The eXene Library Manual

(Version 0.4)

February 11, 1993

John H. Reppy
Emden R. Gansner

AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974

Contents

Chapter 1

Introduction

This document accompanies the beta release of **eXene**, a multi-threaded **X**-window toolkit for Standard ML of New Jersey (**SML/NJ**). **eXene** is implemented on top of **Concurrent ML** (**CML**)[Rep90, Rep91a]. This document is not an introduction to **X**-windows, **SML**, or **CML**; it assumes a working knowledge of all of these systems. The reader should be familiar with **X**-windows at the level of **Xlib** (see [Nye90b] or [SG92] for information); in general, we do not describe the semantics of an **eXene** operation or type when it is the same as in **Xlib**. For information about **SML** see [Pau91]. **CML** is described in detail, including a language tutorial, in [Rep90], and a formal description of its semantics can be found in [Rep91b, Rep92]. Furthermore, this is not an **eXene** tutorial; look at the example applications included in the distribution to see how **eXene** applications are constructed.

eXene provides a user-interaction model that is similar to the one advocated by [Pik89] and [Haa90]. User actions, such as mouse motion, are mapped onto streams of messages, which are routed from a top-level window down the window hierarchy. Each window has an *environment* consisting of three input streams; mouse, keyboard and control, and an output stream for requesting services from its parent. The paper [GR91], which is included in the release documentation as the file `overview.ps`, provides an introduction to the **eXene** system.

A companion document, [GR93], describes the prototype widget library that we have built on top of **eXene**. It is intended that application programmers will primarily use widgets, which hide many of the complexities of the message routing. Widget implementors, however, will need to be conversant with the details of the library.

1.1 Roadmap

The user-level interface of **eXene** is organized into eight modules, which are grouped into the abstraction `EXene`. The `EXene` module is open by default, however, so its substructures are directly visible. Table 1.1 lists these modules together with the source files of their signatures, the chapter(s) in which they are described, and a brief description of their contents. In addition to the library, there is a collection of graphics utility modules. These are discussed in Chapter 9.

Module	Signature source file	Chapter(s)	Description
`Geometry`	`user/geometry.sml`	2	The basic geometric types and operations.
`EXeneBase`	`user/exene-base.sml`	3 and 6	Basic type definitions, display and screen operations, and other miscellaneous operations.
`Font`	`user/font.sml`	5	Support for handling fonts.
`Drawing`	`user/drawing.sml`	5	Types and operations for drawing text and graphics.
`ICCC`	`user/iccc.sml`	4 and 8	Support for the **X** *Inter-Client Conventions*.
`Interact`	`user/interact.sml`	7	Types and operations for handling user interaction.
`EXeneWin`	`user/exene-win.sml`	4	Window creation and management.
`StdCursor`	`user/std-cursor.sml`	3	The names of the cursors in the standard **X** cursor font.

Table 1.1: The **eXene** modules

Acknowledgements

Major portions of this system were implemented by the first author while he was a student at Cornell University and was supported , in part, by the NSF and ONR under NSF grant CCR-85-14862, and by the NSF under grant CCR-89-18233. Thomas Yan has provided corrections to this manual. Lennert Augustsson, Thomas M. Breuel, Timothy Griffin, Clifford Krumvieda, Huimin Lin, Roderick Moten and Thomas Yan have been helpful testers of **eXene**.

Chapter 2

Geometry

eXene follows the **X**-window convention and uses a pixel-based coordinate system with the origin located at the upper left corner of the screen. The `Geometry` structure provides definitions and operations for the basic geometric objects:

```
datatype point = PT of {x : int, y : int}
datatype size = SIZE of {wid : int, ht : int}
datatype line = LINE of (point * point)
datatype rect = RECT of {x : int, y : int, wid : int, ht : int}
datatype arc = ARC of {
    x : int, y : int,
    wid : int, ht : int,
    angle1 : int, angle2 : int
  }
datatype win_geom = WGEOM of {
      pos : point,
      sz : size,
      border : int
    }
```

Most of these types should be self-explanatory. Values of type `win_geom` describe the characteristics of a window's layout; see Chapter 4 for more information.[1] The rest of this chapter describes the operations provided by the `Geometry` structure.

2.1 Point operations

Figure 2.1 gives the signature of the point operations. The value `originPt` is the point $(0, 0)$. The operations `xCoordOfPt` and `yCoordOfPt` project the x and y coordinates of a point, respectively. Point arithmetic is supported by `addPt` and `subPt`; `scalePt` scales both coordinates by a scalar value. Two partial orders, `lessThanPt` and `lessEqPt`, on points are defined by:

$$\begin{array}{lcl} \texttt{lessThanPt}((x_1, y_1), (x_2, y_2)) & \Longleftrightarrow & (x_1 < x_2) \quad \textit{and} \quad (y_1 < y_2) \\ \texttt{lessEqPt}((x_1, y_1), (x_2, y_2)) & \Longleftrightarrow & (x_1 \leq x_2) \quad \textit{and} \quad (y_1 \leq y_2) \end{array}$$

[1]This type may be moved to the `EXeneWin` structure in the future.

```
val originPt   : point
val xCoordOfPt : point -> int
val yCoordOfPt : point -> int
val addPt      : (point * point) -> point
val subPt      : (point * point) -> point
val scalePt    : (int * point) -> point
val lessThanPt : (point * point) -> bool
val lessEqPt   : (point * point) -> bool
```

Figure 2.1: Point operations

2.2 Size operations

Size values are used to describe the dimension of rectangular regions; the size operations are given in Figure 2.2. Like points, sizes can be added, subtracted and scaled. In addition, a size can be added to a point

```
val addSz     : (size * size) -> size
val subSz     : (size * size) -> size
val scaleSz   : (int * size) -> size
val addSzToPt : (point * size) -> point
val limitPt   : (size * point) -> point
```

Figure 2.2: Size operations

by the `addSzToPt` operation. The function `limitPt` clips a point to be within the bounding box defined by the origin and the size, using orthogonal projection. Note that, in almost all uses in **eXene**, the width and height components of a `size` value should be nonnegative, but that this is not enforced by the type.

2.3 Rectangle operations

Figure 2.3 gives the signature of rectangle operations. Rectangles have an origin (a point), corresponding to the upper left corner, and a size; the function `mkRect` builds a rectangle from a point and size. The functions `originOfRect`, `sizeOfRect` and `originAndSzOfRect` extract these values from a rectangle. The function `cornerOfRect` returns the point that is the lower right corner of the rectangle; this is computed by translating the origin of the rectangle by its size. The function `clipPt` constrains a point to be within a rectangle (this is like `limitPt` on sizes). The function `translate` (respectively, `rtranslate`) translates a rectangle by adding (respectively, subtracting) the point to the rectangle's origin. The function `intersect` returns `true` if its two arguments intersect. The function `intersection` returns the largest rectangle contained by both of its arguments; the exception `Intersection` is raised if they do not intersect. The function `union` returns the smallest rectangle containing both of its arguments. The function `within` tests to see if a point is within a rectangle and the function `inside` tests to see if its first argument is inside its second. And lastly, the function `boundBox` returns the smallest rectangle containing a list of points (i.e., a bounding box).

```
val mkRect            : (point * size) -> rect
val originOfRect      : rect -> point
val sizeOfRect        : rect -> size
val originAndSzOfRect : rect -> (point * size)
val cornerOfRect      : rect -> point
val clipPt            : (rect * point) -> point
val translate         : (rect * point) -> rect
val rtranslate        : (rect * point) -> rect
val intersect         : (rect * rect) -> bool
exception Intersection
val intersection : (rect * rect) -> rect
val union        : (rect * rect) -> rect
val within       : (point * rect) -> bool
val inside       : (rect * rect) -> bool
val boundBox     : point list -> rect
```

Figure 2.3: Rectangle operations

Chapter 3

Basic eXene objects

This chapter gives an overview of the principal kinds of objects used by **X** and **eXene**. For the most part, these are defined in the structure `EXeneBase`.

EXene supports a number of opaque types; most of these are declared in the `EXeneBase` structure (see Figure 3.1). For those types that do not support equality, there are identity predicates that return true if, and

```
type   display
type   screen
type   window
type   font
type   pixmap
type   tile
type   cursor
eqtype std_cursor
eqtype atom
type   color
```

Figure 3.1: **eXene** types

only if, the two arguments are the same object. These predicates are given in Figure 3.2. The rest of this

```
val sameDisplay : (display * display) -> bool
val sameScreen  : (screen * screen) -> bool
val sameWindow  : (window * window) -> bool
val sameFont    : (font * font) -> bool
val samePixmap  : (pixmap * pixmap) -> bool
val sameTile    : (tile * tile) -> bool
val sameCursor  : (cursor * cursor) -> bool
val sameColor   : (color * color) -> bool
```

Figure 3.2: **eXene** types

chapter gives more detailed information about most of these types.

3.1 The display

In **X** terminology, each connection from a client program to an **X**-server is called a *display*. The functions

```
exception BadAddr of string
val openDisplay : string -> display
val closeDisplay : display -> unit
```

handle opening and closing a connection with an **X**-server. The argument to `openDisplay` specifies the **X**-server's host and the desired display and screen. In general, this string has the form:

host:*d*.*s*

where *host* specifies the machine running the server, *d* specifies the display number, and *s* specifies the screen number. The *host* may either be the string "`unix`," specifying a UNIX domain connection, or the string representation of an internet address (e.g., "`128.84.254.97`").[1] On systems running Sun's *Network Information Service* (formerly *Yellow Pages*), one can get the internet address using **ypmatch**(1). For example, to get the address of `maui`, one types the following (where `%` is the shell prompt):

```
% ypmatch maui hosts
128.84.254.97   maui.cs.cornell.edu maui.cs maui
%
```

This information is also sometimes available in the `/etc/hosts` file. Then one can open a connection for screen `0` of display `0` on `maui` by evaluating:

```
openDisplay "128.84.254.97:0.0"
```

The following abbreviations are supported in the argument to `openDisplay`:

`""`	==	`"unix:0.0"`
`":`*d*`"`	==	`"unix:`*d*`.0"`
`":`*d*`.`*s*`"`	==	`"unix:`*d*`.`*s*`"`
`"`*host*`:`*d*`"`	==	`"`*host*`:`*d*`.0"`

If a badly formatted address is supplied or if, for some reason, opening the connection fails, then the exception `BadAddr` is raised.

3.2 Screens

Each display supports one or more *screens*. One of these screens is designated as the *default screen* by the command to open the display connection (as described in Section 3.1). The following two functions extract screens from a display:

```
val defaultScreenOf : display -> screen
val screensOf       : display -> screen list
```

Screens have a number of characteristics; the functions in Figure 3.3 query these. The functions `sizeOfScr`

[1]Because of a limitation in the **SML/NJ** run-time system, an internet connection must be specified using the actual internet address. The examples directory `get-display` has an **SML** structure that implements display name lookup using a sub-process.

```
val displayOfScr : screen -> display
val sizeOfScr    : screen -> G.size
val sizeMMOfScr  : screen -> G.size
val depthOfScr   : screen -> int
```

Figure 3.3: Screen query functions

and `sizeMMOfScr` give the physical dimensions of the screen in pixels and millimeters, respectively, and the depth, or number of planes, of the screen is given by the function `depthOfScr`. Characteristics related to the support of color and grey scale screens are discussed in Chapter 6.

3.3 Drawables

Windows, pixmaps and tiles are the three kinds of *drawable* surfaces (i.e., rectangular arrays of pixels) supported by **eXene**. Windows are "on-screen" pixmaps and have additional properties associated with user interaction, the window manager, and the window hierarchy. Chapter 4 discusses the creation of windows and Chapter 5 discusses graphical operations on drawables. Tiles are immutable pixmaps. The remainder of this section discusses the operations on drawables provided in the `EXeneBase` structure.

3.3.1 Geometry of drawables

All drawables have *depth* and *size*; in addition, windows have a *position* (in their parent's coordinate system) and a *border width*. Operations to query these properties are given in Figure 3.4. Although pixmaps and tiles

```
val depthOfWin    : window -> int
val depthOfPixmap : pixmap -> int
val depthOfTile   : tile -> int
val sizeOfWin     : window -> G.size
val sizeOfPixmap  : pixmap -> G.size
val sizeOfTile    : tile -> G.size
val geomOfWin     : window -> {pos : G.point, sz : G.size, depth : int, border : int}
val geomOfPixmap  : pixmap -> {pos : G.point, sz : G.size, depth : int, border : int}
val geomOfTile    : tile -> {pos : G.point, sz : G.size, depth : int, border : int}
```

Figure 3.4: Drawable geometries

do not have a position or a border, to keep the interfaces uniform, the functions `geomOfWin`, `geomOfPixmap` and `geomOfTile` all return the same type. For pixmaps and tiles, the position is $(0, 0)$ and the border width is also 0.

3.3.2 Images

An *image* is a client-side description of a rectangular array of pixels. Images are useful for specifying icons and tiling patterns. Figure 3.5 gives the image type and operations. The function `imageFromAscii` is used to create an image value from an ASCII specification of the rows, which can be either a string representation

```
datatype image = IMAGE of {
    sz : G.size,
    data : string list list
  }

exception BadImageData

val imageFromAscii : (int * string list list) -> image
```

Figure 3.5: Images

of a binary number (with `"0b"` prefix) or of a hexadecimal number (with `"0x"` prefix). The integer parameter specifies the pixel width of the image; the height is determined by the length of the string list. For example, the following "tic-tac-toe" image

```
IMAGE{
    sz = G.SIZE{wid=8, ht=8},
    data = [[
        "\002\004", "\002\004", "\255\255", "\002\004",
        "\002\004", "\255\255", "\002\004", "\002\004"
      ]]
  }
```

can be specified in ASCII as either

```
imageFromAscii (8, [["0x24", "0x24", "0xff", "0x24", "0x24", "0xff", "0x24", "0x24"]])
```

or as

```
imageFromAscii (8, [[
    "0b00100100",
    "0b00100100",
    "0b11111111",
    "0b00100100",
    "0b00100100",
    "0b11111111",
    "0b00100100",
    "0b00100100"
  ]])
```

The exception `BadImageData` is raised by `imageFromAscii` if there is an error in the supplied data.

3.3.3 Pixmaps and tiles

Pixmaps are off-screen rectangular regions of pixels; tiles are immutable pixmaps. There are a number of different ways to create pixmaps and tiles; these are given in Figure 3.6. The function `createPixmap` creates an uninitialized pixmap of a given size and depth. Tiles and pixmaps may be created from either ASCII data (as described in Section 3.3.2) or from image data. The function `createTileFromPixmap` creates a tile with the same size and contents as an existing pixmap. Note that subsequent changes to the pixmap will not affect the tile.

```
val createPixmap              : screen -> (G.size * int) -> pixmap
val createPixmapFromAsciiData : screen -> (int * string list) -> pixmap
val createPixmapFromImage     : screen -> image -> pixmap

val createTileFromAsciiData : screen -> (int * string list) -> tile
val createTileFromImage     : screen -> image -> tile
val createTileFromPixmap    : pixmap -> tile
```

Figure 3.6: Pixmap and tile creation functions

There are also two functions for creating images from pixmaps or tiles:

```
val createImageFromPixmap : pixmap -> image
val createImageFromTile   : tile -> image
```

3.4 Cursors

Currently, **eXene** only supports the, so called, *standard cursors*. These are described below. The color of a cursor can be changed using the function

```
val recolorCursor : {cursor : cursor, fore_rgb : rgb, back_rgb : rgb} -> unit
```

where the type `rgb` is defined as[2]

```
datatype rgb = RGB of {red : int, green : int, blue : int}
```

The function

```
val changeActiveGrabCursor : display -> cursor -> unit
```

is used to change the cursor during an *active grab* of the pointer device (see Chapter 7).

3.4.1 The standard cursors

The **X** distribution provides a standard cursor font. The **eXene** type `std_cursor` is used to name these cursors. Given a standard cursor name, one can create a cursor using the function

```
val stdCursor : display -> std_cursor -> cursor
```

The structure `StdCursor` contains all of the standard cursor names; see Appendix I of [Nye90c] for more information.

[2]In the long run, we hope to have a device independent color model for cursor colors and eliminate the `rgb` type.

3.5 Miscellaneous types and operations

3.5.1 Other display operations

There are a number of operations that are global to a display connection. For example, the function

```
val ringBell : display -> int -> unit
```

generates an audible beep on the user's terminal. The integer argument, which should be in the range $[-100, 100]$ defines the volume of the beep according to the following formula

$$v = \begin{cases} b - \frac{bp}{100} + p & \mathit{if}\, p \geq 0 \\ b + \frac{bp}{100} & \mathit{if}\, p < 0 \end{cases}$$

where b is the base volume and p is the argument value. Currently, the base volume cannot be changed from **eXene**.

3.5.2 Window hash tables

Because it is often necessary to key searches by a window's identity, generic hash tables with window keys are provided.

```
type 'a window_map
exception WindowNotFound
val newMap : unit -> '1a window_map
val insert : '2a window_map -> (window * '2a) -> unit
val find : 'a window_map -> window -> 'a
val remove : 'a window_map -> window -> 'a
val list : 'a window_map -> 'a list
```

The operations are fairly obvious: `newMap` creates a new hash table; `insert` inserts an item keyed by a window; `find` returns the item keyed by the given window; `remove` removes and returns an item; and `list` returns a list of the items in the table. The exception `WindowNotFound` is raised by `find` and `remove` in the case that the given window is not in the table.

3.5.3 Gravity

The `EXeneBase` structure also includes a datatype for specifying a windows "gravity," i.e., how it is going to be relocated when its parent is resized.

```
datatype gravity
  = ForgetGravity           (* bit gravity only *)
  | UnmapGravity            (* window gravity only *)
  | NorthWestGravity
  | NorthGravity
  | NorthEastGravity
  | WestGravity
  | CenterGravity
  | EastGravity
  | SouthWestGravity
  | SouthGravity
  | SouthEastGravity
  | StaticGravity
```

This type is used in window manager and size hints (see Sections 4 and 8).

Chapter 4

Windows

Windows are the basic unit of graphical output and user input. This chapter discusses the operations for window creation and organization; Chapter 5 discusses drawing on windows and Chapter 7 discusses the **eXene** user interaction model. The types and operations discussed in this chapter can be found in the structure `EXeneWin`.

eXene supports five kinds of windows: *top-level* windows, *subwindows*, *transient* windows, *popup* windows, and *input-only* windows. A top-level window is the root of a window hierarchy, and is managed by the window manager. The descendants of a top-level window in a window hierarchy are called subwindows. Transient windows are managed top-level windows with short lifetimes, such as dialogue boxes, and popup windows are unmanaged top-level windows, such as menus. Input-only windows provide a new input context for an existing window.

4.1 Window creation

Figure 4.1 gives the signatures of the creation functions for the different kinds of windows. The first argument for these functions is the parent of the window: this is the screen for top-level and popup windows, and another window for subwindows. The second argument is a record that specifies the window's preferred geometry,[1] border width, and background color. The point supplied in the window geometry is the origin of the window in its parent's coordinate system. Note that this is the upper left corner of the entire window, including the window's border, and not the origin of the window's drawing region, whose dimensions are specified by the size component of the geometry. For top-level and popup windows, the create function returns the window and its input environment, which is derived from the **X**-event stream.

Once a top-level window has been created, certain window-manager properties can be set for it. This should be done before the window is mapped using the following function:

[1]For top-level windows, the actual geometry is usually decided by the user or window manager, with hints provided by `setWMProperties`.

```
val createSimpleTopWin : EXB.screen -> {
          geom : G.win_geom,
          border : EXB.color,
          backgrnd : EXB.color
        } -> (window * Interact.in_env)
val createSimpleSubwin : window -> {
          geom : G.win_geom,
          border : EXB.color option,
          backgrnd : EXB.color option
        } -> window
val createTransientWin : EXB.window -> {
          geom : G.win_geom,
          border : EXB.color,
          backgrnd : EXB.color
        } -> (window * Interact.in_env)
val createSimplePopupWin : EXB.screen -> {
          geom : G.win_geom,
          border : EXB.color,
          backgrnd : EXB.color
        } -> (window * Interact.in_env)
val createInputOnlyWin : window -> G.rect -> window
exception InputOnly
```

Figure 4.1: Window creation functions

```
val setWMProperties : window -> {
        win_name    : string option,
        icon_name   : string option,
        argv        : string list,
        size_hints  : ICCC.size_hints list,
        wm_hints    : ICCC.wm_hints list,
        class_hints : {res_class : string, res_name : string} option
      } -> unit
```

See Chapter 8 for a description of the size and window manager hints.

To get a window to actually appear on the screen, it (and all of its ancestors) must be mapped. The function

```
val mapWin : window -> unit
```

maps a window. To avoid having the screen flash, it is a good idea to map a window hierarchy from the bottom up, mapping the top-level window last. The functions

```
val unmapWin : window -> unit
val destroyWin : window -> unit
```

respectively unmap and destroy a window. In **X**, destroying a window implicitly destroys all of its subwindows.

4.2 Window configuration

The configuration of a window (position, size, etc.) can be controlled using the following operations:

```
val configureWin : window -> window_config list -> unit
val moveWin : window -> G.point -> unit
val resizeWin : window -> G.size -> unit
val moveAndResizeWin : window -> G.rect -> unit
```

The constructors of the `window_config` datatype are described in Table 4.1.

Description	**Value**
Window origin	`WC_Origin` *point*
Window size	`WC_Size` *size*
Window's border width	`WC_BorderWid` *n*
Stacking mode	`WC_StackMode` *mode*
Stacking mode relative to a sibling window	`WC_RelStackMode(`*win*`,` *mode*`)`

Table 4.1: Window configuration values

4.3 Window attributes

The cursor used by a window is set using the following function:

```
val setCursor : window -> EXB.cursor option -> unit
```

```
val setBackground : window -> EXB.color option -> unit
val changeWinAttrs : window -> window_attr list -> unit
```

4.4 Other window operations

There are a few other miscellaneous operations on windows. Given a point in a window's coordinate system, it can be translated to a point in the screen's absolute coordinate system by the function:

```
val winPtToScrPt : window -> G.point -> G.point
```

The screen and dispay associated with a window can be found using the functions:

```
val screenOfWin : window -> EXB.screen
val displayOfWin : window -> EXB.display
```

Chapter 5

Drawing

This chapter describes the various operations and types provided by **eXene** to support bitmap graphics. These operations and types are defined in the `Drawing` structure.

5.1 Pens

A *pen* is similar to the graphics context provided by **Xlib**. The principal differences are that pens are immutable, do not specify a font, and can specify clipping rectangles and dash lists (which are handled separately in the **X** protocol). The basic operations on pens are:

```
type pen

val newPen     : pen_val list -> pen
val updatePen  : (pen * pen_val list) -> pen
val defaultPen : pen
```

The datatype `pen_val` is used to specify the non-default values when creating a new pen. Table 5.1 lists the components of a pen, the possible values for each component, and the default value. The drawing functions used with `PV_Function` are defined by the datatype `graphics_op`; these specify how source and destination colors are logically combined in graphics operations, and are explained by Table 5.2. The `newPen` function creates a new pen with the specified values. The `updatePen` function does a non-destructive update of a pen. **eXene** provides a default pen for those rare instances when all default values are appropriate. The semantics of a pen basically follow the semantics of the **Xlib** graphics contexts (see Chapter 5 of [Nye90b]).

5.2 Fonts

Unlike in **Xlib**, fonts are not part of the graphics context (pen in our case). The text drawing operations (see Section 5.6) take the font as a separate argument.

5.2.1 Opening a font

To open a font, use the function

Component	**Values**	**Default**
Function	`PV_Function` *op*	`PV_Function OP_Copy`
Plane mask	`PV_PlaneMask` *mask*	*all ones*
Foreground color	`PV_Foreground` *color*	`PV_Foreground color0`
Background color	`PV_Background` *color*	`PV_Background color1`
Line width	`PV_LineWidth` *width*	`PV_LineWidth 0`

Line style	`PV_LineStyle_Solid`
	`PV_LineStyle_OnOffDash`
	`PV_LineStyle_DoubleDash`
		`PV_LineStyle_Solid`

Cap style	`PV_CapStyle_Butt`
	`PV_CapStyle_NotLast`
	`PV_CapStyle_Round`
	`PV_CapStyle_Projecting`
		`PV_CapStyle_Butt`

Join style	`PV_JoinStyle_Miter`
	`PV_JoinStyle_Round`
	`PV_JoinStyle_Bevel`
		`PV_JoinStyle_Miter`

Fill style	`PV_FillStyle_Solid`
	`PV_FillStyle_Tiled`
	`PV_FillStyle_Stippled`
	`PV_FillStyle_OpaqueStippled`
		`PV_FillStyle_Solid`

Fill rule	`PV_FillRule_EvenOdd` `PV_FillRule_Winding`	`PV_FillRule_EvenOdd`
Arc mode	`PV_ArcMode_PieSlice` `PV_ArcMode_Chord`	`PV_ArcMode_PieSlice`
Tile	`PV_Tile` *tile*	
Stipple	`PV_Stipple` *tile*	
Tile/stipple origin	`PV_TSOrigin` *pt*	`PV_TSOrigin(PT{x=0,y=0})`
Subwindow mode	`PV_ClipByChildren` `PV_IncludeInferiors`	`PV_ClipByChildren`
Clip origin	`PV_ClipOrigin` *pt*	`PV_ClipOrigin(PT{x=0,y=0})`
Clip mask	`PV_ClipMask_None` `PV_ClipMask` *pixmap* `PV_ClipMask_UnsortedRects` *rects* `PV_ClipMask_YSortedRects` *rects* `PV_ClipMask_YXSortedRects` *rects* `PV_ClipMask_YXBandedRects` *rects*	`PV_ClipMask_None`
Dash offset	`PV_DashOffset` *n*	`PV_DashOffset 0`
Dashes	`PV_Dash_Fixed` *n* `PV_DashList` *dashes*	`PV_Dash_Fixed 4`

Table 5.1: Pen component values

`OP_Clr`	$dst \leftarrow 0$
`OP_And`	$dst \leftarrow src \wedge dst$
`OP_AndNot`	$dst \leftarrow src \wedge \overline{dst}$
`OP_Copy`	$dst \leftarrow src$
`OP_AndInverted`	$dst \leftarrow \overline{src} \wedge dst$
`OP_Nop`	$dst \leftarrow dst$
`OP_Xor`	$dst \leftarrow src \oplus dst$
`OP_Or`	$dst \leftarrow src \vee dst$
`OP_Nor`	$dst \leftarrow \overline{src} \wedge \overline{dst}$
`OP_Equiv`	$dst \leftarrow \overline{src} \oplus dst$
`OP_Not`	$dst \leftarrow \overline{dst}$
`OP_OrNot`	$dst \leftarrow src \vee \overline{dst}$
`OP_CopyNot`	$dst \leftarrow \overline{src}$
`OP_OrInverted`	$dst \leftarrow \overline{src} \vee dst$
`OP_Nand`	$dst \leftarrow \overline{src} \vee \overline{dst}$
`OP_Set`	$dst \leftarrow 1$

Table 5.2: Graphical operators
The symbol $\vee$ is logical *or*, $\wedge$ is logical *and*, and $\oplus$ is exclusive-or, and the notation $\overline{x}$ is the logical negation of x.

```
exception FontNotFound
val openFont : EXB.display -> string -> font
```

which returns the opened font or raises the exception `FontNotFound`, if the font cannot be found in the server's font path. For information on font naming conventions, see [MIT89].

5.2.2 Character metrics

Fonts and their related character metrics follow the standard **X** model. However, in **eXene**, font information is viewed as logically part of the font; there is no separate font information data structure. Figure 5.1 gives the types and operations related to the character metrics of fonts. Look in [Nye90b] or [SG92] for an explanation of the different character metrics.

The function `charInfoOf` returns information about the give character (specified as an ordinal); it raises the the exception `NoCharInfo` if the integer argument does not correspond to a character in the font. The function `textWidth` returns the width in pixels of the given string in the given font, and `charPositions` returns the position of each character in the string.

5.3 Pixmaps and Tiles

A *pixmap* is an off-screen rectangle of colors; any drawing operation that will work on a window will also work on a pixmap. A *tile* is an immutable pixmap. More information about pixmaps and tiles can be found in Section 3.3.

```
datatype font_draw_dir = FontLeftToRight | FontRightToLeft

datatype font_prop = FontProp of {
    name : EXB.atom,        (* the name of the property *)
    value : string            (* the property value: interpret according to the *)
                              (* property. *)
  }

datatype char_info = CharInfo of {
    left_bearing : int,
    right_bearing : int,
    char_wid : int,
    ascent : int,
    descent : int,
    attributes : int
  }

exception FontPropNotFound
val fontPropertyOf : font -> EXB.atom -> string
val fontInfoOf : font -> {
        min_bounds : char_info,
        max_bounds : char_info,
        min_char : int,
        max_char : int
      }

exception NoCharInfo
val charInfoOf : font -> int -> char_info
val textWidth : font -> string -> int
val charPositions : font -> string -> int list
val textExtents : font -> string -> {
          dir : font_draw_dir,
          font_ascent : int, font_descent : int,
          overall_info : char_info
       }
val fontHt : font -> {ascent : int, descent : int}
```

Figure 5.1: Font and character metrics

5.4 Drawables

A drawable is an abstract type that collects together windows, pixmaps, and overlays (discussed in Section 7.5). The following functions are used to get the drawable of a window or pixmap:

```
val drawableOfPM : pixmap -> drawable
val drawableOfWin : window -> drawable
```

There is also a function to return the depth of a drawable:

```
val depthOfDrawable : drawable -> int
```

5.5 Drawing graphics

EXene provides a number of drawing operations on drawables; Figure 5.2 gives the signature of these operations. The semantics of the drawing operations are essentially the same as defined by **Xlib**, although

```
exception BadDrawParameter

val drawPts     : drawable -> pen -> point list -> unit
val drawPtPath  : drawable -> pen -> point list -> unit
val drawPt      : drawable -> pen -> point -> unit

val drawLines   : drawable -> pen -> point list -> unit
val drawPath    : drawable -> pen -> point list -> unit
val drawSegs    : drawable -> pen -> line list -> unit
val drawSeg     : drawable -> pen -> line -> unit

datatype shape = ComplexShape | NonconvexShape | ConvexShape
val fillPolygon : drawable -> pen -> {verts: point list, shape : shape} -> unit
val fillPath    : drawable -> pen -> {path : point list, shape : shape} -> unit

val drawRects   : drawable -> pen -> rect list -> unit
val drawRect    : drawable -> pen -> rect -> unit
val fillRects   : drawable -> pen -> rect list -> unit
val fillRect    : drawable -> pen -> rect -> unit

val drawArcs    : drawable -> pen -> arc list -> unit
val drawArc     : drawable -> pen -> arc -> unit
val fillArcs    : drawable -> pen -> arc list -> unit
val fillArc     : drawable -> pen -> arc -> unit

val drawCircle  : drawable -> pen -> {center : point, rad : int} -> unit
val fillCircle  : drawable -> pen -> {center : point, rad : int} -> unit
```

Figure 5.2: Drawing operations

the names are different. Functions that draw *paths* (e.g., `drawPtPath`) treat their `point list` argument as a list of relative coordinates. The first element specifies an absolute coordinate and each successive element specifies an offset relative to the previous coordinate. All other operations use absolute coordinates. The exception `BadDrawParameter` is raised if the argument to a drawable is invalid.

5.5.1 Area operations

To clear a rectangular region (or all) of a drawable, use the functions:

```
val clearArea : drawable -> rect -> unit
val clearDrawable : drawable -> unit
```

For a window, these functions fill with the background color; for a pixmap, they fill with 0. For `clearArea`, if the rectangle's width is zero, then the cleared rectangle is extended to the right edge of the drawable, and if the height is zero, then the cleared rectangle is extended to the bottom of the drawable. The function `clearDrawable` clears the entire drawable.

The **X**-protocol provides two operations for copying a rectangle from one drawable to another: `CopyArea` and `CopyPlane`. To further complicate things, these operations can have replies in the form of `GraphicsExpose` and `NoExpose` **X**-events. When the source drawable is a window, then it is possible that some or all of the source rectangle might be obscured; in this case, the portions of the destination that did not get updated need to be redrawn.

In **eXene**, we provide three versions of four operations, which are fully synchronous

```
exception DepthMismatch
exception BadPlane

val pixelBlt : drawable -> pen -> {
        src : draw_src, src_rect : G.rect, dst_pos : G.point
      } -> G.rect list

val bitBlt : drawable -> pen -> {
        src : draw_src, src_rect : G.rect, dst_pos : G.point
      } -> G.rect list

val planeBlt : drawable -> pen -> {
        src : draw_src, src_rect : G.rect, dst_pos : G.point, plane : int
      } -> G.rect list

val copyBlt : drawable -> pen -> {
        dst_pos : G.point, src_rect : G.rect
      } -> G.rect list
```

`pixelBlt` provides the semantics of `CopyArea`; the exception `DepthMismatch` is raised if the source and destination do not have the same depth. `planeBlt` provides the semantics of `CopyPlane`; the exception `BadPlane` is raised if the value of `plane` does not correspond to a legal bitplane in the source. `bitBlt` is the same as `planeBlt` with `plane` set to zero. The `copyBlt` function is a `pixelBlt` operation where the source and destination are the same drawable.

The source drawable may be a window, pixmap or tile, and is specified using the following datatype:

```
datatype draw_src
  = WSRC of window
  | PMSRC of pixmap
  | TSRC of tile
```

The return value is a list of rectangles in the destination, which were not updated because the corresponding source rectangles were obscured. When the source drawable is a tile or pixmap, then the return result will always be the empty list; if the source tile or pixmap is smaller than the destination rectangle, then the extra space will be filled with the zero pixel (i.e., `color0`).

The synchronous forms of the BLT operations can produce a performance bottleneck; this is why the **X**-protocol uses events instead of replies. In **CML**, however, we can provide an *asynchronous remote-procedure call* (or *promise*) interface to these operations, and thus can hide the **X**-events. To do this, we provide event-valued forms of the above operations:

```
val pixelBltEvt : drawable -> pen -> {
        src : draw_src, src_rect : G.rect, dst_pos : G.point
      } -> G.rect list CML.event

val bitBltEvt : drawable -> pen -> {
        src : draw_src, src_rect : G.rect, dst_pos : G.point
      } -> G.rect list CML.event

val planeBltEvt : drawable -> pen -> {
        src : draw_src, src_rect : G.rect, dst_pos : G.point, plane : int
      } -> G.rect list CML.event

val copyBltEvt : drawable -> pen -> {
        dst_pos : G.point, src_rect : G.rect
      } -> G.rect list CML.event
```

Note that when the source drawable is not a window, then no synchronization is necessary.

The operation

```
val tileBlt : drawable -> pen -> {src : tile, dst_pos : G.point} -> unit
```

is a `bitBlt` operation using a depth-1 tile as the source. The source rectangle is the whole tile.

5.6 Drawing text

Figure 5.3 gives the signature of the various text drawing operations provided by **eXene**. There are two styles

```
val drawString  : drawable -> pen -> font -> (point * string) -> unit
val imageString : drawable -> pen -> font -> (point * string) -> unit

datatype text
  = TEXT of (font * text_item list)
and text_item
  = TXT_FONT of (font * text_item list)
  | TXT_STR of string
  | TXT_DELTA of int

val drawText    : drawable -> pen -> (point * text) -> unit
```

Figure 5.3: Text drawing operations

of text drawing: *opaque* and *transparent*. Opaque text, provided by `imageString`, is drawn by first filling in the bounding rectangle with the background color, and then drawing the text with the foreground color. The function and fill-style of the pen are ignored, replaced in effect by `OP_Copy` and `PV_FillStyle_Solid`. In transparent text, as provided by `drawString` and `drawText`, the pixels corresponding to bits set in a character's glyph are drawn using the foreground color in the context of the other relevant pen values, while the other pixels are unmodified. The `drawText` function provides a user-level batching mechanism for drawing multiple strings of the same line with possible intervening font changes or horizontal shifts.

Chapter 6

Color

This release of **eXene** supports the most basic use of color supported by **X**: read-only access to the default colormap using either RGB values or names to specify the color. A device-independent mechanism for specifying colors is part of the X11R5 standard[TA91, SG92]. We plan to use this as the basis for future color support in **eXene**. The current color interface is defined in the `EXeneBase` structure and is given in Figure 6.1. To determine whether a screen supports color, one can use the function `displayClassOfScr` to determine the

```
datatype display_class
  = StaticGray | GrayScale | StaticColor | PseudoColor | TrueColor | DirectColor

val displayClassOfScr : screen -> display_class

datatype color_spec
  = CMS_Name of string
  | CMS_RGB of {red : int, green : int, blue : int}

val white : color_spec
val black : color_spec

val color0 : color
val color1 : color

exception BadRBG
exception NoColorCell

val colorOfScr : screen -> color_spec -> color
val blackOfScr : screen -> color
val whiteOfScr : screen -> color
```

Figure 6.1: **eXene** color operations

screen's display class. A monochrome screen, for example, will usually have the display class `StaticGray` and a depth of one. For a discussion of the display classes and **X** color model, see Chapter 7 of [Nye90b].

Colors are specified by either name or RGB value, using the `color_spec` datatype. The values `black` and `white` specify their respective colors. A `color_spec` value is mapped to an abstract `color` value using the function `colorOfScr`. The functions `blackOfScr` and `whiteOfScr` return the black and white colors for the given screen. The colors `color0` and `color1` represent the `0` and `1` pixel values, and are used to draw on pixmaps.

Chapter 7

User interaction

It is in the area of handling user input that **eXene** differs most significantly from traditional **X** libraries. Traditional **X** libraries, such as **Xlib**, **Xt**, and **CLX**, use event loops and call-back functions to simulate concurrency; in **eXene** we make the concurrency explicit.

The **X** protocol provides 33 different event messages and a complicated semantics of which events a client will receive and under what circumstances. In **eXene** we have tried to simplify the model.

7.1 Modifier buttons

X attempts to provide a portable model of input devices; part of this includes support for *modifier keys*; i.e., keys that do not have an individual meaning, but which modify the meaning of other keys. The following datatype represents the **X** modifier keys:

```
datatype modkey = ShiftKey | LockKey | ControlKey
  | Mod1Key | Mod2Key | Mod3Key | Mod4Key | Mod5Key
  | AnyModifier
```

The state of the modifier buttons (i.e., which are depressed) is represented by the type:

```
eqtype modkey_state
```

A modifier key state can be built using the function

```
val mkModState : modkey list -> modkey_state
```

which returns the state with exactly the listed buttons depressed. The standard set operations are supported on modifier states:

```
val unionMod     : (modkey_state * modkey_state) -> modkey_state
val intersectMod : (modkey_state * modkey_state) -> modkey_state
```

The following predicates test modifier states for status of individual buttons:

```
val emptyMod   : modkey_state -> bool
val shiftIsSet : modkey_state -> bool
val lockIsSet  : modkey_state -> bool
val cntrlIsSet : modkey_state -> bool
val modIsSet   : (modkey_state * int) -> bool
```

A modifier state created with `AnyModifier` is special; essentially it is the $\top$ of the set lattice.

7.2 Mouse buttons

The buttons on the mouse are represented by values of the type

```
datatype mbutton = MButton of int
```

where the integer ranges from 1 to 5. As with the modifier keys, it is often necessary to know the state of the buttons; the abstract type

```
eqtype mbutton_state
```

represents a mouse button state (i.e., a set of depressed mouse buttons). The function

```
val mkButState : mbutton list -> mbutton_state
```

returns a button state with the listed buttons depressed. Some standard set operations on mouse states are provided:

```
val unionMBut     : (mbutton_state * mbutton_state) -> mbutton_state
val intersectMBut : (mbutton_state * mbutton_state) -> mbutton_state
val invertMBut    : (mbutton_state * mbutton) -> mbutton_state
```

The functions `unionMBut` and `intersectMBut` do the obvious. The function `invertMBut` inverts the setting of the given button. There are a number of predicates on mouse states:

```
val mbutAllClr  : mbutton_state -> bool
val mbutSomeSet : mbutton_state -> bool
val mbut1IsSet  : mbutton_state -> bool
val mbut2IsSet  : mbutton_state -> bool
val mbut3IsSet  : mbutton_state -> bool
val mbut4IsSet  : mbutton_state -> bool
val mbut5IsSet  : mbutton_state -> bool
val mbutIsSet   : (mbutton_state * mbutton) -> bool
```

The predicate `mbutAllClr` is true if no buttons are depressed; `mbutSomeSet` is true if one or more buttons is set. The other predicates test the status of single button.

7.3 The window environment

eXene provides a model of input that is similar to that of [Pik89] and [Haa90]. Each window has an *environment*, consisting of three input streams (mouse, keyboard and control) and an output stream for talking to the window's parent. There are two sides to an environment: the parent sees an *output* environment for each child, and each child sees an *input* environment from its parent. Each side of this connection is represented by its own type (see Figure 7.1). The function

```
datatype in_env = InEnv of {      (* this is the window's view of its  *)
                                    (* environment *)
    k : kbd_msg addr_msg CML.event,
    m : mouse_msg addr_msg CML.event,
    ci : cmd_in addr_msg CML.event,
    co : cmd_out -> unit CML.event
  }

datatype out_env = OutEnv of {    (* this is the parent's view of one of its *)
                                    (* children's environment. *)
    k : kbd_msg addr_msg -> unit CML.event,
    m : mouse_msg addr_msg -> unit CML.event,
    ci : cmd_in addr_msg -> unit CML.event,
    co : cmd_out CML.event
  }
```

Figure 7.1: Window environment types

```
val createWinEnv : unit -> (in_env * out_env)
```

creates the channels for a window's environment and returns the input and output sides.

There are a number of operations for reconfiguring input environments. The following operations provide applicative updates of a given input stream:

```
val replaceMouse : (in_env * mouse_msg addr_msg CML.event) -> in_env
val replaceKey   : (in_env * kbd_msg addr_msg CML.event) -> in_env
val replaceCI    : (in_env * cmd_in addr_msg CML.event) -> in_env
```

Often, a window will want to ignore a given input stream, but since communication is synchronous it must still read messages to avoid locking its parent. The following operations attach null threads to the given input stream and replace the stream with another:

```
val ignoreMouse  : in_env -> in_env
val ignoreKey    : in_env -> in_env
val ignoreInput  : in_env -> in_env
val ignoreAll    : in_env -> in_env
```

The function `ignoreInput` causes both the mouse and keyboard streams to be ignored, while the function `ignoreAll` also ignores the control stream.

Sometimes a thread will intercept messages on a single stream, while passing on those on the other streams. A new environment, which has a dummy in the intercepted slot can be created by using the appropriate replace function from above and the value

```
val nullStream : 'a addr_msg CML.event
```

This stream will never produce a message; synchronizing on it will block.

Because many applications, such as menus, need to wait until the mouse has reached a stable state, **eXene** provides the function

```
val whileMouseState : (mbutton_state -> bool) -> (mbutton_state * mouse_msg CML.event)
      -> unit
```

which eats mouse events until the given predicate is satisfied. The `mbutton_state` argument is the initial mouse button state and the `mouse_msg` event value provides the stream of mouse events. The predicates described in Section 7.2 are useful for this purpose. For example, the function

```
fun downLoop (mouseEvt, mouseBut) = let
      val whileSomeSet = whileMouseState mbutSomeSet
      fun loop () = (case (msgBodyOf (sync mevt))
           of (MOUSE_Up {but, state, ...}) => if (but = mouseBut)
                then (action (); whileSomeSet (state, mevt))
                else loop ()
            | (MOUSE_LastUp _) => action()
            | _ => loop ())
      in
        loop ()
      end
```

will read mouse events from the stream represented by `mouseEvt` until the specified mouse button (`mbut`) is released. At that time, it will call the `action` function and then wait until all mouse buttons are up before returning. This idiom is useful for guaranteeing that the mouse buttons are in a stable state before handling more mouse button transitions.

7.3.1 Addressed messages and routing

The messages passed along the environment streams are *addressed* to a particular target window (e.g., the window in which a mouse click occurred). Addressed messages have the type

```
type 'a addr_msg
```

The actual contents of an addressed message can be extracted using

```
val msgBodyOf : 'a addr_msg -> 'a
```

A message address is a path through the window hierarchy. There are a number of operations designed to support routing of addressed messages:

```
datatype 'a next_win = Here of 'a | ToChild of 'a addr_msg
val stripMsg : 'a addr_msg -> 'a next_win

exception NoMatchWin
val whichWindow : (EXB.window * 'a) list -> 'b addr_msg -> 'a

val toWindow : ('a addr_msg * EXB.window) -> bool
val addrLookup : 'a EXB.window_map -> 'b addr_msg -> 'a
```

The function `stripMsg` looks at the next step in the path and returns `Here`, if the message has reached its destination, otherwise it returns `ToChild` with one address stripped from the path. The function `toWindow` compares the next window in a path with a specific window and returns true if they match. The function `whichWindow` searches a list of windows for an address match; it raises the exception `NoMatchWin` if no match is found. When a window has many children, a more efficient lookup scheme is necessary; the function `addrLookup` does an address lookup in a window hash table (see Section 3.5.2).

Because we divide the stream of input events into three separate streams, we lose the causal ordering of input events. For most applications, this isn't important, but to handle the cases in which it is important, there is a total ordering on addressed messages. The function

```
val beforeMsg : ('a addr_msg * 'a addr_msg) -> bool
```

will return true if its first argument is before its second argument in the ordering.

7.3.2 Control messages

Control messages are used by a parent window to notify its children of changes in their status and by a child window to request changes.

The control messages passed down from the top-level window are addressed messages and correspond to **X**-events. The messages currently provided are

```
datatype cmd_in
  = CI_Redraw of G.rect list
  | CI_Resize of G.rect
  | CI_ChildBirth of EXB.window
  | CI_ChildDeath of EXB.window
  | CI_OwnDeath
```

The `CI_Redraw` message is a notification that a window has been damaged; the argument is a list of damaged rectangles. The `CI_Resize` message is a notification of a change in the size of a window. The `CI_ChildBirth` and `CI_ChildDeath` messages are used to inform a window of changes in the status of its children. The system guarantees that a `CI_ChildBirth` message will be seen before any other control messages for that child, and that there will be no control messages for the child after the `CI_ChildDeath` message. In addition, corresponding synchronization messages are also passed down the mouse and keyboard streams to allow a barrier style synchronization on configuration changes (see sections 7.3.3 and 7.3.4). These messages are used in the widget message routers to automatically reconfigure the message routing in composite widgets (see Chapter 5 of [GR93]). The `CI_OwnDeath` message tells a window that it is dead (i.e., that it no longer exists on the **X**-server).

The control messages going from the child to the parent are not addressed, since they only need to go one hop. There are currently only two messages supported:

```
datatype cmd_out
  = CO_ResizeReq
  | CO_KillReq
```

These messages are requests for services that the parent window may choose to honor. The actual protocol for using these messages is left to the widget level, but it is worth noting that the bi-directional communication provided by control messages is a potential source of deadlock.

7.3.3 Keyboard messages

Keyboard messages are addressed messages that notify a window of keyboard events that occurred while the keyboard focus was in the window. The messages are

```
datatype kbd_msg
  = KEY_Press of (keysym * modkey_state)
  | KEY_Release of (keysym * modkey_state)
  | KEY_ConfigSync
```

The first two of these correspond to the pressing and releasing of a key by the user. The argument to these messages specifies the actual key pressed via a *keysym* and the state of the modifier keys. Keysyms are a portable representation of keys; Section 7.4 discusses the translation of keysyms into ASCII strings.

When certain changes occur in a window's configuration, the parent window is notified of these changes through a control message (e.g., the `CI_ChildBirth` and `CI_ChildDeath` messages in Section 7.3.2). In order for the parent to synchronize its state with the three event channels, a `KEY_ConfigSync` message is generated at the same time on its keyboard channel. A similar message is also generated on the mouse channel.

7.3.4 Mouse messages

Mouse messages are addressed messages that notify the target window of mouse events. Figure 7.2 gives the `mouse_msg` datatype. The `MOUSE_Motion` message is a notification of a change in the mouse position; its arguments specify the mouse position in the window's coordinates (`pt`) and in absolute screen coordinates (`scr_pt`). The time of the mouse motion is given as a value of the type

```
datatype time = TIME of {sec : int, usec : int}
```

The messages `MOUSE_FirstDown`,`MOUSE_Down`, `MOUSE_LastUp`, and `MOUSE_Up` are notifications of changes in the state of the mouse buttons. The arguments to these messages includes position and time information, the button being pressed and the state of all of the mouse buttons after the transition[1]. The `MOUSE_Enter` and `MOUSE_Leave` messages notify the window that the mouse has entered or left the window. The `MOUSE_ConfigSync` message plays the same role for the mouse channel that the `KEY_ConfigSync` message plays for the keyboard channel (cf. Section 7.3.3).

7.4 Keysym translation

Keysyms are a portable representation of the symbols on the key caps (see [Nye90a] for the list of keysym codes).

```
datatype keysym = KEYSYM of int | NoSymbol
```

[1]Note that this differs from the semantics of the `ButtonPress` and `ButtonRelease` X-events, which report the pre-transition state.

```
datatype mouse_msg
  = MOUSE_Motion of {
        pt : G.point,
        scr_pt : G.point,
        time : time
      }
  | MOUSE_FirstDown of {
        but : mbutton,
        pt : G.point,
        scr_pt : G.point,
        time : time
      }
  | MOUSE_LastUp of {
        but : mbutton,
        pt : G.point,
        scr_pt : G.point,
        time : time
      }
  | MOUSE_Down of {
        but : mbutton,
        pt : G.point,
        scr_pt : G.point,
        state : mbutton_state,
        time : time
      }
  | MOUSE_Up of {
        but : mbutton,
        pt : G.point,
        scr_pt : G.point,
        state : mbutton_state,
        time : time
      }
  | MOUSE_Enter
  | MOUSE_Leave
  | MOUSE_ConfigSync
```

Figure 7.2: Mouse messages

A complex algorithm is used translate a keysym and modifier state to an actual ASCII character. In **eXene**, this translation is supported by the following type and operations

```
type translation
exception KeysymNotFound
val defaultTranslation : translation
val lookupString : translation -> (keysym * modkey_state) -> string
val rebind : translation -> (keysym * modkey list * string) -> translation
```

The function `lookupString` uses a `translation` to map a keysym and modifier state (as carried by the `KEY_Press` message) to a string. For the `defaultTranslation`, this mapping returns the singleton strings for the ASCII key set. Additional or different bindings can be added using `rebind`.

7.5 Rubberbanding

Rubberbanding is a technique for supplying the user with immediate graphical feedback when specifying a geometric object. **EXene** supports rubberbanding with two separate mechanisms: *overlay windows* and *feedback drawables*.

An overlay window provides exclusive access to a window's drawing surface, so that other graphical operations do not interfere with the feedback drawing. An overlay is created by the function:

```
val createOverlay : window -> {drawable : drawable, release : unit -> unit}
```

from the `Drawing` structure. The result of `createOverlay` is the drawable to use for the feedback graphics, and a function to release the exclusive access when the rubberbanding is finished.

A feedback drawable is an unbuffered connection to the server, which can be used to provide immediate graphical response to user interaction. The function

```
val feedback : drawable -> drawable
```

is used to create a feedback drawable from an existing drawable.[2]

A common example of rubberbanding is sweeping out a rectangle to specify the size of a window. Figure 7.3 gives the code for such an interaction. When the user presses the mouse button, the current cursor position is fixed as an anchor point. As the mouse is moved, feedback in the form of a rectangle with one corner at the fixed anchor point and the opposite corner at the current mouse position. When the mouse button is released, the window is created using the current rectangle as its size and shape.

[2]Note that using feedback drawable reduces performance, because of extra system-call overhead.

```
fun getRect (win, anchorPt, mevt) = let
      val {release, drawable} = createOverlay win
      val draw = drawRect (feedback drawable)
                   (newPen [PV_Function OP_Xor, PV_Foreground color1])
      val {sz=sz as SIZE{wid, ht}, ...} = geomOfWin win
      fun clip (PT{x, y}) = PT{
              x = if x < 0 then 0 else if x >= wid then (wid-1) else x,
              y = if y < 0 then 0 else if y >= ht then (ht-1) else y
            }
      fun ptsToRect (PT{x, y}, PT{x=x', y=y'}) = let
            fun minmax (a : int, b) = if a <= b then (a, b-a) else (b, a-b)
            val (ox, sx) = minmax(x, x')
            val (oy, sy) = minmax(y, y')
            in
              RECT{x=ox, y=oy, wid=sx, ht=sy}
            end
        fun doRect () = let
              val initRect = ptsToRect (anchorPt, anchorPt)
              fun loopRect (r, p) = (case (msgBodyOf (sync mevt))
                     of MOUSE_LastUp{but, pt, ...} => (draw r; release(); r)
                      | MOUSE_Motion{pt, ...} => update (r, p, clip pt)
                      | _ => loopRect (r,p)
                    (* end case *))
              and update (oldRect, oldPt, newPt) =
                    if newPt = oldPt
                      then loopRect (oldRect, oldPt)
                      else let
                        val newRect = ptsToRect (anchorPt, newPt)
                        in
                          draw oldRect; draw newRect;
                          loopRect (newRect, newPt)
                        end
              in
                draw initRect;
                loopRect (initRect, anchorPt)
              end (* doRect *)
        in
          doRect ()
        end (* getRect *)
```

Figure 7.3: Code to get a rectangle from the user

Chapter 8

Inter-client communication

The **X** standard includes a complex set of conventions for inter-client communication[Ros89]. While **eXene** does not currently support these conventions, we consider such support vital to making **eXene** useful for building applications. A future release will support (at the minimum) both selections and cut buffers.

8.1 Atoms

Atoms are unique identifiers corresponding to a string name; the **X**-server maintains the mapping between the string names and atoms. The following operations on atoms are provided:

```
val internAtom : display -> string -> atom
val lookupAtom : display -> string -> atom option
val nameOfAtom : display -> atom -> string
```

The `internAtom` function maps a string name to an atom, creating a new atom if necessary. The function `lookupAtom` also maps a string to an atom, but, if the atom does not already exist, then `NONE` is returned. A client can get the string name associated with an atom by calling `nameOfAtom`.

X defines a set of standard atoms; these atoms are defined in the `ICCC` structure. A standard atom *name* is represented by the identifier `atom_`*name*. For example, the `PRIMARY` atom is represented by `atom_PRIMARY`.

8.2 Selections

Selections are currently unsupported.

8.3 Cut buffers

Cut buffers are currently unsupported.

8.4 Window hints

The various window manager and size hints used by the `setWMProperties` function (described in Chapter 4) are defined in the `ICCC` structure.

```
datatype size_hints
  = HINT_USPosition
  | HINT_PPosition of G.point       (* obsolete in X11R4 *)
  | HINT_USSize
  | HINT_PSize of G.size            (* obsolete in X11R4 *)
  | HINT_PMinSize of G.size
  | HINT_PMaxSize of G.size
  | HINT_PResizeInc of G.size
  | HINT_PAspect of {min : int * int, max : int * int}
  | HINT_PBaseSize of G.size
  | HINT_PWinGravity of EXB.gravity
```

```
datatype wm_hints
   = HINT_Input of bool
   | HINT_WithdrawnState
   | HINT_NormalState
   | HINT_IconicState
   | HINT_IconTile of EXB.tile
   | HINT_IconPixmap of EXB.pixmap
   | HINT_IconWindow of EXB.window
   | HINT_IconMask of EXB.pixmap
   | HINT_IconPosition of G.point
   | HINT_WindowGroup of EXB.window
```

Chapter 9

Graphics utilities

In addition to the library, there is a collection of graphics utilities that are neither part of the library or widget set. We describe these here.

9.1 Ellipses

The `Ellipse` structure provides code for drawing rotated ellipses. It has the following signature:

```
signature ELLIPSE =
  sig
    structure G : GEOMETRY

    exception BadAxis

    val ellipse : (G.point * int * int * real) -> G.point list

  end; (* ELLIPSE *)
```

The application `ellipse` $(pt,\ a,\ b,\ \phi)$ produces a list of points describing the ellipse defined by the following equation:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

translated to the the point *pt* and rotated by ϕ radians in a counterclockwise direction. The function `ellipse` raises the exception `BadAxis`, if either *a* or *b* is less than zero. If either *a* or *b* is zero, then the empty list is returned. The result of applying `ellipse` can be drawn using the `drawLines` function.

9.2 Splines

The `Spline` structure provides routines for drawing Bézier splines. The signature of this structure is:

```
signature SPLINE =
  sig

    structure G : GEOMETRY

    val curve : (G.point * G.point * G.point * G.point) -> G.point list

    val simpleBSpline : G.point list -> G.point list
    val bSpline       : G.point list -> G.point list
    val closedBSpline : G.point list -> G.point list

  end (* SPLINE *)
```

The meanings of the operations are:

`curve` (p_0, p_1, p_2, p_2)
: Return a list of points corresponding to a Bézier cubic section, starting at p_0, ending at p_3, with p_1, p_2 as control points.

`simpleBSpline` $[p_0, \ldots, p_n]$
: Compute a simple B-spline with the given control points.

`bSpline` $[p_0, \ldots, p_n]$
: This is defined as

$$\texttt{simpleBSpline}\ ([p_1,\ p_1,\ p_1,\ \ldots,\ p_n,\ p_n,\ p_n])$$

The replication of p_1 and p_n constrains the resultant spline to connect p_1 and p_n.

`closedBSpline` $[p_0, p_1, p_2, \ldots, p_n]$
: Compute a closed B-spline. This is defined as

$$\texttt{simpleBSpline}\ [p_0, p_1, p_2, \ldots, p_n, p_0, p_1, p_2]$$

Note that the first and last points of the result are the same.

9.3 Rounded rectangles

Two utility functions are provided for drawing rectangles with rounded corners. These can be found in the structure `RoundedRect`, which has the following signature:

```
signature ROUNDED_RECT =
  sig
    structure G : GEOMETRY

    val drawRoundedRect : Drawing.drawable -> Drawing.pen
          -> {rect : G.rect, c_wid : int, c_ht : int} -> unit

    val fillRoundedRect : Drawing.drawable -> Drawing.pen
          -> {rect : G.rect, c_wid : int, c_ht : int} -> unit

  end (* ROUNDED_RECT *)
```

The function `drawRoundedRect` draws the outline of a rectangle, while the function `fillRoundedRect` draws a filled rectangle. The arguments `c_wid` and `c_ht` specify the width and height of the rounded corners.

9.4 Bitmap I/O

Chapter 10

Unsupported X-windows features

X-windows is a large and complicated system and there are many aspects of the **X**-protocol and **Xlib** that **eXene** does not support. Some of these are features that we plan to support in the near future, others are unlikely to ever be supported. The following is a partial list of the currently unsupported features and our plans with respect to support:

Full color support: This release only supports the default static colormap provided for each screen (see chapter 6).

X resources: We have an implementation of the **X** resource database, but it is currently not included in the release. We will include it once we understand how to use resources to configure widgets.

ICCC: There is only limited support for the **X** *Inter-Client Communication Conventions*. As we add more pieces to **eXene** we expect this support to fill out into full compliance with the standard[Ros89].

Extensions: The **X**-window specification is designed to be extensible and there are a fair number of existing extensions, such as the MIT *Shape* extension and the Adobe **Display Postscript** extension. This release of **eXene** does not support extensions, but we are planning on adding such support. A future release will contain support for binding new extensions as well as a sample implementation (probably for the *Shape* extension).

Window-manager support: A large fraction of the **X**-protocol's function is designed to support the implementation of window managers as clients. We have made no attempt to support these operations in **eXene**, and it is unlikely that we ever will. Note, however, that the protocol translation routines (in the files `protocol/xrequest.sml` and `protocol/xreply.sml`) do implement those features.

Bibliography

[GR91] Gansner, E. R. and J. H. Reppy. eXene. In *Proceedings of the 1991 CMU Workshop on Standard ML*, Carnegie Mellon University, September 1991.

[GR93] Gansner, E. R. and J. H. Reppy. *The eXene Widgets Manual*. AT&T Bell Laboratories, Murray Hill, N.J. 07974, February 1993. Included in the eXene distribution.

[Haa90] Haahr, D. Montage: Breaking windows into small pieces. In *USENIX Summer Conference*, June 1990, pp. 289–297.

[MIT89] MIT X Consortium Standard. *Logical Font Description Conventions (version 1.3)*, 1989.

[Nye90a] Nye, A. *X Protocol Reference Manual*, vol. 0. O'Reilly & Associates, Inc., 1990.

[Nye90b] Nye, A. *Xlib Programming Manual*, vol. 1. O'Reilly & Associates, Inc., 1990.

[Nye90c] Nye, A. (ed.). *Xlib Reference Manual*, vol. 2. O'Reilly & Associates, Inc., 1990.

[Pau91] Paulson, L. C. *ML for the Working Programmer*. Cambridge University Press, New York, N.Y., 1991.

[Pik89] Pike, R. A concurrent window system. *Computing Systems*, **2**(2), 1989, pp. 133–153.

[Rep90] Reppy, J. H. *Concurrent programming with events – The Concurrent ML manual*. Department of Computer Science, Cornell University, Ithaca, N.Y., November 1990. (Last revised February 1993).

[Rep91a] Reppy, J. H. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991, pp. 293–305.

[Rep91b] Reppy, J. H. An operational semantics of first-class synchronous operations. *Technical Report TR 91-1232*, Department of Computer Science, Cornell University, August 1991.

[Rep92] Reppy, J. H. *Higher-order concurrency*. Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, NY, January 1992. Available as Technical Report TR 92-1285.

[Ros89] Rosenthal, D. *Inter-Client Conventions Manual (version 1.0)*. MIT X Consortium Standard, 1989.

[SG92] Scheifler, R. W. and J. Gettys. *The X Window System*. Digital Press, 3rd edition, 1992.

[TA91] Tabayoyan, A. and C. Adams. *X Color Management System – An Xlib Enhancement (Public Review Draft)*, April 1991.