

# Montage: Breaking Windows into Small Pieces

Paul Haahr – Princeton University

## ABSTRACT

Window systems are hard to program because they involve connecting an asynchronous world, where a mouse may move or a key may be pressed at any moment, to programs which execute synchronously. This requirement has led to the use of the “inverted program structure” style of programming, which adds complexity to the underlying code. The Montage window system eliminates this complexity by providing a programming model that uses sequential fragments of code connected by synchronous I/O. The system and its clients are written in an extension to the Scheme programming language which supports concurrency.

In the same way as useful routines can be built up from simple programs by composing Unix pipelines, sophisticated applications can be created in Montage by connecting simple, “lightweight” processes. In addition, programs may be trivially (and transparently) modified by adding processes that filter input to or output from applications. This use of concurrency and composability is isomorphic to “object-oriented” programming things that would be objects in other window systems, such as window decoration (borders, title bars, etc.) or menus are processes but, in practice, Montage processes are easier to write than objects in other window systems.

### 1. Introduction

Most windows systems are large programs with complicated programmer interfaces. Montage is window system that was designed to reduce the complexity of both application programs and the window system itself.

Traditional window systems are hard to write programs for largely because they force programmers to program in sequential languages which interact with an inherently asynchronous world where mouse movements or key clicks can happen at any time. The conventional approaches to managing this concurrency are structuring programs as event loops, or using the so-called “inverted program structure,” where the event loop is in a library rather than in a user’s code. Rather than trying to fit the input to a window system into a sequential model, Montage whole-heartedly accepts the asynchrony. The Montage model derives from Hoare’s Communicating Sequential Processes (CSP),<sup>6</sup> by way of Cardelli and Pike’s Squeak<sup>1</sup> and Pike’s Newsqueak.<sup>12</sup>

Montage’s graphics model is also designed to make applications easier to write. Windows, which may be opaque or translucent, provide retained storage so an application programmer never has to respond to a redraw request from the server. The graphical interface is a small set of fundamental operations based on Porter and Duff’s compositing algebra.<sup>13</sup> The current implementation of the graphics library only supports monochrome displays, but the model generalizes to color systems.

Montage is organized, like most current window systems, along the lines of a client-server model. The window system itself, the “server,” runs as one UNIX process, and communicates by either pipes or TCP/IP connections with application programs, the “clients.” Similar to NeWS<sup>7</sup> and the Blit,<sup>9</sup> but distinct from the X Window System,<sup>17</sup> application programs are split in two parts: one piece which runs within the address space of the server, and another which runs in a separate address space, possibly on a remote system.

The server is an interactive Scheme compiler, linked with about 2500 lines of C code to manage the interface with UNIX system calls and handle the low-level graphics operations. The rest of the server plus a basic window manager and several user interface objects (e.g., menus, window decoration) consists of about 1000 lines of Scheme.

The Montage server currently runs on Motorola 68020 based Sun workstations. Except for the low-level graphics library, which is written for the 68020, the code has run without difficulty on other machines. The client code is portable and runs on a variety of UNIX systems.

Section 2 of this paper covers the concurrent programming model. The graphics library is discussed in Section 3. Section 4 covers the construction of the window system itself.

## 2. Concurrent Programming in Scheme

Scheme is a lexically scoped, properly tail recursive dialect of Lisp. Scheme was chosen as the base language for Montage because:

- Scheme supports closures and higher-order functions.
- As an interactive language, the semantics for dynamically loading code into a running program are clear.
- It is possible to write the concurrency routines within Scheme itself.
- Scheme is garbage collected, which is necessary for a window system that runs (potentially bug-infested) application code inside the server.
- Dynamic type checking was seen as beneficial for prototyping pieces of the system when interfaces were changing.
- High quality implementations exist.

Scheme is defined in reference 14; reference 2 provides a good introduction to the language.

### Montage Processes

The concurrent programming model in Montage is a method for structuring the system and does not imply anything about the implementation; the concurrency is virtual in the same way as timesharing on a uniprocessor machine is, and no true parallel execution necessarily occurs.<sup>†</sup>

Programs are structured as collections of (so-called “lightweight”) processes running within the server that communicate with each other over untyped channels. Processes themselves are anonymous: there is no way to name a process, query its state, or make it abort except by sending or receiving messages over channels. Processes are started with the (`spawn`) macro, which is called with a block of code to execute in a new process. `Spawn` returns immediately.

Scheme has no built-in support for concurrency, but the language defines a construct named `call-with-current-continuation`, or `call/cc`, which makes it possible to implement coroutines, exception mechanisms, or other unusual control flow mechanisms, in a portable way.

The form of a `call/cc` operation is

```
(call/cc procedure-a)
```

where *procedure-a* must be a procedure that takes one argument, say, *procedure-b* which is also a procedure of one argument. If the call to *procedure-a* returns without calling *procedure-b*, `call/cc` uses

---

<sup>†</sup>Currently, processes are non-preemptively scheduled. This is a bug, not a feature.

the value returned by *procedure-a* as its return value. If, on the other hand, *procedure-a* does call *procedure-b*, *procedure-a* stops executing, the value passed to *procedure-b* is used as the return value of the `call/cc`, and execution continues from the point where `call/cc` was called. This mechanism is used in Montage to implement context switching and the process abstraction.

It would not be too much a stretch for a C programmer to consider `call/cc` a variant on `setjmp` with the procedure passed to `call/cc`'s argument corresponding to a matching `longjmp` with the appropriate jump buffer. The fundamental difference between the two is that C's `longjmp` is restricted (in the absence of assembly language trickery) to working within a stack discipline function may only jump to an ancestor in the call history whereas Scheme imposes no such restriction.

### Communication Primitives

A channel is created by calling the function (`make-chan`). Channels are to Montage processes what pipes are to UNIX processes, the glue which connects processes. (Unlike UNIX processes there are no implicit channels analogous to standard input, output, and error.) The notation for receiving from a channel is

```
(<- channel)
```

and sending is

```
(>- channel value).
```

Channels are synchronous rendezvous points: a process that sends on a channel blocks until another process reads from the channel. If two processes are both blocked waiting to receive (or send) on a channel, the one that actually gets woken when some other process sends (receives) is picked non-deterministically. There is no way to determine if a send or receive will block, or, in fact, checking afterwards it did block, without using some out of band mechanism like the system clock.

Multiplexing channels is done with `select`. `Select` derives from the non-deterministic guard of CSP and the `select` statement of Newsqueak. Unlike CSP, where boolean tests can precede only receive operations and Newsqueak, where any tests must lie outside the `select` statement, a clause of a `select` may be guarded by an arbitrary number of conditionals in addition to specifying a potential communication. `Select` takes as its argument a series of *clauses*, where *clauses* are defined by the grammar

```

select
: (select clauses)
clauses
:
| (guard expression ...) clauses
guard
: (if test) guard
| (-> channel value)
| value = (<- channel)
| channel = (->* list-of-channels value)
| value channel = (<-* list-of-channels)

```

When a select is executed, the if parts (where present) of all clauses are evaluated; send or receive operations on channels mentioned in clauses where the if part is true or not present are initiated. If no communications can complete, the select blocks until one can. If or when a communication completes, the expressions associated with that guard execute. If multiple channels can complete at any time, one is picked at random. The <-\* and ->\* forms are used to select on any one of the channels on a list. The value of the entire select is the value of the last expression evaluated. Inside the body of code following each guard, the symbols to the left of the = are bound to the value and/or channel used in the communication.

Standard UNIX file operations, such as open, close, read, and write, are available from Scheme code. UNIX I/O calls are deferred until all running Scheme processes block, when a call to UNIX select is made.

One interesting side effect of running processes in a garbage collected environment is that if a set of processes deadlock all are waiting for communications to complete, and none of the channels that are reachable outside the set of processes they are garbage collected. This can be very convenient, because processes do not have to be told explicitly to quit when the rest of the system is done with them; instead, the processes that communicated with them just ignore the channels connected to the processes in question and the resources used by those processes are reclaimed.

On the other hand, it is disconcerting to debug a system where processes that deadlock just disappear: a running program can stop and go back to the evaluator prompt without leaving a trace behind if it enters a deadlocked state. Deadlock prevention, like ensuring the absence of all other forms of bugs, is up to the programmer.

### Examples

A simple example of the use of process, adapted from Newsqueak,<sup>12</sup> is a sieve of Eratosthenes prime number generator.

```

(define (counter chan n)
  (-> chan n)
  (counter chan (1+ n)))

```

```

(define (filter-multiples n in out)
  (let ([m (<- in)])
    (if (not (zero? (remainder m n)))
        (-> out m)))
  (filter-multiples n in out))
(define (sieve in out)
  (let ([p (<- in)] [c (make-chan)])
    (-> out p)
    (spawn (filter-multiples p in c))
    (sieve c out)))
(define (make-sieve)
  (let ([c (make-chan)]
        [prime (make-chan)])
    (spawn (counter c 2))
    (spawn (sieve c prime))
    prime))

```

Running the sieve gives: (the evaluator's responses are shown in italics)

```

> (define primes (make-sieve))
> (<- primes)
2
> (<- primes)
3
> (<- primes)
5
> (<- primes)
7
> (<- primes)
11

```

A buffer process for a channel written using select is:

```

(define (buffer size in)
  (let ([out (make-chan)]
        [buf (make-vector size)])
    (define (incr n)
      (modulo (1+ n) size))
    (define (do-buffer count lo hi)
      (select
        [(if (< count size))
         v = (<- in)
         (vector-set! buf hi v)
         (do-buffer (1+ count)
                    lo (incr hi))]
        [(if (> count 0))
         (-> out (vector-ref buf lo))
         (vector-set! buf lo '())
         (do-buffer (1- count)
                    (incr lo) hi)]])
    (spawn (do-buffer 0 0 0))
    out))

```

### 3. Graphics Model

Graphics in Montage are based on the notion of compositing images. Every image has two parts, one for the data, which contains the color of each pixel, and one for a matte, which contains the degree of opacity for each pixel. In general, one uses a matte channel with as many bits per pixel as are

provided in display hardware per channel of data: for example, a monochrome display will have one bit per pixel of matte as well as data, while a 24-bit RGB display would typically have 8-bits for the matte. Since Montage currently runs on only monochrome hardware, the data and matte portions of each image can be thought of as simple, 1-bit deep bitmaps. In the 1-bit model, if a matte bit is off, the image is clear for that pixel; if it is on, the pixel is opaque and has the color specified in the data bitmap.

Each image has both a data and matte section. Since storing two window-sized bitmaps for every layer is wasteful, especially in the presumably common case of fully opaque windows, Montage introduces an idea called "cards." A card is an infinite plane extending in all dimensions of one value, that may be used in lieu of the bitmaps for either (or both) of the data or matte portions of an image. Two special images exist, `WhiteImage` and `BlackImage`, which have data cards of their respective colors and are completely opaque.

All graphical operations in Montage are specified in a coordinate system that is one-to-one with pixels and has its origin in the upper left hand corner of the window to which the operation refers. Windows are clipped on display to their parent layer's rectangles, but offscreen portions may be drawn to without penalty, and will appear on the screen if the window is moved. Windows coordinates are relative to the upper left in order that windows do not have to be notified when they are moved.

### The COOP Operator

The main graphics operator used in Montage is `coop`, for "compositing operator." A call to `coop` has the following form:

```
(coop dest-image dest-point
  source-image source-rect
  operation)
```

(The form of the call was intentionally modeled on the traditional monochrome `bitblt` operator.) The `dest-point` refers to the upper left of the rectangle affected within `dest-layer`; the size is determined from `source-rect`. The rectangles are clipped to lie within the boundaries of their respective layers.

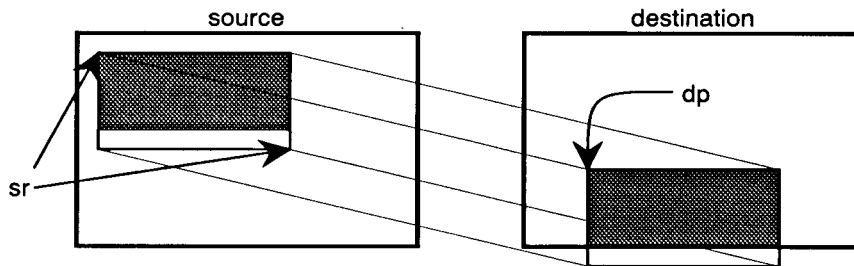


Figure 1. Clipping the coop operator.

Figure 1 gives an example of the clipping that occurs in a `coop` operation; the grey rectangle indicates the parts of the source rectangle that are composited onto the destination.

Eleven compositing modes are defined, in addition to a separate operation which clears both the data and matte portions of an image. The `coop` operator is defined as a two-address instruction, where one operand, the source, is read and the other, the destination, is read and written. Nothing in the operator prohibits using a three-address variant, but the two-address form was picked for reasons of efficiency and simplicity.

Of the eleven operations, one (`D`) is a no-op and one (`S`) copies the source to the destination. `SoverD` is the image that contains the source, where its matte defines it as present, and the destination where it is present if the source is not. `SinD` contains the source where both the source and the destination are present, and nothing else. `SoutD` contains only the source where the destination is not present. `SatopD` consists of the parts of the destination that are not covered by the source and the parts of the source that cover the destination. `DoverS`, `DoutS`, `DinS`, and `DatopS` define the symmetric operations to the previous four. `DxorS`, which is the same as `SxorD`, contains the parts of the source and destination that are not covered by each other, and is clear both where they intersect and where neither is present.

The `coop` operations are illustrated in figure 2, using a grey triangle in the upper left as the source image and a black triangle in the upper right at the (initial) destination image. Both images are assumed to have mattes covering the regions that are not white. The grey image is implemented as a stipple pattern in the data portion of the layer, but the matte portion is not stippled.

Each `coop` operation is implemented as between one and four calls to a conventional `bitblt`<sup>4,10</sup> routine, depending on what operation was specified and whether bitmaps or mattes are used for images.

Several other graphics primitives are implemented in Montage, for drawing standard objects such as lines, circles or text strings. All these operations work by specifying a color and a `coop`

operation in addition to the shapes described. For example, the `coop-string` procedure draws a string as if there were an image with a solid color data card and a matte containing a rasterized version of the string. Currently, only bitmapped fonts of horizontal orientation are supported.

### Layers: Overlapping, Hierarchical, Translucent Images

The term “window” in Montage is reserved for the concept of a window that a user sees on the screen; windows are implemented in the underlying system by one or more “layers,” which correspond to the concept of windows in X, canvases in NeWS, or layers in the Blit. Layers are organized as a forest of rooted trees; the screen is treated as a normal layer, named `*screen*`, at the root of one tree.

The children of a given layer are organized in a front-to-back order. Each layer has an associated priority, which defaults to 0 but may be set to any value in the range  $-128 \leq n < 127$ . Currently the only layer which has a non-zero priority is the cursor, which in all other respects is treated the same as any other layer. The same facility can be used, however, to provide effects like the “Icon Dock” in the NextStep user interface.<sup>8</sup>

There are three calls to create layers:

```
(make-layer parent rectangle)
(make-opaque-layer parent
 rectangle background)
(make-outline-layer parent
 rectangle color)
```

The rectangle specifies the area within the parent which the layer covers. While the child layer exists, drawing in that region of the parent is ignored. If

the parent named evaluates to false, the new layer is created at the root of a new (offscreen) hierarchy.

`Make-opaque-layer` creates a layer with a solid (opaque) card as the matte; these layers are used for most clients, and it is the default type of layer created for a new application. Outline layers have cards for their data part, of the color specified in the creation command, and have bitmaps for their matte; they are similar to overlay canvases in NeWS, and are useful for rubber-banding a line, reshaping a window, or creating a grid for a draw program. `Make-layer` creates a layer with bitmaps for both data and matte; these layers can be used to create arbitrarily shaped windows (for round clocks or other special effects). The mouse cursor is created at system initialization with a `make-layer` request. At creation, opaque layers are textured with their specified background pattern; other layers are created with clear matte bitmaps.

Offscreen or obscured portions of layers have associated backing store, so programs never have to redraw windows. If portions of a layer are obscured, the entire image is stored in an offscreen bitmap. When a graphical operation is executed on an image in backing store, the operation itself is done only once, and the result of the operation are projection forward through the layer stack and up the hierarchy. If a layer is obscured by an opaque layer, the projecting stops at that point; when the obscuring layer is not opaque, the new image is computed with the `DoverS` compositing mode, and the projection continues.

While one might question the cost in memory size of this decision, experience has shown that applications which can count on the server maintaining a window’s contents are significantly easier to

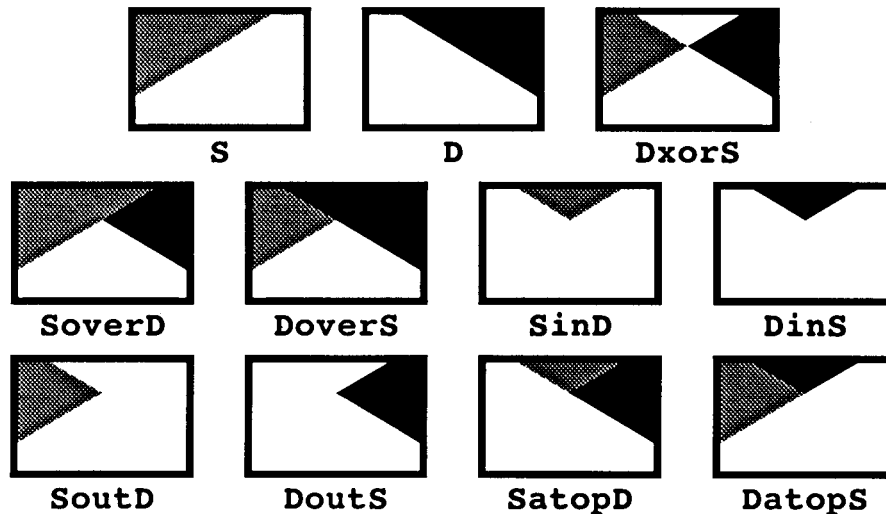


Figure 2. Compositing operations.

write. A window system that requires clients to redraw their contents after “damage” events is similar to a network protocol that requires applications to do their own error correction; while such services are usable, easy to implement, and well-suited to some specific applications (e.g., real-time video in a window or broadcasting a message), most programmers prefer to use a higher level service like reliable bytestreams instead of building their own error-corrected service on top of datagrams.

For more information about compositing, see Porter and Duff’s paper.<sup>13</sup> Salesin and Barzel<sup>16</sup> discuss compositing in the context of monochrome displays. More details about the system used in Montage can be found in reference 5.

#### 4. The Window System

The window system itself is constructed along the lines suggested by Pike.<sup>11</sup> All clients of the window system share a similar interface. Each client is started with a function call of the form

```
(client window keyboard
      mouse co ci fd)
```

The window is a layer for the client to draw in. The keyboard is a channel on which a message is sent for every key press directed at the client. Similarly, the mouse channel receives a packet containing the current state of the buttons and a mouse coordinate every time either the buttons state changes or the mouse is moved. The channel `ci` is used for control messages from the window manager to the client; `co` is used for control messages initiated by the client. The control requests include reshaping or deleting windows. `fd` is a connection to a peer process running on a host machine that implements the back end of the application; for example, in the case of a terminal emulator, the peer process would usually be a shell.<sup>†</sup>

The rules for communication over a client’s channels are fairly simple, and can be summarized as: a client must respond as quickly as possible to messages on the mouse, keyboard, and control channels. Unlike the X Window System, where there is a large body of advisory rules governing communication between clients of the server,<sup>15</sup> Montage has few rules, but they are mandatory. Ignoring a request to quit or letting messages back up on the mouse channel, for example, can cause the system to

---

<sup>†</sup>The Montage terminal emulator does not actually communicate directly with a shell, but rather an intermediary process running on the same host as the shell. The problem with uses a direct connection arises because: (1) many applications run differently if they are invoked on a terminal (or pseudo tty) from the way they run on a pipe or network connection, so a pty is needed; and (2) a connection to a pty cannot be passed over the network.

lock up. In practice, maintaining the necessary invariants to keep the system running is not hard, but it is easier than it should be for one client to make the entire server unusable. One clear advantage that a window system, such as X, with a fixed client-server protocol and no client code running within the server has over a system like Montage is that a misbehaving client will rarely cause a correctly implemented X server to crash.

Each client is created by a window manager running within the server; the file descriptor is usually created in connection with the peer UNIX process of the window manager. There is a small protocol, available to Montage clients that want to create their own connections, which simulates creating a pipe shared by two processes that are connected by some connection. The protocol is implemented by having the process create a socket, tell the Montage server its hostname and port number, and listen on the socket. (If the two processes are on the same machine, a bidirectional pipe is used rather than an internet socket.) The Montage server connects to the socket and hands the file descriptor of the connection to the client.

In addition to creating clients, the window manager is responsible for multiplexing input among clients. It is in the window manager that user interface policy regarding input distribution is encoded. For example, two window managers have been written for Montage, one which implements a “click-to-type” user interface, the other a “focus-follows-cursor” policy. Changing interfaces requires only using a different manager. Each window manager is roughly 200 lines of concurrent Scheme, and they differ in only a few places.

The window manager itself has the same interface as its clients. Instead of being passed a child layer of the screen, its window is the screen itself. Similarly, at the other end of the window manager’s keyboard and mouse channels are processes that read directly from the the input devices. The control channels for the window manager are connected to processes that let the window manager act as if it were running as a client of another window. The file descriptor for the window manager is connected to a UNIX process which forks child processes to run in windows and prepares their I/O connections to the server.

Pike has made the observation that if the window manager has the same interface as its clients, it can be used as a client as well as at the root of the window hierarchy.<sup>11</sup> One possible use for running the window manager recursively, is that an application which itself contains subwindows can delegate the responsibility for managing the windows to a new instance of the same window manager routine, called with slightly different parameters. Another use of running the window recursively is creating a subsidiary window manager in a different context

from the root window manager, e.g., logged in to a remote machine or running as a different user.

It is precisely this feature which makes Montage convenient to use as a networked window system. The sequence to start a remote window manager in montage is quite easy: a user opens a new window, logins in remotely to another host, and gives the command `rws`. `Rws` (recursive window system) sends an escape sequence to the terminal emulator that causes the client part of the terminal emulator running within the Scheme interpreter to execute a fragment of Scheme code. The code fragment replaces the terminal with a window manager, which communicates with a window manager peer process that runs on the remote machine. After these steps (which appear to the user as logging in to another system and running one command), the window has become a new window manager. Inside the window, opening a new (sub)window causes a shell to start running on the remote machine; opening a new window from the root window manager still creates a local shell.

Figure 3 illustrates a window manager with three clients. Two of the clients are terminal emulators, the third is a recursive window system running on a remote machine. The lines with double arrows indicate pipes or TCP/IP sockets connecting Montage processes to UNIX processes. Lines with single arrows are drawn from UNIX processes to their child processes.

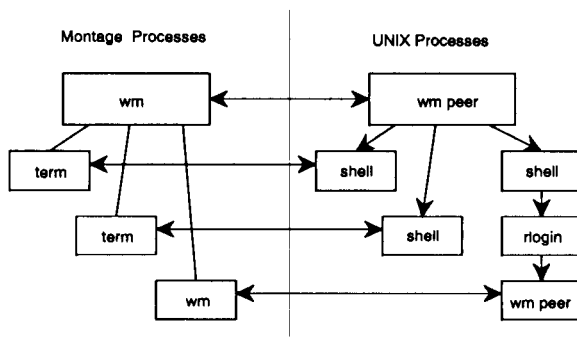


Figure 3: Configuration of processes

## 5. Objects, Widgets, and Processes

The most common technique for structuring window systems, and programmers' interfaces to window systems, is the object-oriented model. In fact, Smalltalk,<sup>3</sup> which is generally considered the first window system, is also the most consistently object-oriented programming environment in existence. Two aspects of the object-oriented model make it well-suited for constructing window systems: the object model forces programmers to identify and formalize interfaces; and the distribution of program state into objects, rather than a less-structured global context, hides some aspects of the

asynchrony inherent in a window system from the programmer.

Montage consciously rejects the object-oriented model, using instead the multiple process model as the structuring paradigm. Processes in Montage are analogous to objects in object-oriented window systems. Things that would be objects in other systems, for example, the so-called "widgets" of X, are processes in Montage.

One can look at processes and objects as two different notations for the same concept. Both objects and processes associate local data with procedure, and both specify formal interfaces from outside to the private data: where objects provide this interface as part of a type system, processes specify a protocol to be used on their channels. The fundamental difference I would identify between objects and processes is that in an object, all state is explicitly maintained, whereas part of the state of a process (the location the "program counter" points to and the dynamic call stack) is maintained implicitly. This implicit state saving is a large part of the notational convenience behind using processes.

As an example, in an object-structured window system, unadorned windows would be one class, and windows with decoration would be a subclass inheriting from the plain windows. In Montage, a client, with the interface described above, can be wrapped in decoration with the following procedure.

`*decorate*` returns a client procedure with the same calling interface as the original, but the wrapped procedure has a thin border, that highlights when the window is focused, and responds to requests from the window manager to move to front or the back of the window hierarchy. This procedure implements a minimalist interface, similar to the one found on the Blit.<sup>9</sup> More complex intermediate processes are possible, and no part other than the wrapping code has to be aware of what is being done. In particular, the window manager knows no details of what `*decorate*` does, it just follows a predefined protocol, which is filtered by the decoration routine. (The reason that `tofront` and `toback` are handled in the decoration routine and not the window manager is that one decoration routine creates physically separate layers for the window itself, one for a title bar that move to the front and back of the layer stack as a group.)

Other wrapping procedures are possible. One that has been implemented already is a keyboard abbreviation expander: the intervening process watches the keyboard channel for certain key sequences those with the meta key depressed and maps them to words from an association list. Again, the process that eventually receives the expanded key sequences does not know, and, in fact, can not determine, whether an abbreviated or expanded form was typed by the user.

A similar structuring can be applied to other user interface widgets, such as dialogue boxes, menus, scrollbars, etc. All a client of such a service is aware of is that a channel has at the other end of it a process which sends, say, selections from a menu. The anonymity of connections allows complex user interfaces to be built up from simpler ones, similar to the way UNIX pipelines are composed of independent filters.

## 6. Conclusion

Montage is an experiment in making window systems and their applications easier to write. The multi-process structure of Montage is a solid basis for constructing a window system, and provides a clean abstraction for implementing the services usually offered by a window system.

The graphics model used in Montage is perhaps its best feature. Using one data structure, the layer, for all visible objects is an effective unifying mechanism which gives the system a consistent feel for the programmer. The compositing algebra meshes extremely well with the idea of overlapping windows in a way that neither `bitblt` or `PostScript`<sup>7</sup> does.

Montage is not a practical system. The `call/cc` construct of Scheme, while powerful and portable, does not achieve the performance necessary for the near real time constraints that a window system imposes. Specifically, in every one of the dozen Scheme implementations that I have tested `call/cc` allocates so much memory that most of the run time is spent in the garbage collector. The Montage concurrency implementation provides the functionality one would want in a window system, but with performance only suitable for a prototype. For a language to support concurrency of the granularity used in Montage with adequate efficiency, it must be designed from scratch with concurrency in mind.

## Acknowledgements

The work described in this paper was advised by Pat Hanrahan; Pat is also responsible for pointing me towards the compositing model. Dave Hanson advised my earlier work on window systems which led into Montage. Many of the ideas presented here were hashed out with Sean Dorward. Discussions with Norman Ramsey about his implementation of concurrency in ML influenced much of my own work. Matt Blaze, Sean Dorward, Pat Hanrahan, Pat Parseghian, and Byron Rakitzis provided comments

```
(define (*decorate* client)
  (define (decor $w $k $m $co $ci $fd)
    (let* ([lrect (layer-rect $w)]
          [border *decor-border*]
          [frect (rect-inset lrect border)]
          [w (make-opaque-layer $w
                                (rect-inset frect (* 2 border)) White)]
          [ci (make-chan)])
      (define (intercept)
        (let ([mesg (<- $ci)])
          (case (car mesg)
            ((focus)
             (draw-border $w frect
                          border Black SoverD))
            ((unfocus)
             (draw-border $w frect
                          border White SoverD))
            ((tofront)
             (tofront $w))
            ((toback)
             (toback $w))
            (else
             (-> ci mesg))))
        (intercept))
      (draw-border $w lrect *decor-border* Black SoverD)
      (spawn (client w $k $m $co ci $fd))
      (intercept)))
  decor)
```

on drafts of this paper.

I would also like to thank Kent Dybvig and Cadence Research Systems for providing a copy of Chez Scheme for development of this system.

#### References

1. Luca Cardelli and Rob Pike, "Squeak: A language for communicating with mice," *Computer Graphics* 19(3) pp. 199-204 (1985).
2. R. Kent Dybvig, *The Scheme Programming Language*, Prentice-Hall, Englewood Cliffs, NJ (1987).
3. Adele Goldberg, David Robson, and Daniel H. Ingalls, *Smalltalk-80: The Language and Its Implementation*, Addison Wesley, Reading, MA (1983).
4. Leo J. Guibas and J. Stolfi, "A language for bitmap manipulation," *ACM Transactions on Graphics* 1(3) pp. 191-214 (July 1982).
5. Paul Haahr and Pat Hanrahan, "If They're Called Windows, Why Can't We See Through Them?," Princeton University Computer Science Tech. Report (1990). (In preparation.)
6. C.A.R. Hoare, "Communicating Sequential Processes," *Communications of the ACM* 21(8) pp. 666-678 (August, 1978).
7. *NeWS 1.1 Manual*. January 1988.
8. *NeXT System Reference Manual, Version 0.9*. 1989.
9. Rob Pike, "The Blit: a multiplexed graphics terminal," *AT&T Bell Labs. Tech J.* 63(8) pp. 1607-1631 (1984).
10. Rob Pike, Bart Locanthi, and John Reiser, "Hardware/Software Tradeoffs for Bitmap Graphics on the Blit," *Software -- Practice and Experience* 15(2) pp. 131-151 (February 1985).
11. Rob Pike, "A Concurrent Window System," *USENIX Computing Systems* 2(2) pp. 133-153 (Spring 1989).
12. Rob Pike, "Newsqueak: A language for communicating with mice," Computing Science Technical Report 143, AT&T Bell Laboratories, Murray Hill, New Jersey (January 1989).
13. Thomas Porter and Tom Duff, "Compositing Digital Images," *Computer Graphics (Proceedings of SIGGRAPH 84)* 18(3) pp. 253-259 (July 1984).
14. Jonathan Rees<sup>3</sup> and William Clinger (editors), "The Revised<sup>3</sup> Report on the Algorithmic Language Scheme," *ACM SIGPLAN Notices* 21(12)(December 1986).
15. David S. H. Rosenthal, *Inter-Client Communication Conventions Manual*, MIT X Consortium Standard, Version 1.0 (1989).
16. David Salesin and Ronen Barzel, "Two-Bit Graphics," *IEEE Computer Graphics and Applications* 6(6) pp. 36-42 (June 1986).
17. R.W. Scheifler, J. Gettys, and R. Newman, *X Window System: C Library and Protocol Reference*. 1988.

Paul Haahr recently completed an AB in Computer Science at Princeton University. His research interests include window systems, programming language design, operating systems, and computer architecture. Haahr has previously worked at Oracle Corp., Belmont, CA, and Polygen Corp., Waltham, MA.

