minor_version contains now the minor version of the protocol */
```

viii. VendorRelease

You get the number of the release of the server hardware in the xcb_setup_t structure, with the function xcb_get_setup:

```
xcb_connection_t *c;
uint32_t          release_number;

/* you init the connection */

release_number = xcb_get_setup (c)->release_number;

/* release_number contains now the number of the release of the server
 ↪  hardware */
```

ix. DisplayString

The name of the display is not stored in XCB. You have to store it by yourself.

x. BitmapUnit

You get the bitmap scanline unit in the xcb_setup_t structure, with the function xcb_get_setup:

```
xcb_connection_t *c;
uint8_t           bitmap_format_scanline_unit;

/* you init the connection */

bitmap_format_scanline_unit = xcb_get_setup
 ↪  (c)->bitmap_format_scanline_unit;

/* bitmap_format_scanline_unit contains now the bitmap scanline unit */
```

xi. BitmapBitOrder

You get the bitmap bit order in the xcb_setup_t structure, with the function xcb_get_setup:

```
xcb_connection_t *c;
uint8_t           bitmap_format_bit_order;

/* you init the connection */
```

```
5
6  bitmap_format_bit_order = xcb_get_setup (c)->bitmap_format_bit_order;
7
8  /* bitmap_format_bit_order contains now the bitmap bit order */
```

xii. BitmapPad

You get the bitmap scanline pad in the xcb_setup_t structure, with the function xcb_get_setup:

```
1  xcb_connection_t *c;
2  uint8_t          bitmap_format_scanline_pad;
3
4  /* you init the connection */
5
6  bitmap_format_scanline_pad = xcb_get_setup (c)->bitmap_format_scanline_pad;
7
8  /* bitmap_format_scanline_pad contains now the bitmap scanline pad */
```

xiii. ImageByteOrder

You get the image byte order in the xcb_setup_t structure, with the function xcb_get_setup:

```
1  xcb_connection_t *c;
2  uint8_t          image_byte_order;
3
4  /* you init the connection */
5
6  image_byte_order = xcb_get_setup (c)->image_byte_order;
7
8  /* image_byte_order contains now the image byte order */
```

(b) ScreenOfDisplay related functions

in Xlib, ScreenOfDisplay returns a Screen structure that contains several characteristics of your screen. XCB has a similar structure (xcb_screen_t), but the way to obtain it is a bit different. With Xlib, you just provide the number of the screen and you grab it from an array. With XCB, you iterate over all the screens to obtain the one you want. The complexity of this operation is O(n). So the best is to store this structure if you use it often. See screen_of_display just below.

Xlib provides generally two functions to obtain the characteristics related to the screen. One with the display and the number of the screen, which calls ScreenOfDisplay, and the other that uses the Screen structure. This might be a bit confusing. As mentioned above, with XCB, it is better to store the xcb_screen_t structure. Then, you have to read the members of this structure. That's why the Xlib functions are put by pairs (or more) as, with XCB, you will use the same code.

i. ScreenOfDisplay

This function returns the Xlib Screen structure. With XCB, you iterate over all the screens and once you get the one you want, you return it:

```
1  xcb_screen_t *screen_of_display (xcb_connection_t *c,
2                                   int               screen)
3  {
4    xcb_screen_iterator_t iter;
5
6    iter = xcb_setup_roots_iterator (xcb_get_setup (c));
```

```
  for (; iter.rem; --screen, xcb_screen_next (&iter))
    if (screen == 0)
      return iter.data;

  return NULL;
}
```

As mentioned above, you might want to store the value returned by this function.

All the functions below will use the result of that function, as they just grab a specific member of the xcb_screen_t structure.

ii. DefaultScreenOfDisplay

It is the default screen that you obtain when you connect to the X server. It suffices to call the screen_of_display function above with the connection and the number of the default screen.

```
xcb_connection_t *c;
int               screen_default_nbr;
xcb_screen_t     *default_screen;  /* the returned default screen */

/* you pass the name of the display you want to xcb_connect_t */

c = xcb_connect (display_name, &screen_default_nbr);
default_screen = screen_of_display (c, screen_default_nbr);

/* default_screen contains now the default root window, or a NULL window if
 ↪  no screen is found */
```

iii. RootWindow / RootWindowOfScreen

```
xcb_connection_t *c;
xcb_screen_t     *screen;
int               screen_nbr;
xcb_window_t      root_window = { 0 };  /* the returned window */

/* you init the connection and screen_nbr */

screen = screen_of_display (c, screen_nbr);
if (screen)
  root_window = screen->root;

/* root_window contains now the root window, or a NULL window if no screen
 ↪  is found */
```

iv. DefaultRootWindow

It is the root window of the default screen. So, you call ScreenOfDisplay with the default screen number and you get the root window as above:

```
xcb_connection_t *c;
xcb_screen_t     *screen;
int               screen_default_nbr;
xcb_window_t      root_window = { 0 };  /* the returned root window */

/* you pass the name of the display you want to xcb_connect_t */

```

```
c = xcb_connect (display_name, &screen_default_nbr);
screen = screen_of_display (c, screen_default_nbr);
if (screen)
  root_window = screen->root;

/* root_window contains now the default root window, or a NULL window if no
 ↪  screen is found */
```

v. DefaultVisual / DefaultVisualOfScreen

While a Visual is, in Xlib, a structure, in XCB, there are two types: xcb_visualid_t, which is the Id of the visual, and xcb_visualtype_t, which corresponds to the Xlib Visual. To get the Id of the visual of a screen, just get the root_visual member of a xcb_screen_t:

```
xcb_connection_t *c;
xcb_screen_t     *screen;
int               screen_nbr;
xcb_visualid_t    root_visual = { 0 };    /* the returned visual Id */

/* you init the connection and screen_nbr */

screen = screen_of_display (c, screen_nbr);
if (screen)
  root_visual = screen->root_visual;

/* root_visual contains now the value of the Id of the visual, or a NULL
 ↪  visual if no screen is found */
```

To get the xcb_visualtype_t structure, it's a bit less easy. You have to get the xcb_screen_t structure that you want, get its root_visual member, then iterate over the xcb_depth_ts and the xcb_visualtype_ts, and compare the xcb_visualid_t of these xcb_visualtype_ts: with root_visual:

```
xcb_connection_t *c;
xcb_screen_t     *screen;
int               screen_nbr;
xcb_visualid_t    root_visual = { 0 };
xcb_visualtype_t  *visual_type = NULL;    /* the returned visual type */

/* you init the connection and screen_nbr */

screen = screen_of_display (c, screen_nbr);
if (screen) {
  xcb_depth_iterator_t depth_iter;

  depth_iter = xcb_screen_allowed_depths_iterator (screen);
  for (; depth_iter.rem; xcb_depth_next (&depth_iter)) {
    xcb_visualtype_iterator_t visual_iter;

    visual_iter = xcb_depth_visuals_iterator (depth_iter.data);
    for (; visual_iter.rem; xcb_visualtype_next (&visual_iter)) {
      if (screen->root_visual == visual_iter.data->visual_id) {
        visual_type = visual_iter.data;
        break;
      }
    }
```

```
  }
}

/* visual_type contains now the visual structure, or a NULL visual
 ↪  structure if no screen is found */
```

vi. DefaultGC / DefaultGCOfScreen

This default Graphic Context is just a newly created Graphic Context, associated to the root window of a xcb_screen_t, using the black white pixels of that screen:

```
xcb_connection_t *c;
xcb_screen_t     *screen;
int               screen_nbr;
xcb_gcontext_t    gc = { 0 };    /* the returned default graphic context */

/* you init the connection and screen_nbr */

screen = screen_of_display (c, screen_nbr);
if (screen) {
  xcb_drawable_t draw;
  uint32_t       mask;
  uint32_t       values[2];

  gc = xcb_generate_id (c);
  draw = screen->root;
  mask = XCB_GC_FOREGROUND | XCB_GC_BACKGROUND;
  values[0] = screen->black_pixel;
  values[1] = screen->white_pixel;
  xcb_create_gc (c, gc, draw, mask, values);
}

/* gc contains now the default graphic context */
```

vii. BlackPixel / BlackPixelOfScreen

It is the Id of the black pixel, which is in the structure of an xcb_screen_t.

```
xcb_connection_t *c;
xcb_screen_t     *screen;
int               screen_nbr;
uint32_t          black_pixel = 0;    /* the returned black pixel */

/* you init the connection and screen_nbr */

screen = screen_of_display (c, screen_nbr);
if (screen)
  black_pixel = screen->black_pixel;

/* black_pixel contains now the value of the black pixel, or 0 if no screen
 ↪  is found */
```

viii. WhitePixel / WhitePixelOfScreen

It is the Id of the white pixel, which is in the structure of an xcb_screen_t.

```
xcb_connection_t *c;
xcb_screen_t     *screen;
int               screen_nbr;
```

```
uint32_t          white_pixel = 0;    /* the returned white pixel */

/* you init the connection and screen_nbr */

screen = screen_of_display (c, screen_nbr);
if (screen)
  white_pixel = screen->white_pixel;

/* white_pixel contains now the value of the white pixel, or 0 if no screen
 ↪  is found */
```

ix. DisplayWidth / WidthOfScreen

It is the width in pixels of the screen that you want, and which is in the structure of the corresponding xcb_screen_t.

```
xcb_connection_t *c;
xcb_screen_t     *screen;
int               screen_nbr;
uint32_t          width_in_pixels = 0;    /* the returned width in pixels
 ↪  */

/* you init the connection and screen_nbr */

screen = screen_of_display (c, screen_nbr);
if (screen)
  width_in_pixels = screen->width_in_pixels;

/* width_in_pixels contains now the width in pixels, or 0 if no screen is
 ↪  found */
```

x. DisplayHeight / HeightOfScreen

It is the height in pixels of the screen that you want, and which is in the structure of the corresponding xcb_screen_t.

```
xcb_connection_t *c;
xcb_screen_t     *screen;
int               screen_nbr;
uint32_t          height_in_pixels = 0;    /* the returned height in pixels
 ↪  */

/* you init the connection and screen_nbr */

screen = screen_of_display (c, screen_nbr);
if (screen)
  height_in_pixels = screen->height_in_pixels;

/* height_in_pixels contains now the height in pixels, or 0 if no screen is
 ↪  found */
```

xi. DisplayWidthMM / WidthMMOfScreen

It is the width in millimeters of the screen that you want, and which is in the structure of the corresponding xcb_screen_t.

```
xcb_connection_t *c;
xcb_screen_t     *screen;
int               screen_nbr;
```

```
uint32_t          width_in_millimeters = 0;    /* the returned width in
 ↪  millimeters */

/* you init the connection and screen_nbr */

screen = screen_of_display (c, screen_nbr);
if (screen)
  width_in_millimeters = screen->width_in_millimeters;

/* width_in_millimeters contains now the width in millimeters, or 0 if no
 ↪  screen is found */
```

xii. DisplayHeightMM / HeightMMOfScreen

It is the height in millimeters of the screen that you want, and which is in the structure of the corresponding xcb_screen_t.

```
xcb_connection_t *c;
xcb_screen_t     *screen;
int               screen_nbr;
uint32_t          height_in_millimeters = 0;    /* the returned height in
 ↪  millimeters */

/* you init the connection and screen_nbr */

screen = screen_of_display (c, screen_nbr);
if (screen)
  height_in_millimeters = screen->height_in_millimeters;

/* height_in_millimeters contains now the height in millimeters, or 0 if no
 ↪  screen is found */
```

xiii. DisplayPlanes / DefaultDepth / DefaultDepthOfScreen / PlanesOfScreen

It is the depth (in bits) of the root window of the screen. You get it from the xcb_screen_t structure.

```
xcb_connection_t *c;
xcb_screen_t     *screen;
int               screen_nbr;
uint8_t           root_depth = 0;  /* the returned depth of the root window
 ↪  */

/* you init the connection and screen_nbr */

screen = screen_of_display (c, screen_nbr);
if (screen)
  root_depth = screen->root_depth;

/* root_depth contains now the depth of the root window, or 0 if no screen
 ↪  is found */
```

xiv. DefaultColormap / DefaultColormapOfScreen

This is the default colormap of the screen (and not the (default) colormap of the default screen !). As usual, you get it from the xcb_screen_t structure:

```
xcb_connection_t *c;
xcb_screen_t     *screen;
```

```
int               screen_nbr;
xcb_colormap_t    default_colormap = { 0 };  /* the returned default
 ↪  colormap */

/* you init the connection and screen_nbr */

screen = screen_of_display (c, screen_nbr);
if (screen)
  default_colormap = screen->default_colormap;

/* default_colormap contains now the default colormap, or a NULL colormap
 ↪  if no screen is found */
```

xv. MinCmapsOfScreen

You get the minimum installed colormaps in the xcb_screen_t structure:

```
xcb_connection_t *c;
xcb_screen_t     *screen;
int               screen_nbr;
uint16_t          min_installed_maps = 0;  /* the returned minimum
 ↪  installed colormaps */

/* you init the connection and screen_nbr */

screen = screen_of_display (c, screen_nbr);
if (screen)
  min_installed_maps = screen->min_installed_maps;

/* min_installed_maps contains now the minimum installed colormaps, or 0 if
 ↪  no screen is found */
```

xvi. MaxCmapsOfScreen

You get the maximum installed colormaps in the xcb_screen_t structure:

```
xcb_connection_t *c;
xcb_screen_t     *screen;
int               screen_nbr;
uint16_t          max_installed_maps = 0;  /* the returned maximum
 ↪  installed colormaps */

/* you init the connection and screen_nbr */

screen = screen_of_display (c, screen_nbr);
if (screen)
  max_installed_maps = screen->max_installed_maps;

/* max_installed_maps contains now the maximum installed colormaps, or 0 if
 ↪  no screen is found */
```

xvii. DoesSaveUnders

You know if save_unders is set, by looking in the xcb_screen_t structure:

```
xcb_connection_t *c;
xcb_screen_t     *screen;
int               screen_nbr;
uint8_t           save_unders = 0;  /* the returned value of save_unders */
```

```
5
6  /* you init the connection and screen_nbr */
7
8  screen = screen_of_display (c, screen_nbr);
9  if (screen)
10   save_unders = screen->save_unders;
11
12 /* save_unders contains now the value of save_unders, or FALSE if no screen
    ↪  is found */
```

xviii. DoesBackingStore

You know the value of backing_stores, by looking in the xcb_screen_t structure:

```
1  xcb_connection_t *c;
2  xcb_screen_t     *screen;
3  int               screen_nbr;
4  uint8_t           backing_stores = 0;  /* the returned value of
    ↪  backing_stores */
5
6  /* you init the connection and screen_nbr */
7
8  screen = screen_of_display (c, screen_nbr);
9  if (screen)
10   backing_stores = screen->backing_stores;
11
12 /* backing_stores contains now the value of backing_stores, or FALSE if no
    ↪  screen is found */
```

xix. EventMaskOfScreen

To get the current input masks, you look in the xcb_screen_t structure:

```
1  xcb_connection_t *c;
2  xcb_screen_t     *screen;
3  int               screen_nbr;
4  uint32_t          current_input_masks = 0;  /* the returned value of
    ↪  current input masks */
5
6  /* you init the connection and screen_nbr */
7
8  screen = screen_of_display (c, screen_nbr);
9  if (screen)
10   current_input_masks = screen->current_input_masks;
11
12 /* current_input_masks contains now the value of the current input masks,
    ↪  or FALSE if no screen is found */
```

(c) Miscellaneous macros

i. DisplayOfScreen

in Xlib, the Screen structure stores its associated Display structure. This is not the case in the X Window protocol, hence, it's also not the case in XCB. So you have to store it by yourself.

ii. DisplayCells / CellsOfScreen

To get the colormap entries, you look in the xcb_visualtype_t structure, that you grab like here:

```
xcb_connection_t *c;
xcb_visualtype_t *visual_type;
uint16_t          colormap_entries = 0;  /* the returned value of the
 ↪  colormap entries */

/* you init the connection and visual_type */

if (visual_type)
  colormap_entries = visual_type->colormap_entries;

/* colormap_entries contains now the value of the colormap entries, or
 ↪  FALSE if no screen is found */
```