User Preference Extensions to eXene

Dusty deBoer
KSU CIS 705 Final Project

March 8, 2005

X applications are generally user customizable. This allows the user to customize applications to fit his or her needs, and to make make the applications more usable. For example, a user may wish to set all applications to use a large font, or set all applications to use a common background to fit a desktop theme, or set one particular application to use a unique background so that it can easily be distinguished from other programs. Applications should provide a way for users to easily and uniformly apply these preferences.

In Xlib released with version 11 of the X Window system, there are several standard ways that a user may set application preferences. Application preferences may be stored in a preferences file in the format "`application[.subcomponent...].attribute: attributevalue`". These preferences may be loaded by an application into a "resource database" by the routine `XrmGetFileDatabase`.

However, if a user were displaying several applications from several different hosts on the same X-server, the user would be forced to ensure that the preferences file on each host was the same to ensure that the same preferences were applied to all applications. To address this issue, there is the `xrdb` utility that reads a preferences file and loads the preferences (as strings) into a property of the X-server. An application then may read preferences directly from the X-server.

In addition, the user may wish to customize an application at run time. In Xlib, an application may use the routine `XrmParseCommand` that, given a

specification of the command line options that the application wishes to recognize, returns a resource database that may be queried for user preferences.

Finally, an application must have a method for combining resource databases, giving priority to the preferences of one database over another. For example, an application may have default preferences programmed into the application, preferences from a application preferences file, preferences loaded from the X-server, and run time preferences loaded from the command line. These preferences would usually be combined in an order such that run time preferences have priority over X-server (xrdb) preferences, and X-server preferences have priority over application defaults. Xlib provides the routine `XrmMergeDatabases(sourcedb,targetdb)` to merge preferences of a source database into a target database, with the preferences of the source database taking priority over those of the target.

1 eXene Styles

Currently, eXene provides (perhaps limited) methods for customizing widgets. Generally, widgets take a "view" as an argument to the widget constructor, containing information about user preferences. A view is a pair consisting of a "style" and a "styleView", where a style is a database of attribute/value pairs (very similar in utility to the Xlib resource database) and a styleView is a search key into that style, such as the name of the application. The widget may then use the style, combined with the styleView, to search for values of attributes it wishes to use, perhaps under the application name given in the styleView.

There is a function in eXene, `Widget.styleFromStrings`, that takes a list of strings and creates a style. This is similar in use to the routines in Xlib that load strings from a preferences file into a resource database. However, eXene is missing the ability to load preferences stored by xrdb in the X-server, the ability to parse options and resources from command line arguments, and the ability to merge resource databases (styles) together.

While it would certainly not be appropriate to expect eXene to emulate

Xlib in every way, the Xlib resource management paradigm is both useful and widely used. In this project, I shall therefore seek to implement in eXene methods for parsing command line arguments, methods for loading preferences stored on the X-server, and methods for merging eXene styles similar to Xlib where appropriate.

2 Parsing Command Line Options

Xlib provides the routine `XrmParseCommand` that, given a specification of the command line options that the application wishes to recognize, parses command line arguments into a resource database. While I think it is useful to emulate the option specification used by Xlib, so that eXene applications might recognize the same types of command line options, I also recognize that eXene does not have any other command line parsing functions available. Therefore, a command line argument parsing function may be useful not only for obtaining user preferences, but for obtaining data for processing by the application. For example, a application may wish to accept requests to set the background color by "`-background blue`", but may also wish to obtain the value of a filename by "`-filename foo`". There is really no need for the filename to be recognized as a user preference for the application and all of its widgets to view. Let us therefore distinguish between two types of options - "resource" options whose purpose is to be loaded into an eXene style, and "named" options, whose purpose is to be used for application processing.

It is the application's responsibility to set up a command line option specification table. This table, `Styles.optSpec`, shall be a:
`(optName * argName * optKind * Attrs.attr_type) list` .

The option name, `Styles.optName`, shall be of either `Styles.OPT_NAMED of string` or `Styles.OPT_RESSPEC of string`, where the former is a named option and the latter is a resource option. The string given for a named option shall be any string of the application's choosing, simply used to identify the option when retrieving a value later, such as `OPT_NAMED("filename")`. The string given for a resource option is a resource name, such as `OPT_RESSPEC("*background")` or `OPT_RESSPEC("appname.background")`.

The argument name, `Styles.argName`, shall be a string that is valid to be used on the command line to specify the setting of this option. For example, `"-background"` or `"--bg"` or `"/bg="` might be used as argument name values.

The option "kind" (term taken from Xlib), `Styles.optKind`, shall be of one of the following:

`OPT_NOARG of string`	Similar to Xlib's `XrmOptionNoArg`. Option will assume the value of the string given if set.
`OPT_ISARG`	Similar to Xlib's `XrmOptionIsArg`. Option will assume the value of the argument name itself if set.
`OPT_STICKYARG`	Similar to Xlib's `XrmOptionStickyArg`. Option will assume the value of the substring following the argument name if set. For example, if `"-bg="` is the argument name, and "`-bg=blue`" is given on the command line, the value of the option will be "`blue`".
`OPT_SEPARG`	Similar to Xlib's `XrmOptionSepArg`. Option will assume the value of the command line argument immediately following if set.
`OPT_RESARG`	Similar to Xlib's `XrmOptionResArg`. A resource specification argument should follow on the command line. For example, if `"-res"` is the argument name, and "`-res "*background:blue"`" is given on the command line, there will be two option values created in the option db returned: one with a name of `"-res"`, of either named or resource specification name, and a value of "`*background:blue`"; and the other of name `OPT_RESSPEC("*background")` and value of "`*background:blue`".
`OPT_SKIPARG`	Similar to Xlib's `XrmOptionSkipArg`. Skip the next argument given on the command line; do not attempt to match it to any option nor to assign it as a value to any option.
`OPT_SKIPLINE`	Similar to Xlib's `XrmOptionSkipLine`. Ignore all following command line arguments given.

Finally, the option type, of type `Attrs.attr_type`, is the type of the value

to be returned.

An example option specification is given here, for an application wishing to find values for named options of “help”, “flag”, “pi”, and “cmd”; and resource options of “*background”, “*foreground”, and “*borderWidth”. It also allows the user to skip an argument or all following arguments on the command line with “-skip” or “-ignore”. In addition, note that the option “help” may be toggled on with “-help” or off with “-nohelp”.

```
structure S = Styles
structure A = Attrs
val optSpec =
 [(S.OPT_NAMED("help"), "-help",   S.OPT_NOARG("on"),    A.AT_Bool),
  (S.OPT_NAMED("help"), "-nohelp", S.OPT_NOARG("off"),   A.AT_Bool),
  (S.OPT_NAMED("flag"), "FLAG",    S.OPT_ISARG,          A.AT_Str),
  (S.OPT_NAMED("pi"),   "-pi=",    S.OPT_STICKYARG,      A.AT_Real),
  (S.OPT_NAMED("cmd"),  "-cmd",    S.OPT_SEPARG,         A.AT_Str),
  (S.OPT_NAMED("res"),  "-res",    S.OPT_RESARG,         A.AT_Str),
  (S.OPT_NAMED("skip"), "-skip",   S.OPT_SKIPARG,        A.AT_Str),
  (S.OPT_NAMED("ign"),  "-ignore", S.OPT_SKIPLINE,       A.AT_Str),
  (S.OPT_RESSPEC("*background"), "-bg",    S.OPT_SEPARG, A.AT_Str),
  (S.OPT_RESSPEC("*foreground"), "-fg",    S.OPT_SEPARG, A.AT_Str),
  (S.OPT_RESSPEC("*borderWidth"),"-border",S.OPT_SEPARG, A.AT_Str)]
```

```
Styles.parseCommand:  Styles.ctxt * Styles.optSpec -> string
list -> Styles.optDb * string list.
```

Command line arguments may be parsed into an “option database”, `Styles.optDb`, with the function `parseCommand`. ParseCommand takes a styles context (basically, a screen, used in converting attribute values), an option specification, and a list of command line arguments as strings, and returns an option database and the list of strings that were not recognized as option arguments. Note that any unique prefix of an argument name will be recognized as a valid command line option. Also note that the function will not throw any exceptions upon encountering an unknown command line argument, it will simply add that string to the list of unrecognized arguments. In addition, the position of the unrecognized arguments is not noted in the list returned; this may be a future

enhancement, as the position of these arguments may be important to some applications.

```
Styles.findNamedOpt:  Styles.optDb -> Styles.optName ->
Attrs.attr_value list
```

Named command line option values may be retrieved from an option database using the `findNamedOpt` function. The list of attribute values returned are the list of all values specified for the named option on the command line, in reverse order. For example, if "`-pi=3.14 -pi=3.15 -pi=3.16`" were given on the command line, and an option db "`optDb`" was returned from parsing these arguments based on the option specification given above, `(Styles.findNamedOpt optDb (Styles.OPT_NAMED("pi")))` would return a list `[A.AV_Real(3.16),A.AV_Real(3.15),A.AV_Real(3.14)]`. In this way, the application could choose to let the last option given have priority over the others, in which case it would choose the head of the list. Or, the application could choose to use all of the option values given, and process the whole list.

Note that only "named" options may be returned by `findNamedOpt`; if a resource option is searched on, an empty list will be returned.

```
Styles.styleFromOptDb:  Styles.ctxt * Styles.optDb ->
Styles.style
```

Finally, resource options and their values in an option database may be converted to an eXene style. The function `styleFromOptDb` takes a context and an option database and returns a style.

3 Loading X-Server (xrdb) Preferences

```
val Display.rootWinOfScr:  Display.screen -> XProtTypes.win_id
```

When xrdb is run, it loads preferences (as strings) into the `XA_RESOURCE_MANAGER` property of the root window of the X-Server. The function `Display.rootWinOfScr` returns the root window id of a screen.

```
val ICCC.xrdbOfScr :  EXB.screen -> string list
```

The function `ICCC.xrdbOfScr` uses the `Display.rootWinOfScr` to retrieve the resource properties stored by xrdb in the `XA_RESOURCE_MANAGER` property and convert them into strings. Note that managing user preference strings has nothing really to do with ICCC; it was located in this structure simply because `ICCC` had the necessary data structures available, in addition to the get-property functions. In contrast, `Styles` is not currently aware of `Display` and `ICCC`; it would be much harder to locate this function there. In the future, this function should probably be relocated (before being used by many users).

4 Merging Styles

When an application obtains eXene styles from several sources (such as command line arguments and X-server preferences) it is necessary to combine or "merge" these styles together. It is also necessary to perform this merge in such a way that certain preferences have priority over others. For example, an application may wish to allow run-time preferences in a style obtained from the command line to have priority over preferences from a style obtained from X-server preferences, which may have priority over preferences from an application default style.

```
val Styles.mergeStyles :  Styles.style * Styles.style ->
Styles.style
```

The function `mergeStyles` takes two eXene style arguments, the first the "source" style and the second the "target" style, and merges them into one "merged" style. The "merged" style should consist of the same resource specifications that would exist in the "target" style if all resource specifications of the "source" were inserted into the "target." That is, in particular, a tight binding of a particular resource specification in targetStyle would not be overwritten by a loose binding of the same specification in sourceStyle.

The behavior of this should be similar to `XrmMergeDatabases(db1,db2)` of Xlib; in particular, resources specified in db1 should override those in db2.

5 Implementation Details

The following functions and types were added to the `Widget` structure, to make it easier for the user of eXene widgets to use the user preference extensions. In addition, the `styleFromXRDB` function is a utility function that simplifies creating a style from xrdb properties - it obtains the properties stored by xrdb in the X-server, and calls `styleFromStrings` on these strings to create a style.

```
type Widget.optName
type Widget.argName
type Widget.optKind
type Widget.optSpec
type Widget.optDb
type Widget.attr_value
val Widget.mergeStyles :
        Widget.style * Widget.style -> Widget.style
val Widget.styleFromXRDB :
        Widget.root -> Widget.style
val Widget.parseCommand :
        Widget.root * Widget.optSpec -> Widget.string list
            -> Widget.optDb * string list
val Widget.findNamedOpt :
        Widget.optDb -> Widget.optName -> Attrs.attr_value list
val Widget.styleFromOptDb :
        Widget.root * Widget.optDb -> Widget.style
```

The following files from the eXene source tree were modified. The line numbers of the modifications are listed, or a search for "ddeboer" in the eXene tree will also list these files and lines.

```
Command line parsing; Merging styles:
  eXene/styles/styles-func.sml: 312-314, 342-344, 448-718

Retrieve xrdb properties:
  eXene/lib/window/display.sml: 181-190
  eXene/lib/user/iccc-sig.sml: 161-174
```

```
  eXene/lib/iccc/property-sig.sml: 92-104
  eXene/lib/iccc/property.sml: 206-227

Adding extensions API to Widgets:
  eXene/widgets/basics/root.sml: 37-90, 165-196
```

6 Further Work

The signatures (interfaces) for the three extensions implemented in this project should be further refined and documented, and the structures implementing them should be relocated (in some cases) into more natural locations in the eXene source. The merge function should be better defined, in addition to perhaps being rewritten to eliminate possible errors. The unrecognized arguments returned from command line parsing should perhaps be marked according to position among the command line arguments. Finally, eXene in general is very much in need of updated documentation and cleanup (for example, some abandoned code in Styles used deprecated CML functions like "CML.accept").

7 References

Nye, A. *Xlib Programming Manual, Volume One.* O'Reilly & Associates, 1990.

Nye, A. *Xlib Programming Manual, Volume Two.* O'Reilly & Associates, 1988.

Reppy, J. "eXene/styles/Notes" of eXene source distribution.

Reppy, J.H., and E.R. Gansner. "The eXene Library Manual", AT&T Bell Laboratories, 1993.