Abstract

McCLIM is a free implementation of the Common Lisp Interface Manager, or CLIM, specification. In this paper we review the distinguishing features of CLIM, describe the McCLIM implementation, recount some of the history of the McCLIM effort, give a status report, and contemplate future directions.

McCLIM is a portable implementation of the Common Lisp Interface Manager, or CLIM, specification[9] released under the Lesser GNU Public License (LGPL). CLIM was originally conceived as a library for bringing advanced features of the Symbolics Genera system[3], such as presentations, context sensitive input, sophisticated command processing, and interactive help, to Common Lisp implementations running on stock hardware. The CLIM 2.0 specification added “look-and-feel” support so that CLIM applications could adopt the appearance and behavior of native applications without source code changes.

Several Lisp vendors formed a consortium in the late 80s to share code in a common CLIM implementation, but for a variety of reasons the visibility of CLIM has been limited in the Common Lisp community. The McCLIM project has created an implementation of CLIM that, in the summer of 2002, is almost feature complete. Initially created by merging several developers’ individual efforts, McCLIM is being used by several programs including a music editor, a web browser, and an IRC client (see Figure 1). Some non-graphic parts of CLIM have been adopted into the IMHO web server[2]. In this paper we review the distinguishing features of CLIM, describe the McCLIM implementation, recount some of the history of the McCLIM effort, give a status report, and contemplate future directions for McCLIM.

1 CLIM Features and Concepts

CLIM abstracts the input and output features of window systems in order to separate the applications programmer, and the specification, from any particular system. Graphics are rendered on sheets, which correspond to most systems’
Figure 1: Jochen Schmidt’s weird-irc IRC client in McCLIM
notion of a window. The root window is represented by a *graft*; visible sheets are
direct or indirect children of the graft and are said to be *grafted*. Input events
happen in sheets or are associated with sheets. The concrete window system
window or “drawable” associated with a sheet is called a *mirror*. CLIM defines
generic functions for drawing graphics and text into sheets which generally map
directly into the host window system’s primitives.

For screen layout CLIM defines a rich set of *pane* classes, interface *gadgets*
such as buttons and text fields, methods for composing these panes and gadgets
into complete interfaces, and support for user input. CLIM is very tightly
integrated with Common Lisp streams. Graphics operations such as drawing
lines are performed on an output stream, but that output stream is also a valid
argument to standard Lisp output functions such as *format*. Input streams are
generalized to return mouse and button gestures as well as keyboard input.

Many GUI toolkits have a prettier appearance than CLIM’s default appear-
ance. What distinguishes CLIM are features that are built on top of conventional
toolkit features. CLIM output streams support recording – similar to display
lists in OpenGL[7] – so that a program does not need to maintain output state.
A common example would be a program that writes its output to a continually
scrolling window in the style of a conventional terminal application. For input,
CLIM extends the traditional Lisp read - eval - print loop into a command loop
which reads commands, executes them, and refreshes the application display[10].
A *command* consists of a function and typed arguments that will be supplied
to it. There are several ways in which a user can specify commands and their
arguments. She can choose a simple command from a menu or type the com-
mand in an interactor pane, which supports a rich set of editing, completion,
and help features.

The user can also enter arguments by clicking on graphic objects on the
screen. This is supported by a fundamental feature of CLIM, *presentations*[4].
Program objects, instead of only being printed or represented by screen graph-
ics, are *presented*, which associates the screen output with both the object and
a presentation type (which can be different from the object’s actual type). In
the CLIM command loop one or more input contexts, which expect input of a
certain type, are active. Any presentation whose type is a subtype of the input
context type can then be clicked on to provide input. Presentation translators
enable a presentation to satisfy an input context of a completely different type;
in fact, a translator can translate an object to a command, so that an entire
command can be entered by clicking on a presentation. For example, an ob-
ject could be selected for some purpose by clicking on it if there’s a translator
from the presentation type associated with the object to the “select” command.
Translators are also associated with mouse buttons and combinations of modifier
keys.

CLIM provides interactive feedback during the input process. If the user
moves the mouse pointer over an applicable presentation it will be highlighted,
indicating that it will satisfy one of the current input contexts. A mouse docu-
mentation pane provides real-time information on the effects of clicking a mouse
button with the current modifier keys; this documentation can be unique to
each presentation. CLIM also provides other application-building features such as table, text and graph formatting, and an application definition facility which supports both definition of application commands and layout of application screen real estate.

CLIM excels in problem spaces where the user interface is highly dynamic and not specifiable by a static GUI building tool. It also succeeds where the program interface is complex due to the nature of the problem domain. CLIM remains interesting and unique in spite of competition from younger, more attractive toolkits.

2 Implementation of McCLIM

CLIM is specified in terms of protocols defined on CLOS generic functions and classes[6]. Generally the classes are protocol classes, which are not intended to be instantiated but are required superclasses of objects that participate in the protocols. Each protocol class has a corresponding concrete class which can be instantiated and usefully subclassed. By convention a concrete class has the name of its protocol class with standard- prepended. For example, the protocol class application-frame defines the behavior of a CLIM application’s top level window; standard-application-frame or a subclass of it is instantiated when starting a CLIM application.

CLIM follows a common Lisp pattern of specifying functions in pairs: a regular function with few required arguments and many keyword arguments, and a generic function in which some of those keyword arguments are required arguments. The regular function is called by application programmers for whom the keyword argument defaults are convenient. It in turn calls the generic function, which can be specialized on arguments that were keyword arguments of the regular function. In our description of the McCLIM implementation we will generally refer to the regular function and not make a distinction between it and the generic function unless needed.

2.1 Language and Host Requirements for McCLIM

McCLIM is written in ANSI Common Lisp[1] and uses common extensions, specifically Gray streams[5] and the Metaobject Protocol[8]. Threads are optional but are required to support multiple applications in a single Lisp image. McCLIM currently runs in CMU Common Lisp (CMUCL), Steel Bank Common Lisp (SBCL), Allegro Common Lisp, and OpenMCL.

McCLIM has backends for interfacing to several different graphics systems. The most robust interface is to the X Window System[12] which uses the CLX Common Lisp bindings. There is also an experimental interface to OpenGL[7] that uses a foreign function interface to directly make OpenGL calls for output and X Window System calls for input. Additionally there is a backend for outputting PostScript from McCLIM graphics.
2.2 Graphics Substrate

CLIM defines a powerful set of operators on regions. This includes boolean operations on shapes such as lines, rectangles, collections of rectangles, and general polygons. McCLIM fully implements this region protocol including translation of complex regions to rectangle strips, which is required for most window system clipping operations. McCLIM uses the region protocol in many other parts of the system, particularly for calculating rectangle – rectangle overlap. Regions control both the clipping and transformation of graphics in sheets.

CLIM makes it easy to allow sheets with or without mirrors to be the targets for graphics operations. A sheet without a direct mirror will simply redirect such operations to the mirror of its parent, with a combination of the transformations and regions of itself and its parent.

The CLIM specification is a bit unclear about the terminology here, as sometimes a mirrored sheet is considered to be sheet that can have a direct mirror, sometimes one that currently has one. Given that ungrafted sheet hierarchies can move to different ports, potentially with different underlying windowing systems, the only reasonable interpretation seems to be the first one. Thus, the only use for sheets that are not mirrored would be for sheets that should never have a direct mirror.

In McCLIM, we decided to implement mirrored sheets by storing a mirror in all sheets (whether mirrored or not, and whether they have a direct mirror or not). All graphics operations use this mirror for graphics output using either the device region and device transformation or the native region and native transformation. This implementation gives a uniform and fast implementation of all graphics operations. In addition, computing these regions and transformations from those of the parent becomes straightforward in our implementation.

Associated with sheets are mediums which hold graphics state such as current foreground and background color. CLIM graphics generic functions such as draw-line and draw-text dispatch on the medium, render to the sheet’s mirror, and use the sheet transformation (and possibly a medium transformation). Most of these operations also have “trampoline” methods, defined on sheets, that immediately dispatch to the medium graphics function.

2.3 Output and Input Streams

Extended output streams are an abstraction layered on top of sheets that support the semantics of Common Lisp streams for character output as well as CLIM graphics operations. CLIM output streams are a subclass of sheet, so the sheet graphics functions discussed above “just work.” A text cursor is maintained which is updated when the Common Lisp printing functions – via Gray Streams interfaces such as stream-write-char – print to an output stream. terpri ultimately results in a simple movement of the text cursor.

Extended input streams add window system capabilities to Common Lisp input streams. The basic reading function is read-gesture, which is similar to the Common Lisp function read-char but also implements peeking and timeout.
This function can return pointer gesture objects and keyboard gestures with modifier keys as well as characters. McCLIM implements the generic functions of the Gray streams protocol such as `stream-read-char` and `stream-read-line` by calling this function and filtering out non-character gestures. Thus a program can call any Lisp input function, such as `read`, on an extended input stream and get meaningful results.

Each CLIM sheet has an associated event queue. Events from the host window system are translated to CLIM events and delivered to the queue of the sheet where the event occurred. Extended streams in CLIM are subclasses of `sheet`, so McCLIM can use the sheet event queue as the stream buffer. `read-gesture` translates keyboard events to characters when possible as it reads them out of the queue. In order to implement “unreading” McCLIM allows characters to be pushed back on the event queue.

While Common Lisp defines functions that block waiting for input from streams, neither it nor the Gray streams extensions provide any functionality for implementing user-defined streams that block. CLIM implements blocking via two strategies. In multi-threaded implementations, a single thread gets input from the host graphics system and distributes it to waiting applications via queues. Programs wait for input in the queues using the waiting primitives of the thread implementation.

In a single-threaded implementation the backend’s event mechanism is used to block the single thread of execution; when an event is available it is put in the stream’s event queue and then the same code is invoked that’s used in the multi-threaded implementation.

`read-gesture` also takes `wait test` and `input handler` parameters which, if supplied, filter all device events received by an application. The defaults come from special variables and are rarely supplied in application code, so McCLIM can bind these filters dynamically and nest them for its own purposes. The wait tester is invoked when input becomes available. If it returns `nil` waiting will continue; otherwise the input handler is invoked. The input handler can either consume the available input or leave it for `read-gesture` to return.

### 2.4 Output Recording

CLIM manages the redisplay of output streams for the user by maintaining output records. An output record can contain other output records or information needed to recreate primitive drawing operations such as the operation, current graphics state, or the string associated with drawing text. A stream has a current output record which records all graphics and text operations while recording is enabled. A new output record can be established with the `with-new-output-record` macro; it will be a child of the current output record and can be a subclass of `output-record`. In general, then, a stream’s output records form a tree rooted at a top level record.

In McCLIM primitive operations are captured via around methods. The bounding box of an output record is incrementally updated as output is added to it, as is the bounding box of the record’s parent (and so forth up the tree).
```lisp
(define-presentation-type integer (optional low high)
  :options ((base 10) radix)
  :inherit-from '((rational ,low ,high) :base ,base :radix ,radix))

(define-presentation-method presentation-typep (object (type integer))
  (and (integerp object)
       (or (eq low '*)
           (<= low object))
       (or (eq high '*)
           (<= object high))))

(defun-presentation-type-of (object integer)
  'integer)

Figure 2: Example of presentation type definition

Generic functions are provided that map over all the output records that in-
tersect a position and a geometric region. The latter is obviously needed to
support redisplay on exposure events and such; the former provides powerful
support for mouse input.

2.5 Presentation Types and Presentation Methods

The type of presentation is not the type of its object. In fact, it is not a Lisp
type at all; it is a presentation type. Presentation types form a parallel lattice
to the Common Lisp type system and extend the capabilities of user-defined
Lisp types. They are defined via `define-presentation-type', a deftype - like
macro. Subtype relationships are implied by the type specification “macroex-
pansion”; additionally the programmer can write predicate methods for type
membership and subtype relations within a type. In addition to parameters,
presentation types may also have options which affect the visual representation
of presentations. Many builtin Lisp types have presentation type equivalents of
the same name. Figure 2 shows the definition of the integer presentation type
that is built into McCLIM.

Although they are represented as lists, presentation types have many char-
acteristics of CLOS objects. Their parameters and options are similar to class
slots, and they have an inheritance relation with their supertypes. However,
parameters and options are not inherited from supertypes – they parameterize
the supertypes and they may be arbitrarily transformed1 between the subtype
and supertype.

To support dispatching on a single presentation type argument, CLIM pro-
vides presentation generic functions and presentation methods that are similar
to their CLOS equivalents – for example, method combination and effective

1Actually, there are significant restrictions on the :inherit-from function

7
(define-presentation-method present (object (type integer) stream
  (view textual-view)
  &key acceptably for-context-type)
  (declare (ignore acceptably for-context-type))
  (let ((*print-base* base)
         (*print-radix* radix))
    (princ object stream)))

(define-presentation-method accept ((type integer)
  stream (view textual-view)
  &key (default nil defaultp) default-type)
  (let ((*read-base* base))
    (let* ((token (read-token stream)))
      (when (and (zerop (length token)) defaultp)
        (return-from accept (values default default-type)))
      (parse-integer token))))

Figure 3: Example present and accept methods

method computation work as expected – but that also make parameters and options available as variables in the methods, properly transformed for the presentation type. In Figure 2, presentation-typep is a presentation generic function defined by CLIM. The type argument is a presentation type.

Presentation types and methods are implemented in McCLIM using the Metaobject Protocol[8]. A class metaobject of type presentation-type-class, a subclass of standard-class, is created for each presentation type. The supertypes of the presentation type, retrieved by running the :inherit-from function with dummy arguments when a type is defined, become superclasses of the metaobject. Presentation methods are implemented as regular methods with an extra argument, the class prototype of the class corresponding to the presentation type argument. That is, of course, used to compute the effective method in the usual way.

The body of a presentation method is wrapped by code that expands the presentation type argument from its actual type to the supertype expected by the method and binds parameter and option variables in the body. In effective methods that contain many constituent methods this strategy could lead to poor performance because the expansion functions for the most specific classes need to be be run repeatedly as less specific methods are called. Near term future work will include introducing a caching mechanism to mitigate this effect. An alternate strategy, used by the commercial CLIM implementation, is to perform the expansion outside of the method body, in the method combination code. This approach avoids unnecessary expansion, but it breaks all non-standard
method combination.

Because hidden arguments are passed to presentation methods, they need to be called with special syntax. However, the application programmer never needs to use that syntax unless she defines new presentation functions that are not in the CLIM specification. The user calls a function in the interface – for example, present, and that function calls the presentation generic function of the same name, perhaps after establishing dynamic state and defaulting arguments.

2.6 Input and Output With Presentations

Presentations are implemented as a subclass of output-record which stores the associated object and presentation type. The present function creates a presentation using with-new-output-record and within its dynamic extent invokes the present presentation method for the object and presentation type. The present method for the integer presentation type in Figure 3 shows a typical presentation method that uses presentation type options (base and radix) to alter its output. The view argument – generally supplied from a default view associated with the output stream – provides an additional way to customize the output of the method. textual-view is a view for simple text output; other view classes can imply graphics output. As is typical for textual-view methods the method body is quite simple; the calling present function does most of the work of creating a presentation.

The dual of presentation output, and the motivation for it, is input via presentations. In a CLIM program, the input of a data type is implemented by writing an accept method for that data type. A simple accept method for integer is shown in Figure 3. The accept function performs an important step in addition to calling the accept method for a type: it establishes an input context for the type. Within the dynamic extent of the input context, if the user clicks on a presentation which matches the context’s type the context is exited with the object and type from the presentation as values. Nested input contexts may be created by recursive calls to accept. Figure 4 shows the control stack when a program prompts for a command and the corresponding input contexts. At this point the user can proceed in several ways:

- type a command name and exit the inner accept call, causing the command parser to start parsing command arguments;
- exit the inner accept by clicking on a presentation matching command-name, perhaps from a list of command names generated by a “Help” command or a menu of possibilities;
- click on a presentation matching command, exiting the outer accept.

Input contexts are established with the with-input-context macro. An abridged version of the use of with-input-context in accept is shown in

---

2The output might even go to a string stream.
active function call | input context type
---|---
(read-command 'user-command-table) | input context type

| (accept '(command :command-table 'user-command-table)) |
| (funcall *command-parser* 'user-command-table) |
| (accept '(command-name :command-table 'user-command-table)) |

Figure 4: Input function calls and the corresponding dynamic input context

(with-input-context (type) (object object-type event options) ;; Results of this form are returned if no presentation is ;; clicked on. (funcall-presentation-generic-function accept type stream view) ;; Clauses chosen by matching with presentation-subtypep (t (values object object-type)))

Figure 5: The with-input-context form inside accept

Figure 5. This macro establishes a context for type. If the user clicks on a matching presentation then the variables in the second form are bound and the first matching clause at the tail is executed; otherwise values are returned from the third form are returned.

with-input-context binds the input wait test and input handlers described in Section 2.3 and pushes a cons of the type and a continuation onto the internal dynamic variable *input-context*. The expansion of the form in Figure 5 is shown in Figure 6. The convoluted nesting of blocks allows for fallthrough into the cond form if the context continuation is called while returning all values from the body form if “nothing happens.”

The input wait handler is called for pointer events and checks if there is a presentation under the pointer that matches any of the types in *input-context*. If the event is a pointer press event and there is a matching presentation the handler funcalls the input context continuation to return to the enclosing with-input-context. The wait handler also performs other tasks such as updating any pointer documentation feedback on the screen.

2.7 Input Editing

CLIM specifies an input editing interface that looks, to the user, like a simple Emacs editor with conventional key bindings for ordinary operations such as
Figure 6: The expansion of \texttt{with-input-context} in Figure 5

cursor motion and character insertion and deletion. From the point of view of
the program, input still appears to be coming from a stream. This is achieved
through a mechanism very similar to "Lisp machine rubout handling"[11]. Editing
can be done within the dynamic extent of a \texttt{with-input-editing} macro;
any editing change to input that has already been read by the program causes
the body of \texttt{with-input-editing} to be rerun with the modified input in the
stream. \texttt{with-input-editing} takes a stream argument which is rebound to an
encapsulating stream which delegates all generic functions to the original stream
except those that read input. These functions process input gestures from original
stream as editor commands, update editor state and the screen, and return
the next available gesture that isn't an editing command.

The McCLIM input editor uses an editing substrate, called Goatee\textsuperscript{3},
that is being developed in parallel with McCLIM and which is itself implemented
using CLIM graphics and input features. Goatee will eventually be an Emacs-
like editor in its own right. Goatee's internal design resembles that of Zincs:
its buffers are represented as linked lists of lines. By using a "real" editor to
implement CLIM input editing instead of an ad-hoc editor we leave open the
potential of tight integration between all the editing that a user might do across
all applications.

\textsuperscript{3}Goatee Owes All to Emacs, Evidently
3 McCLIM Development History

Several attempts have been made to create a free implementation of CLIM. In the late 1980s, a supposed project was referenced by the cons.org site, but no CLIM code was actually written, at least as far as an external observer could tell.

In 1998, Mike McDonald and Gilbert Baumann independently started working on a free implementation of CLIM.

McDonald called his implementation “McCLIM”. He elected to work in a “horizontal” manner, supplying enough code for a large number of chapters of the specification in order to run a simple demo application (a version of the “address book” demo supplied by Franz’ CLIM implementation). McDonald’s idea was to distribute his code according to some free or open-source license, but only once the address-book demo was fully functional, requiring among other things a fairly complete implementation of presentation types. Unfortunately, McDonald ultimately had to considerably decrease the time spent on this project.

Baumann, on the other hand, decided to work in a more systematic (but perhaps less rewarding) way. He worked “vertically”, starting with the chapters on regions and transformations. The main reason for Baumann to work this way was that he needed regions and transformations in his Closure web browser. Rather than inventing a new interface, he decided to use an existing one, i.e., the one given in the CLIM specification. The result of this effort was a fairly complete implementation of regions and transformations. Also for reasons of less available time, Baumann had to abandon further work on CLIM. Baumann decided to make his code available to all according to the GNU LGPL license.

In 1999, Robert Strandh was looking for an interface library for his Gsharp score editor. He had just decided to rewrite the Gsharp editor in Common Lisp, so a fully-functional interface library was required. Having read about CLIM, he decided that CLIM would be a good candidate. Gsharp being distributed according to the GNU GPL, it was out of the question to use commercial version of CLIM. He thus decided to suspend work on Gsharp and first work on getting enough of CLIM to work that Gsharp would be feasible. First, Strandh tried to get McDonald’s code in order not to have to rewrite existing functionality, but McDonald maintained his decision not to provide his code. Thus, Strandh started with Baumann’s code and started adding many of the basic sheet protocols, code for ports, grafts, and mirrored sheets, and also drawing functions, drawing options, text styles, and drawing in color.

When McDonald realized in early 2000 that his code would no longer be needed with the progress being made, he agreed to distribute his code, also according to the LGPL so that ultimately his code could be integrated into that of Baumann and Strandh. Because McDonald’s code contained a partial implementation of regions and transformations, Strandh temporarily abandoned Baumann’s code and merged his existing code into that of McDonald. This combination became the code base of McCLIM in that all further modifications were made to this code. In particular, Arthur Lemmens contributed the first
code for some of the gadgets in June of 2000 to this code base.

During the LSM 2000 meeting in Bordeaux, Baumann merged his code for
regions and translations into the main code base. This event, in our opinion,
marked the point at which we were convinced that the McCLIM project would
ultimately succeed. There were a sufficient number of people involved and ev-
everyone was working with the same code base. Strandh wrote the code for menu
bars using this code base.

Various undergraduate student projects contributed code for more gadgets,
layout panes, code for manipulating bitmap images, and a file selector gad-
et. Longer projects by Iban Hatchondo, Julien Boninfante, Arnaud Rouanet,
and Lionel Salabartan (all students at the university of Bordeaux) during 2000
and 2001 contributed a better implementation of the layout protocol, the first
code for the OpenGL backend (the first backend was CLX), output recording,
PostScript output, and several demo applications.

Alexey Dejneka wrote the code for table formatting, bordered output, and
more PostScript backend code. He also contributed (and still does) a very
large number of bug fixes. Motivated by a comment of Strandh’s that “nobody
I know of has any clue as to how to implement presentation types,”[13] Tim
Moore joined in the fall of 2001. Moore wrote the presentation type system,
command completion, input editing, encapsulating streams, the Goatee editor
substrate and much more.

Tim Moore, Gilbert Baumann, Alexey Dejneka, and Mike McDonald are
currently the major contributors to the McCLIM code.

4 Current Status of McCLIM

As we mentioned earlier, McCLIM is almost feature complete, and is mak-
ing rapid progress toward completeness and beyond. This section provides an
overview of the implementation as of this writing (August 2002) with respect
to the CLIM 2 specification

Regions, bounding rectangles and transformations are still largely imple-
mented by Gilbert Baumann’s original code. Some additions and improvements
are necessary, but this code is mostly stable and nearly complete. The entire
geometry substrate can thus be considered almost complete.

The same thing holds for the windowing substrate. Sheet properties and
sheet protocols are mostly complete and functional, as are ports, grafts and
mirrored sheets.

For the sheet and medium output facilities, there is a major gap in McCLIM
with respect to general designs, including transparency, patterns, stencils, tiling,
etc. There are also some minor gaps such as absence of arbitrary ellipses and
elliptical arcs (due to X11 restrictions).

With respect to the extended stream output facilities, basic extended stream
output and output recording are mostly implemented, as are features such as
table formatting, bordered output and text formatting. Graph formatting is
not yet implemented, although it should be fairly easy given that excellent
algorithms and (even some code) exist in some archives. Incremental redisplay is not yet implemented.

The presentation type system and command processing are essentially complete. The input editor is quite usable; at this time it lacks support for killing and yanking text and yanking presentation history.

Finally, the part in the specification regarding building applications is partly complete. Most panes and gadgets are implemented, except that the specification is quite vague on the required features of gadgets such as the text-editor gadget. Parts of the features of stream panes are also missing.

5 Conclusions and Future Work

McCLIM is well on its way to becoming a complete implementation of the CLIM 2 specification. As such, it is just another addition to the family of existing CLIM implementations. However, McCLIM does not share any of the code base of the other CLIM implementations. The fact that McCLIM is a completely new implementation makes it easier to improve on features of the other implementations. In particular, the "native" look of McCLIM is more modern than that of the other implementations. A fresh code base with clear licensing also makes it much easier to integrate new features and bug fixes from users.

We believe that McCLIM can represent the beginning of a period of renewed interest for CLIM and, by association, for Common Lisp in general. Certainly, it means that low-budget users such as university students can try out and explore some of the features of an advanced Common Lisp application, until now reserved for professional users.

The first goal is obviously to supply all, or at least most, of the functionality of the specification. We are convinced that this goal is in sight, and that within another year, probably less, this goal will have been reached. But what next?

We think most users will want to use the CLX backend. For that reason, we want to improve performance for that backend. The current system does not handle X11 event very well, and much too much consing degrades performance even more. That said, recent performance improvements make the system quite usable. Gilbert Baumann has done some promising experiments with anti-aliased text using FreeType and the XRender extension to X11.

For a project of this type, it is important to allow as many users as possible to contribute. For that reason, we need to improve the quality of the existing code. A number of temporary kludges need to be replaced by more thought-out code. Comments and documentation strings should be added wherever reasonable.

We are naturally considering several extensions to the CLIM specification, not just in the form of modern panes and gadgets that did not exist when the CLIM specification was written, but also new functionalities of existing features. In particular, we would like to provide arbitrarily transformable fonts, a library for manipulating various kinds of images, support for sound, and 3D drawing capabilities.
Although CLIM documentation is readily available in the form of the spe-
cification itself and in vendor manuals, we would like to write our own. There
are several reasons for that. First, we need to document holes, ambiguities,
and contradictions in the specification as well as the ways in which they were
resolved in McCLIM. Next, we are not pleased with the structure of existing
manuals. We would like a complete manual starting from a chapter on how to
write simple applications using existing functionalities and eventually leading
up to mechanisms the advanced user might want to take advantage of in order
to add new features to CLIM itself, or to replace existing features with new
ones. Implementation-specific documentation is also needed for extensions to
the CLIM specification.

Finally, we would like to supply backends for other native windowing systems
such as Microsoft Windows and Apple MacOS, and perhaps a backend for GGI.

References

American National Standard for Information Technology: programming
language — Common LISP. American National Standards Institute, 1430
Broadway, New York, NY 10018, USA, 1996. Approved December 8,
reference/HyperSpec/.


the Lisp Machine, second ed. Kluwer Academic Publishers, Boston, Mas-


stream-definition-by-user-notes.html.

[6] Keene, S. E. Object-Oriented Programming in Common Lisp. Addison-
Wesley, Reading, Massachusetts, 1989.

OpenGL, Version 1.1, second ed. Addison-Wesley Developers Press, Reading,
MA, USA, 1997.


