XlibScm

SCM Language X Interface
Version 5f3

Aubrey Jaffer

This manual documents the X Interface for SCM Language (version 5f3, February 2020).

Table of Contents

1 XlibScm

XlibScm is a SCM interface to *X*. The X Window System is a network-transparent window system that was designed at MIT. SCM is a portable Scheme implementation written in C. The interface can be compiled into SCM or, on those platforms supporting dynamic linking, compiled separately and loaded with `(require 'Xlib)`.

The most recent information about SCM can be found on SCM's *WWW* home page:
`http://people.csail.mit.edu/jaffer/SCM`

Much of this X documentation is dervied from:

Xlib - C Language X Interface
X Consortium Standard
X Version 11, Release 6.3

2 Display and Screens

`x:open-display` *display-name* [Function]

display-name Specifies the hardware display name, which determines the display and communications domain to be used. On a POSIX-conformant system, if the display-name is #f, it defaults to the value of the *DISPLAY* environment variable.

The encoding and interpretation of *display-name* is implementation-dependent. On POSIX-conformant systems, the *display-name* or *DISPLAY* environment variable can be a string in the format:

`hostname:number.screen-number` [Special Form]

hostname specifies the name of the host machine on which the display is physically attached. Follow the *hostname* with either a single colon (:) or a double colon (::).

number specifies the number of the display server on that host machine. You may optionally follow this display number with a period (.). A single CPU can have more than one display. Multiple displays are usually numbered starting with zero.

screen-number specifies the screen to be used on that server. Multiple screens can be controlled by a single X server. The *screen-number* sets an internal variable that can be accessed by using the x:default-screen procedure.

`x:close` *display* [Function]

display specifies the connection to the X server.

The `x:close` function closes the connection to the X server for the *display* specified and destroys all windows, resource IDs (Window, Font, Pixmap, Colormap, Cursor, and GContext), or other resources that the client has created on this display, unless the close-down mode of the resource has been changed (see `x:set-close-down-mode`). Therefore, these windows, resource IDs, and other resources should not be used again or an error will be generated. Before exiting, you should call *x:close-display* or *x:flush* explicitly so that any pending errors are reported.

`x:protocol-version` *display* [Function]

Returns cons of the major version number (11) of the X protocol associated with the connected *display* and the minor protocol revision number of the X server.

`x:server-vendor` *display* [Function]

Returns a string that provides some identification of the owner of the X server implementation. The contents of the string are implementation-dependent.

`x:vendor-release` *display* [Function]

Returns a number related to a vendor's release of the X server.

A display consists of one or more *Screens*. Each screen has a *root-window*, *default-graphics-context*, and *colormap*.

`x:screen-count` *display* [Function]
Returns the number of available screens.

`x:default-screen` *display* [Function]
Returns the default screen number specified by the `x:open-display` function. Use this screen number in applications which will use only a single screen.

`x:root-window` *display screen-number* [Function]
`x:root-window` *display* [Function]
screen-number, if givien, specifies the appropriate screen number on the host server. Otherwise the default-screen for *display* is used.

Returns the root window for the specified *screen-number*. Use `x:root-window` for functions that need a drawable of a particular screen or for creating top-level windows.

`x:root-window` *window* [Function]
Returns the root window for the specified *window*'s screen.

`x:default-colormap` *display screen-number* [Function]
`x:default-colormap` *display* [Function]
`x:default-colormap` *window* [Function]
Returns the default colormap of the specified screen.

`x:default-ccc` *display screen-number* [Function]
`x:default-ccc` *display* [Function]
`x:default-ccc` *window* [Function]
Returns the default Color-Conversion-Context (ccc) of the specified screen.

`x:default-gc` *display screen-number* [Function]
`x:default-gc` *display* [Function]
`x:default-gc` *window* [Function]
Returns the default graphics-context of the specified screen.

`x:screen-depths` *display screen-number* [Function]
`x:screen-depths` *display* [Function]
`x:screen-depths` *window* [Function]
Returns an array of depths supported by the specified screen.

The *Visual* type describes possible colormap depths and arrangements.

`x:default-visual` *display screen-number* [Function]
`x:default-visual` *display* [Function]
`x:default-visual` *window* [Function]
Returns the default Visual type for the specified screen.

`x:make-visual` *display depth class* [Function]
`x:make-visual` *window depth class* [Function]
The integer *depth* specifies the number of bits per pixel. The *class* argument specifies one of the possible visual classes for a screen:

- x:Static-Gray

- x:Static-Color
- x:True-Color
- x:Gray-Scale
- x:Pseudo-Color
- x:Direct-Color

`X:make-visual` returns a visual type for the screen specified by *display* or *window* if successful; #f if not.

`x:visual-class` *visual* [Function]
`x:visual-class` *screen* [Function]
`x:visual-class` *display* [Function]
Returns the (integer) visual class of its argument.

`x:visual-geometry` *visual* [Function]
`x:visual-geometry` *screen* [Function]
`x:visual-geometry` *display* [Function]
Returns a list of the:

- red_mask
- green_mask
- blue_mask
- colormap_size

`x:screen-cells` *display screen-number* [Function]
`x:screen-cells` *display* [Function]
`x:screen-cells` *window* [Function]
Returns the number of entries in the default colormap.

`x:screen-depth` *display screen-number* [Function]
Returns the depth of the root window of the specified screen.

`x:screen-depth` *display* [Function]
`x:screen-depth` *window* [Function]
`x:screen-depth` *visual* [Function]
Returns the depth of argument.

The *depth* of a window or pixmap is the number of bits per pixel it has. The *depth* of a graphics context is the depth of the drawables it can be used in conjunction with graphics output.

`x:screen-size` *display screen-number* [Function]
`x:screen-size` *display* [Function]
`x:screen-size` *window* [Function]
Returns a list of integer height and width of the screen in pixels.

`x:screen-dimensions` *display screen-number* [Function]
`x:screen-dimensions` *display* [Function]
`x:screen-dimensions` *window* [Function]
Returns a list of integer height and width of the screen in millimeters.

`x:screen-white` *display screen-number* [Function]
`x:screen-white` *display* [Function]
`x:screen-white` *window* [Function]
Returns the white pixel value of the specified screen.

`x:screen-black` *display screen-number* [Function]
`x:screen-black` *display* [Function]
`x:screen-black` *window* [Function]
Returns the black pixel value of the specified screen.

3 Drawables

A *Drawable* is either a window or pixmap.

3.1 Windows and Pixmaps

`x:create-window` *window position size border-width depth class visual field-name value . . .* [Function]

Creates and returns an unmapped Input-Output subwindow for a specified parent *window* and causes the X server to generate a CreateNotify event. The created window is placed on top in the stacking order with respect to siblings. Any part of the window that extends outside its parent *window* is clipped. The *border-width* for an x:Input-Only window must be zero.

The coordinate system has the X axis horizontal and the Y axis vertical with the origin [0, 0] at the upper-left corner. Coordinates are integral, in terms of pixels, and coincide with pixel centers. Each window and pixmap has its own coordinate system. For a window, the origin is inside the border at the inside, upper-left corner.

Class can be x:Input-Output, x:Input-Only, or x:Copy-From-Parent. For class x:Input-Output, the *visual* type and *depth* must be a combination supported for the screen. The *depth* need not be the same as the parent, but the parent must not be a window of class x:Input-Only. For an x:Input-Only window, the *depth* must be zero, and the *visual* must be one supported by the screen.

The returned window will have the attributes specified by *field-name*s and *value*.

`x:create-window` *window position size border-width border background* [Function]

The returned window inherits its depth, class, and visual from its parent. All other window attributes, except *background* and *border*, have their default values.

`x:create-pixmap` *drawable size depth* [Function]
`x:create-pixmap` *display size depth* [Function]

size is a list, vector, or pair of nonzero integers specifying the width and height desired in the new pixmap.

x:create-pixmap returns a new pixmap of the width, height, and *depth* specified. It is valid to pass an x:Input-Only window to the drawable argument. The *depth* argument must be one of the depths supported by the screen of the specified *drawable*.

`x:close` *window* [Function]

Destroys the specified *window* as well as all of its subwindows and causes the X server to generate a DestroyNotify event for each window. The window should not be used again. If the window specified by the *window* argument is mapped, it is unmapped automatically. The ordering of the DestroyNotify events is such that for any given window being destroyed, DestroyNotify is generated on any inferiors of the window before being generated on the window itself. The ordering among siblings and across subhierarchies is not otherwise constrained. If the *window* you specified is a root window, an error is signaled. Destroying a mapped *window* will generate x:Expose events on other windows that were obscured by the window being destroyed.

`x:close` *pixmap* [Function]
Deletes the association between the *pixmap* and its storage. The X server frees the pixmap storage when there are no references to it.

`x:window-geometry` *drawable* [Function]
Returns a list of:

coordinates
: `list` of x and y coordinates that define the location of the *drawable*. For a window, these coordinates specify the upper-left outer corner relative to its parent's origin. For pixmaps, these coordinates are always zero.

size
: `list` of the *drawable*'s dimensions (width and height). For a window, these dimensions specify the inside size, not including the border.

border-width
: The border width in pixels. If the *drawable* is a pixmap, this is zero.

depth
: The depth of the *drawable* (bits per pixel for the object).

`x:window-geometry-set!` *window field-name value . . .* [Function]
Changes the *Configuration* components specified by *field-name*s for the specified *window*.

These are the attributes settable by `x:window-geometry-set!`. That these attributes are encoded by small integers – just like those of the next section. Be warned therefore that confusion of attribute names will likely not signal errors, just cause mysterious behavior.

`x:CWX` [Attribute]
`x:CWY` [Attribute]
`x:CW-Width` [Attribute]
`x:CW-Height` [Attribute]
The x:CWX and x:CYY members are used to set the window's x and y coordinates, which are relative to the parent's origin and indicate the position of the upper-left outer corner of the window. The x:CW-Width and x:CW-Height members are used to set the inside size of the window, not including the border, and must be nonzero. Attempts to configure a root window have no effect.

If a window's size actually changes, the window's subwindows move according to their window gravity. Depending on the window's bit gravity, the contents of the window also may be moved

`x:CW-Border-Width` [Attribute]
The integer x:CW-Border-Width is used to set the width of the border in pixels. Note that setting just the border width leaves the outer-left corner of the window in a fixed position but moves the absolute position of the window's origin. It is an error to set the border-width attribute of an InputOnly window nonzero.

`x:CW-Sibling` [Attribute]
The sibling member is used to set the sibling window for stacking operations.

`x:CW-Stack-Mode` [Attribute]
: The x:CW-Stack-Mode member is used to set how the window is to be restacked and can be set to x:Above, x:Below, x:Top-If, x:Bottom-If, or x:Opposite.

If a sibling and a stack-mode are specified, the window is restacked as follows:

`x:Above` The window is placed just above the sibling.

`x:Below` The window is placed just below the sibling.

`x:Top-If` If the sibling occludes the window, the window is placed at the top of the stack.

`x:Bottom-If`
: If the window occludes the sibling, the window is placed at the bottom of the stack.

`x:Opposite`
: If the sibling occludes the window, the window is placed at the top of the stack. If the window occludes the sibling, the window is placed at the bottom of the stack.

If a stack-mode is specified but no sibling is specified, the window is restacked as follows:

`x:Above` The window is placed at the top of the stack.

`x:Below` The window is placed at the bottom of the stack.

`x:Top-If` If any sibling occludes the window, the window is placed at the top of the stack.

`x:Bottom-If`
: If the window occludes any sibling, the window is placed at the bottom of the stack.

`x:Opposite`
: If any sibling occludes the window, the window is placed at the top of the stack. If the window occludes any sibling, the window is placed at the bottom of the stack.

3.2 Window Attributes

`x:window-set!` *window field-name value . . .* [Function]
: Changes the components specified by *field-name*s for the specified *window*. The restrictions are the same as for `x:create-window`. The order in which components are verified and altered is server dependent. If an error occurs, a subset of the components may have been altered.

The `x:create-window` and `x:window-set!` procedures take five and one argument (respectively) followed by pairs of arguments, where the first is one of the property-name symbols (or its top-level value) listed below; and the second is the value to associate with that property.

`x:CW-Back-Pixmap` [Attribute]
: Sets the background pixmap of the *window* to the specified pixmap. The background pixmap can immediately be freed if no further explicit references to it are to be made.

If x:Parent-Relative is specified, the background pixmap of the window's parent is used, or on the root window, the default background is restored. It is an error to perform this operation on an x:Input-Only window. If the background is set to #f or None, the window has no defined background.

`x:CW-Back-Pixel` [Attribute]
Sets the background of the *window* to the specified pixel value. Changing the background does not cause the *window* contents to be changed. It is an error to perform this operation on an x:Input-Only window.

`x:CW-Border-Pixmap` [Attribute]
Sets the border pixmap of the *window* to the pixmap you specify. The border pixmap can be freed if no further explicit references to it are to be made. If you specify x:Copy-From-Parent, a copy of the parent window's border pixmap is used. It is an error to perform this operation on an x:Input-Only *window*.

`x:CW-Border-Pixel` [Attribute]
Sets the border of the *window* to the pixel *value*. It is an error to perform this operation on an x:Input-Only window.

`x:CW-Bit-Gravity` [Attribute]
`x:CW-Win-Gravity` [Attribute]
The bit gravity of a window defines which region of the window should be retained when an x:Input-Output window is resized. The default value for the bit-gravity attribute is x:Forget-Gravity. The window gravity of a window allows you to define how the x:Input-Output or x:Input-Only window should be repositioned if its parent is resized. The default value for the win-gravity attribute is x:North-West-Gravity.

If the inside width or height of a window is not changed and if the window is moved or its border is changed, then the contents of the window are not lost but move with the window. Changing the inside width or height of the window causes its contents to be moved or lost (depending on the bit-gravity of the window) and causes children to be reconfigured (depending on their win-gravity). For a change of width and height, the (x, y) pairs are defined:

Gravity Direction	Coordinates
x:North-West-Gravity	(0, 0)
x:North-Gravity	(Width/2, 0)
x:North-East-Gravity	(Width, 0)
x:West-Gravity	(0, Height/2)
x:Center-Gravity	(Width/2, Height/2)
x:East-Gravity	(Width, Height/2)
x:South-West-Gravity	(0, Height)
x:South-Gravity	(Width/2, Height)
x:South-East-Gravity	(Width, Height)

When a window with one of these bit-gravity values is resized, the corresponding pair defines the change in position of each pixel in the window. When a window with one of these win-gravities has its parent window resized, the corresponding pair

defines the change in position of the window within the parent. When a window is so repositioned, a x:Gravity-Notify event is generated (see section 10.10.5).

A bit-gravity of x:Static-Gravity indicates that the contents or origin should not move relative to the origin of the root window. If the change in size of the window is coupled with a change in position (x, y), then for bit-gravity the change in position of each pixel is (-x, -y), and for win-gravity the change in position of a child when its parent is so resized is (-x, -y). Note that x:Static-Gravity still only takes effect when the width or height of the window is changed, not when the window is moved.

A bit-gravity of x:Forget-Gravity indicates that the window's contents are always discarded after a size change, even if a backing store or save under has been requested. The window is tiled with its background and zero or more x:Expose events are generated. If no background is defined, the existing screen contents are not altered. Some X servers may also ignore the specified bit-gravity and always generate x:Expose events.

The contents and borders of inferiors are not affected by their parent's bit-gravity. A server is permitted to ignore the specified bit-gravity and use x:Forget-Gravity instead.

A win-gravity of x:Unmap-Gravity is like x:North-West-Gravity (the window is not moved), except the child is also unmapped when the parent is resized, and an x:Unmap-Notify event is generated.

`x:CW-Backing-Store` [Attribute]

Some implementations of the X server may choose to maintain the contents of x:Input-Output windows. If the X server maintains the contents of a window, the off-screen saved pixels are known as backing store. The backing store advises the X server on what to do with the contents of a window. The backing-store attribute can be set to x:Not-Useful (default), x:When-Mapped, or x:Always. A backing-store attribute of x:Not-Useful advises the X server that maintaining contents is unnecessary, although some X implementations may still choose to maintain contents and, therefore, not generate x:Expose events. A backing-store attribute of x:When-Mapped advises the X server that maintaining contents of obscured regions when the window is mapped would be beneficial. In this case, the server may generate an x:Expose event when the window is created. A backing-store attribute of x:Always advises the X server that maintaining contents even when the window is unmapped would be beneficial. Even if the window is larger than its parent, this is a request to the X server to maintain complete contents, not just the region within the parent window boundaries. While the X server maintains the window's contents, x:Expose events normally are not generated, but the X server may stop maintaining contents at any time.

When the contents of obscured regions of a window are being maintained, regions obscured by noninferior windows are included in the destination of graphics requests (and source, when the window is the source). However, regions obscured by inferior windows are not included.

`x:CW-Backing-Planes` [Attribute]
`x:CW-Backing-Pixel` [Attribute]

You can set backing planes to indicate (with bits set to 1) which bit planes of an x:Input-Output window hold dynamic data that must be preserved in backing store

and during save unders. The default value for the backing-planes attribute is all bits set to 1. You can set backing pixel to specify what bits to use in planes not covered by backing planes. The default value for the backing-pixel attribute is all bits set to 0. The X server is free to save only the specified bit planes in the backing store or the save under and is free to regenerate the remaining planes with the specified pixel value. Any extraneous bits in these values (that is, those bits beyond the specified depth of the window) may be simply ignored. If you request backing store or save unders, you should use these members to minimize the amount of off-screen memory required to store your window.

`x:CW-Override-Redirect` [Attribute]

To control window placement or to add decoration, a window manager often needs to intercept (redirect) any map or configure request. Pop-up windows, however, often need to be mapped without a window manager getting in the way. To control whether an x:Input-Output or x:Input-Only window is to ignore these structure control facilities, use the override-redirect flag.

The override-redirect flag specifies whether map and configure requests on this window should override a x:Substructure-Redirect-Mask on the parent. You can set the override-redirect flag to #t or #f (default). Window managers use this information to avoid tampering with pop-up windows.

`x:CW-Save-Under` [Attribute]

Some server implementations may preserve contents of x:Input-Output windows under other x:Input-Output windows. This is not the same as preserving the contents of a window for you. You may get better visual appeal if transient windows (for example, pop-up menus) request that the system preserve the screen contents under them, so the temporarily obscured applications do not have to repaint.

You can set the save-under flag to True or False (default). If save-under is True, the X server is advised that, when this window is mapped, saving the contents of windows it obscures would be beneficial.

`x:CW-Event-Mask` [Attribute]

The event mask defines which events the client is interested in for this x:Input-Output or x:Input-Only window (or, for some event types, inferiors of this window). The event mask is the bitwise inclusive OR of zero or more of the valid event mask bits. You can specify that no maskable events are reported by setting x:No-Event-Mask (default).

The following table lists the event mask constants you can pass to the event-mask argument and the circumstances in which you would want to specify the event mask:

Event Mask	Circumstances
x:No-Event-Mask	No events wanted
x:Key-Press-Mask	Keyboard down events wanted
x:Key-Release-Mask	Keyboard up events wanted
x:Button-Press-Mask	Pointer button down events wanted
x:Button-Release-Mask	Pointer button up events wanted
x:Enter-Window-Mask	Pointer window entry events wanted
x:Leave-Window-Mask	Pointer window leave events wanted

x:Pointer-Motion-Mask	Pointer motion events wanted
x:Pointer-Motion-Hint-Mask	If x:Pointer-Motion-Hint-Mask is selected in combination with one or more motion-masks, the X server is free to send only one x:Motion-Notify event (with the is_hint member of the X:Pointer-Moved-Event structure set to x:Notify-Hint) to the client for the event window, until either the key or button state changes, the pointer leaves the event window, or the client calls X:Query-Pointer or X:Get-Motion-Events. The server still may send x:Motion-Notify events without is_hint set to x:Notify-Hint.
x:Button1-Motion-Mask	Pointer motion while button 1 down
x:Button2-Motion-Mask	Pointer motion while button 2 down
x:Button3-Motion-Mask	Pointer motion while button 3 down
x:Button4-Motion-Mask	Pointer motion while button 4 down
x:Button5-Motion-Mask	Pointer motion while button 5 down
x:Button-Motion-Mask	Pointer motion while any button down
x:Keymap-State-Mask	Keyboard state wanted at window entry and focus in
x:Exposure-Mask	Any exposure wanted
x:Visibility-Change-Mask	Any change in visibility wanted
x:Structure-Notify-Mask	Any change in window structure wanted
x:Resize-Redirect-Mask	Redirect resize of this window
x:Substructure-Notify-Mask	Substructure notification wanted
x:Substructure-Redirect-Mask	Redirect structure requests on children
x:Focus-Change-Mask	Any change in input focus wanted
x:Property-Change-Mask	Any change in property wanted
x:Colormap-Change–Mask	Any change in colormap wanted
x:Owner-Grab-Button–Mask	Automatic grabs should activate with owner_events set to True

`x:CW-Dont-Propagate` [Attribute]

The do-not-propagate-mask attribute defines which events should not be propagated to ancestor windows when no client has the event type selected in this x:Input-Output or x:Input-Only window. The do-not-propagate-mask is the bitwise inclusive OR of zero or more of the following masks: x:Key-Press, x:Key-Release, x:Button-Press, x:Button-Release, x:Pointer-Motion, x:Button1Motion, x:Button2Motion, x:Button3Motion, x:Button4Motion, x:Button5Motion, and x:Button-Motion. You can specify that all events are propagated by setting x:No-Event-Mask (default).

`x:CW-Colormap` [Attribute]

The colormap attribute specifies which colormap best reflects the true colors of the x:Input-Output window. The colormap must have the same visual type as the window. X servers capable of supporting multiple hardware colormaps can use this information,

and window managers can use it for calls to X:Install-Colormap. You can set the colormap attribute to a colormap or to x:Copy-From-Parent (default).

If you set the colormap to x:Copy-From-Parent, the parent window's colormap is copied and used by its child. However, the child window must have the same visual type as the parent. The parent window must not have a colormap of x:None. The colormap is copied by sharing the colormap object between the child and parent, not by making a complete copy of the colormap contents. Subsequent changes to the parent window's colormap attribute do not affect the child window.

`x:CW-Cursor` [Attribute]

The cursor attribute specifies which cursor is to be used when the pointer is in the x:Input-Output or x:Input-Only window. You can set the cursor to a cursor or x:None (default).

If you set the cursor to x:None, the parent's cursor is used when the pointer is in the x:Input-Output or x:Input-Only window, and any change in the parent's cursor will cause an immediate change in the displayed cursor. On the root window, the default cursor is restored.

`x:window-ref` *window field-name . . .* [Function]

Returns a list of the components specified by *field-name*s for the specified *window*. Allowable *field-name*s are a subset of those for `x:window-set!`:

- x:CW-Back-Pixel
- x:CW-Bit-Gravity
- x:CW-Win-Gravity
- x:CW-Backing-Store
- x:CW-Backing-Planes
- x:CW-Backing-Pixel
- x:CW-Override-Redirect
- x:CW-Save-Under
- x:CW-Event-Mask
- x:CW-Dont-Propagate
- x:CW-Colormap

3.3 Window Properties and Visibility

`x:get-window-property` *window property* [Function]

Returns the (string or list of numbers) value of *property* of *window*.

`x:get-window-property` *window property #t* [Function]

Removes and returns the (string or list of numbers) value of *property* of *window*.

`x:list-properties` *window* [Function]

Returns a list of the properties (strings) defined for *window*.

In X parlance, a window which is hidden even when not obscured by other windows is *unmapped*; one which shows is *mapped*. It is an unfortunate name-collision with Scheme, and is ingrained in the attribute names.

`x:map-window` *window* [Function]

Maps the *window* and all of its subwindows that have had map requests. Mapping a window that has an unmapped ancestor does not display the window but marks it as eligible for display when the ancestor becomes mapped. Such a window is called unviewable. When all its ancestors are mapped, the window becomes viewable and will be visible on the screen if it is not obscured by another window. This function has no effect if the *window* is already mapped.

If the override-redirect of the window is False and if some other client has selected x:Substructure-Redirect-Mask on the parent window, then the X server generates a MapRequest event, and the `x:map-window` function does not map the *window*. Otherwise, the *window* is mapped, and the X server generates a MapNotify event.

If the *window* becomes viewable and no earlier contents for it are remembered, the X server tiles the *window* with its background. If the window's background is undefined, the existing screen contents are not altered, and the X server generates zero or more x:Expose events. If backing-store was maintained while the *window* was unmapped, no x:Expose events are generated. If backing-store will now be maintained, a full-window exposure is always generated. Otherwise, only visible regions may be reported. Similar tiling and exposure take place for any newly viewable inferiors.

If the window is an Input-Output window, `x:map-window` generates x:Expose events on each Input-Output window that it causes to be displayed. If the client maps and paints the window and if the client begins processing events, the window is painted twice. To avoid this, first ask for x:Expose events and then map the window, so the client processes input events as usual. The event list will include x:Expose for each window that has appeared on the screen. The client's normal response to an x:Expose event should be to repaint the window. This method usually leads to simpler programs and to proper interaction with window managers.

`x:map-subwindows` *window* [Function]

Maps all subwindows of a specified *window* in top-to-bottom stacking order. The X server generates x:Expose events on each newly displayed window. This may be much more efficient than mapping many windows one at a time because the server needs to perform much of the work only once, for all of the windows, rather than for each window.

`x:unmap-window` *window* [Function]

Unmaps the specified *window* and causes the X server to generate an UnmapNotify event. If the specified *window* is already unmapped, `x:unmap-window` has no effect. Normal exposure processing on formerly obscured windows is performed. Any child window will no longer be visible until another map call is made on the parent. In other words, the subwindows are still mapped but are not visible until the parent is mapped. Unmapping a *window* will generate x:Expose events on windows that were formerly obscured by it.

`x:unmap-subwindows` *window* [Function]

Unmaps all subwindows for the specified *window* in bottom-to-top stacking order. It causes the X server to generate an UnmapNotify event on each subwindow and x:Expose events on formerly obscured windows. Using this function is much more efficient than unmapping multiple windows one at a time because the server needs to perform much of the work only once, for all of the windows, rather than for each window.

4 Graphics Context

Most attributes of graphics operations are stored in *GC*s. These include line width, line style, plane mask, foreground, background, tile, stipple, clipping region, end style, join style, and so on. Graphics operations (for example, drawing lines) use these values to determine the actual drawing operation.

`x:create-gc` *drawable field-name value . . .* [Function]
Creates and returns graphics context. The graphics context can be used with any destination drawable having the same root and depth as the specified *drawable*.

`x:gc-set!` *graphics-context field-name value . . .* [Function]
Changes the components specified by *field-name*s for the specified *graphics-context*. The restrictions are the same as for `x:create-gc`. The order in which components are verified and altered is server dependent. If an error occurs, a subset of the components may have been altered.

`x:copy-gc-fields!` *gcontext-src gcontext-dst field-name . . .* [Function]
Copies the components specified by *field-name*s from *gcontext-src* to *gcontext-dst*. *Gcontext-src* and *gcontext-dst* must have the same root and depth.

`x:gc-ref` *graphics-context field-name . . .* [Function]
Returns a list of the components specified by *field-name*s . . . from the specified *graphics-context*.

GC Attributes

Both `x:create-gc` and `x:change-gc` take one argument followed by pairs of arguments, where the first is one of the property-name symbols (or its top-level value) listed below; and the second is the value to associate with that property.

`x:GC-Function` [Attribute]
The function attributes of a GC are used when you update a section of a drawable (the destination) with bits from somewhere else (the source). The function in a GC defines how the new destination bits are to be computed from the source bits and the old destination bits. x:G-Xcopy is typically the most useful because it will work on a color display, but special applications may use other functions, particularly in concert with particular planes of a color display. The 16 functions are:

```
x:G-Xclear          0
x:G-Xand            (AND src dst)
x:G-Xand-Reverse    (AND src (NOT dst))
x:G-Xcopy           src
x:G-Xand-Inverted   (AND (NOT src) dst)
x:G-Xnoop           dst
x:G-Xxor            (XOR src dst)
x:G-Xor             (OR src dst)
x:G-Xnor            (AND (NOT src) (NOT dst))
```

```
x:G-Xequiv           (XOR (NOT src) dst)
x:G-Xinvert          (NOT dst)
x:G-Xor-Reverse      (OR src (NOT dst))
x:G-Xcopy-Inverted   (NOT src)
x:G-Xor-Inverted     (OR (NOT src) dst)
x:G-Xnand            (OR (NOT src) (NOT dst))
x:G-Xset             1
```

`x:GC-Plane-Mask` [Attribute]

Many graphics operations depend on either pixel values or planes in a GC. The planes attribute is an integer which specifies which planes of the destination are to be modified, one bit per plane. A monochrome display has only one plane and will be the least significant bit of the integer. As planes are added to the display hardware, they will occupy more significant bits in the plane mask.

In graphics operations, given a source and destination pixel, the result is computed bitwise on corresponding bits of the pixels. That is, a Boolean operation is performed in each bit plane. The plane-mask restricts the operation to a subset of planes. `x:All-Planes` can be used to refer to all planes of the screen simultaneously. The result is computed by the following:

(OR (AND (FUNC src dst) plane-mask) (AND dst (NOT plane-mask)))

Range checking is not performed on a plane-mask value. It is simply truncated to the appropriate number of bits.

`x:GC-Foreground` [Attribute]
`x:GC-Background` [Attribute]

Range checking is not performed on the values for foreground or background. They are simply truncated to the appropriate number of bits.

Note that foreground and background are not initialized to any values likely to be useful in a window.

`x:GC-Line-Width` [Attribute]

The line-width is measured in pixels and either can be greater than or equal to one (wide line) or can be the special value zero (thin line).

Thin lines (zero line-width) are one-pixel-wide lines drawn using an unspecified, device-dependent algorithm. There are only two constraints on this algorithm.

- If a line is drawn unclipped from [x1,y1] to [x2,y2] and if another line is drawn unclipped from [x1+dx,y1+dy] to [x2+dx,y2+dy], a point [x,y] is touched by drawing the first line if and only if the point [x+dx,y+dy] is touched by drawing the second line.
- The effective set of points comprising a line cannot be affected by clipping. That is, a point is touched in a clipped line if and only if the point lies inside the clipping region and the point would be touched by the line when drawn unclipped.

A wide line drawn from [x1,y1] to [x2,y2] always draws the same pixels as a wide line drawn from [x2,y2] to [x1,y1], not counting cap-style and join-style. It is recommended that this property be true for thin lines, but this is not required. A line-width of zero

may differ from a line-width of one in which pixels are drawn. This permits the use of many manufacturers' line drawing hardware, which may run many times faster than the more precisely specified wide lines.

In general, drawing a thin line will be faster than drawing a wide line of width one. However, because of their different drawing algorithms, thin lines may not mix well aesthetically with wide lines. If it is desirable to obtain precise and uniform results across all displays, a client should always use a line-width of one rather than a linewidth of zero.

`x:GC-Line-Style` [Attribute]

The line-style defines which sections of a line are drawn:

`x:Line-Solid`
: The full path of the line is drawn.

`x:Line-Double-Dash`
: The full path of the line is drawn, but the even dashes are filled differently from the odd dashes (see fill-style) with x:Cap-Butt style used where even and odd dashes meet.

`x:Line-On-Off-Dash`
: Only the even dashes are drawn, and cap-style applies to all internal ends of the individual dashes, except x:Cap-Not-Last is treated as x:Cap-Butt.

`x:GC-Cap-Style` [Attribute]

The cap-style defines how the endpoints of a path are drawn:

`x:Cap-Not-Last`
: This is equivalent to x:Cap-Butt except that for a line-width of zero the final endpoint is not drawn.

`x:Cap-Butt`
: The line is square at the endpoint (perpendicular to the slope of the line) with no projection beyond.

`x:Cap-Round`
: The line has a circular arc with the diameter equal to the line-width, centered on the endpoint. (This is equivalent to x:Cap-Butt for line-width of zero).

`x:Cap-Projecting`
: The line is square at the end, but the path continues beyond the endpoint for a distance equal to half the line-width. (This is equivalent to x:Cap-Butt for line-width of zero).

`x:GC-Join-Style` [Attribute]

The join-style defines how corners are drawn for wide lines:

`x:Join-Miter`
: The outer edges of two lines extend to meet at an angle. However, if the angle is less than 11 degrees, then a x:Join-Bevel join-style is used instead.

`x:Join-Round`
: The corner is a circular arc with the diameter equal to the line-width, centered on the x:Join-point.

`x:Join-Bevel`
: The corner has x:Cap-Butt endpoint styles with the triangular notch filled.

`x:GC-Fill-Style` [Attribute]

The fill-style defines the contents of the source for line, text, and fill requests. For all text and fill requests (for example, X:Draw-Text, X:Fill-Rectangle, X:Fill-Polygon, and X:Fill-Arc); for line requests with linestyle x:Line-Solid (for example, X:Draw-Line, X:Draw-Segments, X:Draw-Rectangle, X:Draw-Arc); and for the even dashes for line requests with line-style x:Line-On-Off-Dash or x:Line-Double-Dash, the following apply:

`x:Fill-Solid`
: Foreground

`x:Fill-Tiled`
: Tile

`x:Fill-Opaque-Stippled`
: A tile with the same width and height as stipple, but with background everywhere stipple has a zero and with foreground everywhere stipple has a one

`x:Fill-Stippled`
: Foreground masked by stipple

When drawing lines with line-style x:Line-Double-Dash, the odd dashes are controlled by the fill-style in the following manner:

`x:Fill-Solid`
: Background

`x:Fill-Tiled`
: Same as for even dashes

`x:Fill-Opaque-Stippled`
: Same as for even dashes

`x:Fill-Stippled`
: Background masked by stipple

`x:GC-Fill-Rule` [Attribute]

The fill-rule defines what pixels are inside (drawn) for paths given in X:Fill-Polygon requests and can be set to x:Even-Odd-Rule or x:Winding-Rule.

`x:Even-Odd-Rule`
: A point is inside if an infinite ray with the point as origin crosses the path an odd number of times.

`x:Winding-Rule`
: A point is inside if an infinite ray with the point as origin crosses an unequal number of clockwise and counterclockwise directed path segments.

A clockwise directed path segment is one that crosses the ray from left to right as observed from the point. A counterclockwise segment is one that crosses the ray from right to left as observed from the point. The case where a directed line segment is coincident with the ray is uninteresting because you can simply choose a different ray that is not coincident with a segment.

For both x:Even-Odd-Rule and x:Winding-Rule, a point is infinitely small, and the path is an infinitely thin line. A pixel is inside if the center point of the pixel is inside and the center point is not on the boundary. If the center point is on the boundary, the pixel is inside if and only if the polygon interior is immediately to its right (x increasing direction). Pixels with centers on a horizontal edge are a special case and are inside if and only if the polygon interior is immediately below (y increasing direction).

`x:GC-Tile` [Attribute]
`x:GC-Stipple` [Attribute]
: The tile/stipple represents an infinite two-dimensional plane, with the tile/stipple replicated in all dimensions.

The tile pixmap must have the same root and depth as the GC, or an error results. The stipple pixmap must have depth one and must have the same root as the GC, or an error results. For stipple operations where the fill-style is x:Fill-Stippled but not x:Fill-Opaque-Stippled, the stipple pattern is tiled in a single plane and acts as an additional clip mask to be ANDed with the clip-mask. Although some sizes may be faster to use than others, any size pixmap can be used for tiling or stippling.

`x:GC-Tile-Stip-X-Origin` [Attribute]
`x:GC-Tile-Stip-Y-Origin` [Attribute]
: When the tile/stipple plane is superimposed on a drawable for use in a graphics operation, the upper-left corner of some instance of the tile/stipple is at the coordinates within the drawable specified by the tile/stipple origin. The tile/stipple origin is interpreted relative to the origin of whatever destination drawable is specified in a graphics request.

`x:GC-Font` [Attribute]
: The font to be used for drawing text.

`x:GC-Subwindow-Mode` [Attribute]
: You can set the subwindow-mode to x:Clip-By-Children or x:Include-Inferiors.

`x:Clip-By-Children`
: Both source and destination windows are additionally clipped by all viewable Input-Output children.

`x:Include-Inferiors`
: Neither source nor destination window is clipped by inferiors. This will result in including subwindow contents in the source and drawing

through subwindow boundaries of the destination. The use of `x:Include-Inferiors` on a window of one depth with mapped inferiors of differing depth is not illegal, but the semantics are undefined by the core protocol.

`x:GC-Graphics-Exposures` [Attribute]
The graphics-exposure flag controls x:Graphics-Expose event generation for X:Copy-Area and X:Copy-Plane requests (and any similar requests defined by extensions).

`x:GC-Clip-X-Origin` [Attribute]
`x:GC-Clip-Y-Origin` [Attribute]
The clip-mask origin is interpreted relative to the origin of whatever destination drawable is specified in a graphics request.

`x:GC-Clip-Mask` [Attribute]
The clip-mask restricts writes to the destination drawable. If the clip-mask is set to a pixmap, it must have depth one and have the same root as the GC, or an error results. If clip-mask is set to *x:None*, the pixels are always drawn regardless of the clip origin. The clip-mask also can be set by calling `X:Set-Region`. Only pixels where the clip-mask has a bit set to 1 are drawn. Pixels are not drawn outside the area covered by the clip-mask or where the clip-mask has a bit set to 0. The clip-mask affects all graphics requests. The clip-mask does not clip sources. The clip-mask origin is interpreted relative to the origin of whatever destination drawable is specified in a graphics request.

`x:GC-Dash-Offset` [Attribute]
Defines the phase of the pattern, specifying how many pixels into the dash-list the pattern should actually begin in any single graphics request. Dashing is continuous through path elements combined with a join-style but is reset to the dash-offset between each sequence of joined lines.

The unit of measure for dashes is the same for the ordinary coordinate system. Ideally, a dash length is measured along the slope of the line, but implementations are only required to match this ideal for horizontal and vertical lines. Failing the ideal semantics, it is suggested that the length be measured along the major axis of the line. The major axis is defined as the x axis for lines drawn at an angle of between -45 and +45 degrees or between 135 and 225 degrees from the x axis. For all other lines, the major axis is the y axis.

`x:GC-Dash-List` [Attribute]
There must be at least one element in the specified *dash-list*. The initial and alternating elements (second, fourth, and so on) of the *dash-list* are the even dashes, and the others are the odd dashes. Each element specifies a dash length in pixels. All of the elements must be nonzero. Specifying an odd-length list is equivalent to specifying the same list concatenated with itself to produce an even-length list.

`x:GC-Arc-Mode` [Attribute]
The arc-mode controls filling in the X:Fill-Arcs function and can be set to x:Arc-Pie-Slice or x:Arc-Chord.

`x:Arc-Pie-Slice`
: The arcs are pie-slice filled.

`x:Arc-Chord`
: The arcs are chord filled.

5 Cursor

`x:create-cursor` *display shape* [Function]

X provides a set of standard cursor shapes in a special font named *cursor*. Applications are encouraged to use this interface for their cursors because the font can be customized for the individual display type. The *shape* argument specifies which glyph of the standard fonts to use.

The hotspot comes from the information stored in the cursor font. The initial colors of a cursor are a black foreground and a white background (see X:Recolor-Cursor). The names of all cursor shapes are defined with the prefix XC: in `x11.scm`.

`x:create-cursor` *source-font source-char mask-font mask-char fgc bgc* [Function]

Creates a cursor from the source and mask bitmaps obtained from the specified font glyphs. The integer *source-char* must be a defined glyph in *source-font*. The integer *mask-char* must be a defined glyph in *mask-font*. The origins of the *source-char* and *mask-char* glyphs are positioned coincidently and define the hotspot. The *source-char* and *mask-char* need not have the same bounding box metrics, and there is no restriction on the placement of the hotspot relative to the bounding boxes.

`x:create-cursor` *source-font source-char #f #f fgc bgc* [Function]

If *mask-font* and *mask-char* are #f, all pixels of the source are displayed.

`x:create-cursor` *source-pixmap mask-pixmap fgc bgc origin* [Function]

mask-pixmap must be the same size as the pixmap defined by the *source-pixmap* argument. The foreground and background RGB values must be specified using *foreground-color* and *background-color*, even if the X server only has a x:Static-Gray or x:Gray-Scale screen. The hotspot must be a point within the *source-pixmap*.

`X:Create-Cursor` creates and returns a cursor. The *foreground-color* is used for the pixels set to 1 in the source, and the *background-color* is used for the pixels set to 0. Both source and mask must have depth one but can have any root. The *mask-pixmap* defines the shape of the cursor. The pixels set to 1 in *mask-pixmap* define which source pixels are displayed, and the pixels set to 0 define which pixels are ignored.

`x:create-cursor` *source-pixmap #f fgc bgc origin* [Function]

If *mask-pixmap* is #f, all pixels of the source are displayed.

6 Colormap

A *colormap* maps pixel values to *RGB* color space values.

`x:create-colormap` *window visual alloc-policy* [Function]

window specifies the window on whose screen you want to create a colormap. *visual* specifies a visual type supported on the screen. *alloc-policy* Specifies the colormap entries to be allocated. You can pass `X:Alloc-None` or `X:Alloc-All`.

The `X:Create-Colormap` function creates and returns a colormap of the specified *visual* type for the screen on which *window* resides. Note that *window* is used only to determine the screen.

'`X:Gray-Scale`'
'`X:Pseudo-Color`'
'`X:Direct-Color`'
: The initial values of the colormap entries are undefined.

'`X:Static-Gray`'
'`X:Static-Color`'
'`X:True-Color`'
: The entries have defined values, but those values are specific to *visual* and are not defined by X. The *alloc-policy* must be '`X:Alloc-None`'.

For the other visual classes, if *alloc-policy* is '`X:Alloc-None`', the colormap initially has no allocated entries, and clients can allocate them.

If *alloc-policy* is '`X:Alloc-All`', the entire colormap is allocated writable. The initial values of all allocated entries are undefined.

'`X:Gray-Scale`'
'`X:Pseudo-Color`'
: The effect is as if an `XAllocColorCells` call returned all pixel values from zero to N - 1, where N is the colormap entries value in *visual*.

'`X:Direct-Color`'
: The effect is as if an `XAllocColorPlanes` call returned a pixel value of zero and red_mask, green_mask, and blue_mask values containing the same bits as the corresponding masks in the specified visual.

To create a new colormap when the allocation out of a previously shared colormap has failed because of resource exhaustion, use:

`x:copy-colormap-and-free` *colormap* [Function]

Creates and returns a colormap of the same visual type and for the same screen as the specified *colormap*. It also moves all of the client's existing allocation from the specified *colormap* to the new colormap with their color values intact and their read-only or writable characteristics intact and frees those entries in the specified colormap. Color values in other entries in the new colormap are undefined. If the specified colormap was created by the client with alloc set to '`X:Alloc-All`', the new colormap is also created with '`X:Alloc-All`', all color values for all entries are

copied from the specified *colormap*, and then all entries in the specified *colormap* are freed. If the specified *colormap* was not created by the client with '`X:Alloc-All`', the allocations to be moved are all those pixels and planes that have been allocated by the client and that have not been freed since they were allocated.

A *colormap* maps pixel values to elements of the *RGB* datatype. An *RGB* is a list or vector of 3 integers, describing the red, green, and blue intensities respectively. The integers are in the range 0 - 65535.

`x:alloc-colormap-cells` *colormap ncolors nplanes* [Function]
`x:alloc-colormap-cells` *colormap ncolors nplanes contiguous?* [Function]

The `X:Alloc-Color-Cells` function allocates read/write color cells. The number of colors, *ncolors* must be positive and the number of planes, *nplanes* nonnegative. If *ncolors* and nplanes are requested, then *ncolors* pixels and nplane plane masks are returned. No mask will have any bits set to 1 in common with any other mask or with any of the pixels. By ORing together each pixel with zero or more masks, *ncolors* * 2^*nplanes* distinct pixels can be produced. All of these are allocated writable by the request.

'`x:Gray-Scale`'
'`x:Pseudo-Color`'
: Each mask has exactly one bit set to 1. If *contiguous?* is non-false and if all masks are ORed together, a single contiguous set of bits set to 1 is formed.

'`x:Direct-Color`'
: Each mask has exactly three bits set to 1. If *contiguous?* is non-false and if all masks are ORed together, three contiguous sets of bits set to 1 (one within each pixel subfield) is formed.

The RGB values of the allocated entries are undefined. `X:Alloc-Color-Cells` returns a list of two uniform arrays if it succeeded or #f if it failed. The first array has the pixels allocated and the second has the plane-masks.

`x:alloc-colormap-cells` *colormap ncolors rgb* [Function]
`x:alloc-colormap-cells` *colormap ncolors rgb contiguous?* [Function]

The specified *ncolors* must be positive; and *rgb* a list or vector of 3 nonnegative integers. If *ncolors* colors, *nreds* reds, *ngreens* greens, and *nblues* blues are requested, *ncolors* pixels are returned; and the masks have *nreds*, *ngreens*, and *nblues* bits set to 1, respectively. If *contiguous?* is non-false, each mask will have a contiguous set of bits set to 1. No mask will have any bits set to 1 in common with any other mask or with any of the pixels.

Each mask will lie within the corresponding pixel subfield. By ORing together subsets of masks with each pixel value, *ncolors* * 2(*nreds*+*ngreens*+*nblues*) distinct pixel values can be produced. All of these are allocated by the request. However, in the colormap, there are only *ncolors* * 2^*nreds* independent red entries, *ncolors* * 2^*ngreens* independent green entries, and *ncolors* * 2^*nblues* independent blue entries.

`X:Alloc-Color-Cells` returns a list if it succeeded or #f if it failed. The first element of the list has an array of the pixels allocated. The second, third, and fourth elements are the red, green, and blue plane-masks.

`x:free-colormap-cells` *colormap pixels planes* [Function]
`x:free-colormap-cells` *colormap pixels* [Function]
Frees the cells represented by pixels whose values are in the *pixels* unsigned-integer uniform-vector. The *planes* argument should not have any bits set to 1 in common with any of the pixels. The set of all pixels is produced by ORing together subsets of the *planes* argument with the pixels. The request frees all of these pixels that were allocated by the client. Note that freeing an individual pixel obtained from `X:Alloc-Colormap-Cells` with a planes argument may not actually allow it to be reused until all of its related pixels are also freed. Similarly, a read-only entry is not actually freed until it has been freed by all clients, and if a client allocates the same read-only entry multiple times, it must free the entry that many times before the entry is actually freed.

All specified pixels that are allocated by the client in the *colormap* are freed, even if one or more pixels produce an error. It is an error if a specified pixel is not allocated by the client (that is, is unallocated or is only allocated by another client) or if the colormap was created with all entries writable (by passing '`x:Alloc-All`' to `X:Create-Colormap`). If more than one pixel is in error, the one that gets reported is arbitrary.

`x:colormap-find-color` *colormap rgb* [Function]
rgb is a list or vector of 3 integers, describing the red, green, and blue intensities respectively; or an integer '`#x`*rrggbb*', packing red, green and blue intensities in the range 0 - 255.

`x:colormap-find-color` *colormap color-name* [Function]
The case-insensitive string *color_name* specifies the name of a color (for example, `red`)

`X:Colormap-Find-Color` allocates a read-only colormap entry corresponding to the closest RGB value supported by the hardware. `X:Colormap-Find-Color` returns the pixel value of the color closest to the specified *RGB* or *color_name* elements supported by the hardware, if successful; otherwise `X:Colormap-Find-Color` returns #f.

Multiple clients that request the same effective RGB value can be assigned the same read-only entry, thus allowing entries to be shared. When the last client deallocates a shared cell, it is deallocated.

`x:color-ref` *colormap pixel* [Function]
Returns a list of 3 integers, describing the red, green, and blue intensities respectively of the *colormap* entry of the cell indexed by *pixel*.

The integer *pixel* must be a valid index into *colormap*.

`X:Color-Set!` *colormap pixel rgb* [Function]
rgb is a list or vector of 3 integers, describing the red, green, and blue intensities respectively; or an integer '`#x`*rrggbb*', packing red, green and blue intensities in the range 0 - 255.

`X:Color-Set!` *colormap pixel color-name* [Function]

The case-insensitive string *color_name* specifies the name of a color (for example, `red`)

The integer *pixel* must be a valid index into *colormap*.

`X:Color-Set!` changes the *colormap* entry of the read/write cell indexed by *pixel*. If the *colormap* is an installed map for its screen, the changes are visible immediately.

`x:install-colormap` *colormap* [Function]

Installs the specified *colormap* for its associated screen. All windows associated with *colormap* immediately display with true colors. A colormap is associated with a window when the window is created or its attributes changed.

If the specified colormap is not already an installed colormap, the X server generates a ColormapNotify event on each window that has that colormap.

`x:ccc` *colormap* [Function]

Returns the Color-Conversion-Context of *colormap*.

7 Rendering

`x:flush` *display* [Function]
`x:flush` *window* [Function]
Flushes the output buffer. Some client applications need not use this function because the output buffer is automatically flushed as needed by calls to X:Pending, X:Next-Event, and X:Window-Event. Events generated by the server may be enqueued into the library's event queue.

`x:flush` *gc* [Function]
Forces sending of GC component changes.

Xlib usually defers sending changes to the components of a GC to the server until a graphics function is actually called with that GC. This permits batching of component changes into a single server request. In some circumstances, however, it may be necessary for the client to explicitly force sending the changes to the server. An example might be when a protocol extension uses the GC indirectly, in such a way that the extension interface cannot know what GC will be used.

`x:clear-area` *window* (*x-pos y-pos*) (*width height*) *expose?* [Function]
Paints a rectangular area in the specified *window* according to the specified dimensions with the *window*'s background pixel or pixmap. The subwindow-mode effectively is '`x:Clip-By-Children`'. If width is zero, it is replaced with the current width of the *window* minus x. If height is zero, it is replaced with the current height of the *window* minus y. If the *window* has a defined background tile, the rectangle clipped by any children is filled with this tile. If the *window* has background x:None, the contents of the *window* are not changed. In either case, if *expose?* is True, one or more x:Expose events are generated for regions of the rectangle that are either visible or are being retained in a backing store. If you specify a *window* whose class is x:Input-Only, an error results.

`x:fill-rectangle` *window gcontext position size* [Function]

Draw Strings

`x:draw-string` *drawable gc position string* [Function]
Position specifies coordinates relative to the origin of *drawable* of the origin of the first character to be drawn.

`x:draw-string` draws the characters of *string*, starting at *position*.

`x:image-string` *drawable gc position string* [Function]
Position specifies coordinates relative to the origin of *drawable* of the origin of the first character to be drawn.

`x:image-string` draws the characters *and background* of *string*, starting at *position*.

Draw Shapes

`x:draw-points` *drawable gc position . . .* [Function]
Position . . . specifies coordinates of the point to be drawn.

`x:draw-points` *drawable gc x y . . .* [Function]
(*x*, *y*) . . . specifies coordinates of the point to be drawn.

`x:draw-points` *drawable gc point-array* [Function]
point-array is a uniform short array of rank 2, whose rightmost index spans a range of 2.

The `X:Draw-Points` procedure uses the foreground pixel and function components of the *gc* to draw points into *drawable* at the positions (relative to the origin of *drawable*) specified.

`X:Draw-Points` uses these *gc* components: function, planemask, foreground, subwindow-mode, clip-x-origin, clip-y-origin, and clip-mask.

`x:draw-segments` *drawable gc pos1 pos2 . . .* [Function]
Pos1, *pos2*, . . . specify coordinates to be connected by segments.

`x:draw-segments` *drawable gc x1 y1 x2 y2 . . .* [Function]
(*x1*, *y1*), (*x2*, *y2*) . . . specify coordinates to be connected by segments.

`x:draw-segments` *drawable gc point-array* [Function]
point-array is a uniform short array of rank 2, whose rightmost index spans a range of 2.

The `X:Draw-Segments` procedure uses the components of the specified *gc* to draw multiple unconnected lines between disjoint adjacent pair of points passed as arguments. It draws the segments in order and does not perform joining at coincident endpoints. For any given line, `X:Draw-Segments` does not draw a pixel more than once. If thin (zero line-width) segments intersect, the intersecting pixels are drawn multiple times. If wide segments intersect, the intersecting pixels are drawn only once, as though the entire PolyLine protocol request were a single, filled shape. `X:Draw-Segments` treats all coordinates as relative to the origin of *drawable*.

`X:Draw-Segments` uses these *gc* components: function, plane-mask, line-width, line-style, cap-style, fill-style, subwindow-mode, clip-x-origin, clip-y-origin, and clip-mask, join-style. It also use these *gc* mode-dependent components: foreground, background, tile, stipple, tilestipple-x-origin, tile-stipple-y-origin, dash-offset, and dash-list.

`x:draw-lines` *drawable gc pos1 pos2 . . .* [Function]
Pos1, *pos2*, . . . specify coordinates to be connected by lines.

`x:draw-lines` *drawable gc x1 y1 x2 y2 . . .* [Function]
(*x1*, *y1*), (*x2*, *y2*) . . . specify coordinates to be connected by lines.

`x:draw-lines` *drawable gc point-array* [Function]
point-array is a uniform short array of rank 2, whose rightmost index spans a range of 2.

The `X:Draw-Lines` procedure uses the components of the specified *gc* to draw lines between each adjacent pair of points passed as arguments. It draws the lines in order. The lines join correctly at all intermediate points, and if the first and last points coincide, the first and last lines also join correctly. For any given line, `X:Draw-Lines` does not draw a pixel more than once. If thin (zero line-width) lines intersect, the intersecting pixels are drawn multiple times. If wide lines intersect, the intersecting pixels are drawn only once, as though the entire PolyLine protocol request were a single, filled shape. `X:Draw-Lines` treats all coordinates as relative to the origin of *drawable*.

`X:Draw-Lines` uses these *gc* components: function, plane-mask, line-width, line-style, cap-style, fill-style, subwindow-mode, clip-x-origin, clip-y-origin, and clip-mask, join-style. It also use these *gc* mode-dependent components: foreground, background, tile, stipple, tilestipple-x-origin, tile-stipple-y-origin, dash-offset, and dash-list.

`x:fill-polygon` *drawable gc pos1 pos2 . . .* [Function]
Pos1, *pos2*, . . . specify coordinates of the border path.

`x:fill-polygon` *drawable gc x1 y1 x2 y2 . . .* [Function]
(*x1*, *y1*), (*x2*, *y2*) . . . specify coordinates of the border path.

`x:fill-polygon` *drawable gc point-array* [Function]
point-array is a uniform short array of rank 2, whose rightmost index spans a range of 2.

The path is closed automatically if the last point in the list or *point-array* does not coincide with the first point.

The `X:Fill-Polygon` procedure uses the components of the specified *gc* to fill the region closed by the specified path. `X:Fill-Polygon` does not draw a pixel of the region more than once. `X:Fill-Polygon` treats all coordinates as relative to the origin of *drawable*.

`X:Fill-Polygon` uses these *gc* components: function, planemask, fill-style, fill-rule, subwindow-mode, clip-x-origin, clip-y-origin, and clip-mask. It also use these *gc* mode-dependent components: foreground, background, tile, stipple, tile-stipple-x-origin, and tile-stipple-y-origin.

8 Images

`x:read-bitmap-file` *drawable file* [Function]

9 Event

These three status routines always return immediately if there are events already in the queue.

`x:q-length` *display* [Function]
: Returns the length of the event queue for the connected *display*. Note that there may be more events that have not been read into the queue yet (see X:Events-Queued).

`x:pending` *display* [Function]
: Returns the number of events that have been received from the X server but have not been removed from the event queue.

`x:events-queued` *display* [Function]
: Returns the number of events already in the queue if the number is nonzero. If there are no events in the queue, `X:Events-Queued` attempts to read more events out of the application's connection without flushing the output buffer and returns the number read.

Both of these routines return an object of type *event*.

`x:next-event` *display* [Function]
: Removes and returns the first event from the event queue. If the event queue is empty, `X:Next-Event` flushes the output buffer and blocks until an event is received.

`x:peek-event` *display* [Function]
: Returns the first event from the event queue, but it does not remove the event from the queue. If the queue is empty, `X:Peek-Event` flushes the output buffer and blocks until an event is received.

Each event object has fields dependent on its sub-type.

`x:event-ref` *event field-name* [Function]

window	The window on which *event* was generated and is referred to as the event window.
root	is the event window's root window.
subwindow	If the source window is an inferior of the event window, the *subwindow* is the child of the event window that is the source window or the child of the event window that is an ancestor of the source window. Otherwise, '`None`'.

X-event:type
: An integer: *x:Key-Press*, *x:Key-Release*, *x:Button-Press*, *x:Button-Release*, *x:Motion-Notify*, *x:Enter-Notify*, *x:Leave-Notify*, *x:Focus-In*, *x:Focus-Out*, *x:Keymap-Notify*, *x:Expose*, *x:Graphics-Expose*, *x:No-Expose*, *x:Visibility-Notify*, *x:Create-Notify*, *x:Destroy-Notify*, *x:Unmap-Notify*, *x:Map-Notify*, *x:Map-Request*, *x:Reparent-Notify*, *x:Configure-Notify*, *x:Configure-Request*, *x:Gravity-Notify*, *x:Resize-Request*, *x:Circulate-Notify*, *x:Circulate-Request*, *x:Property-Notify*, *x:Selection-Clear*, *x:Selection-Request*, *x:Selection-Notify*, *x:Colormap-Notify*, *x:Client-Message*, or *x:Mapping-Notify*.

X-event:serial
: The serial number of the protocol request that generated the *event*.

X-event:send-event
: Boolean that indicates whether the event was sent by a different client.

X-event:time
: The time when the *event* was generated expressed in milliseconds.

X-event:x
X-event:y
: For window entry/exit events the x and y members are set to the coordinates of the pointer position in the event window. This position is always the pointer's final position, not its initial position. If the event window is on the same screen as the root window, x and y are the pointer coordinates relative to the event window's origin. Otherwise, x and y are set to zero.

  For expose events The x and y members are set to the coordinates relative to the drawable's origin and indicate the upper-left corner of the rectangle.

  For configure, create, gravity, and reparent events the x and y members are set to the window's coordinates relative to the parent window's origin and indicate the position of the upper-left outside corner of the created window.

X-event:x-root
X-event:y-root
: The pointer's coordinates relative to the root window's origin at the time of the *event*.

X-event:state	For keyboard, pointer and window entry/exit events, the state member is set to indicate the logical state of the pointer buttons and modifier keys just prior to the *event*, which is the bitwise inclusive OR of one or more of the button or modifier key masks: *x:Button1-Mask*, *x:Button2-Mask*, *x:Button3-Mask*, *x:Button4-Mask*, *x:Button5-Mask*, *x:Shift-Mask*, *x:Lock-Mask*, *x:Control-Mask*, *x:Mod1-Mask*, *x:Mod2-Mask*, *x:Mod3-Mask*, *x:Mod4-Mask*, and *x:Mod5-Mask*. For visibility events, the state of the window's visibility: *x:Visibility-Unobscured*, *x:Visibility-Partially-Obscured*, or *x:Visibility-Fully-Obscured*. For colormap events, indicates whether the colormap is installed or uninstalled: x:Colormap-Installed or x:Colormap-Uninstalled. For property events, indicates whether the property was changed to a new value or deleted: x:Property-New-Value or x:Property-Delete.
X-event:keycode	An integer that represents a physical key on the keyboard.
X-event:same-screen	Indicates whether the event window is on the same screen as the root window. If #t, the event and root windows are on the same screen. If #f, the event and root windows are not on the same screen.
X-event:button	The pointer button that changed state; can be the *x:Button1*, *x:Button2*, *x:Button3*, *x:Button4*, or *x:Button5* value.
X-event:is-hint	Detail of motion-notify events: *x:Notify-Normal* or *x:Notify-Hint*.
X-event:mode	Indicates whether the *event* is a normal event, pseudo-motion event when a grab activates, or a pseudo-motion event when a grab deactivates: *x:Notify-Normal*, *x:Notify-Grab*, or *x:Notify-Ungrab*.
X-event:detail	Indicates the notification detail: *x:Notify-Ancestor*, *x:Notify-Virtual*, *x:Notify-Inferior*, *x:Notify-Nonlinear*, or *x:Notify-Nonlinear-Virtual*.
X-event:focus	If the event window is the focus window or an inferior of the focus window, #t; otherwise #f.
X-event:width X-event:height	The size (extent) of the rectangle.

X-event:count	For mapping events is the number of keycodes altered. For expose events Is the number of Expose or GraphicsExpose events that are to follow. If count is zero, no more Expose events follow for this window. However, if count is nonzero, at least that number of Expose events (and possibly more) follow for this window. Simple applications that do not want to optimize redisplay by distinguishing between subareas of its window can just ignore all Expose events with nonzero counts and perform full redisplays on events with zero counts.
X-event:major-code	The major_code member is set to the graphics request initiated by the client and can be either X_CopyArea or X_CopyPlane. If it is X_CopyArea, a call to XCopyArea initiated the request. If it is X_CopyPlane, a call to XCopyPlane initiated the request.
X-event:minor-code	Not currently used.
X-event:border-width	For configure events, the width of the window's border, in pixels.
X-event:override-redirect	The override-redirect attribute of the window. Window manager clients normally should ignore this window if it is #t.
X-event:from-configure	True if the event was generated as a result of a resizing of the window's parent when the window itself had a win-gravity of x:Unmap-Gravity.
X-event:value-mask	Indicates which components were specified in the ConfigureWindow protocol request. The corresponding values are reported as given in the request. The remaining values are filled in from the current geometry of the window, except in the case of above (sibling) and detail (stack-mode), which are reported as None and Above, respectively, if they are not given in the request.
X-event:place	The window's position after the restack occurs and is either x:Place-On-Top or x:Place-On-Bottom. If it is x:Place-On-Top, the window is now on top of all siblings. If it is x:Place-On-Bottom, the window is now below all siblings.
X-event:new	indicate whether the colormap for the specified window was changed or installed or uninstalled and can be True or False. If it is True, the colormap was changed. If it is False, the colormap was installed or uninstalled.

X-event:format	Is 8, 16, or 32 and specifies whether the data should be viewed as a list of bytes, shorts, or longs
X-event:request	Indicates the kind of mapping change that occurred and can be *x:Mapping-Modifier*, *x:Mapping-Keyboard*, or *x:Mapping-Pointer*. If it is *x:Mapping-Modifier*, the modifier mapping was changed. If it is *x:Mapping-Keyboard*, the keyboard mapping was changed. If it is *x:Mapping-Pointer*, the pointer button mapping was changed.
X-event:first-keycode	The X-event:first-keycode is set only if the X-event:request was set to *x:Mapping-Keyboard*. The number in X-event:first-keycode represents the first number in the range of the altered mapping, and X-event:count represents the number of keycodes altered.

Indexes

Procedure and Macro Index

H

X

Variable Index

Concept Index

C

D

M

N

R

U

V

X