Generalized Pointing: Enabling Multiagent Interaction

Dan R. Olsen Jr., Daniel Boyarski, Thom Verratti, Matthew Phelps, Jack L. Moffett, Edson L. Lo
Human-Computer Interaction Institute, Carnegie-Mellon University
Pittsburgh, PA 15213, dolsen@cs.cmu.edu

ABSTRACT

We describe an architecture which allows any external agent (human or software) to point into the visual space of an interactive application. We describe the visual design of a scheme for highlighting any information in any application. This architecture requires the application to provide information about its semantic structure as part of its redraw algorithms. Based on this semantic map generalized pointer descriptions are defined and used to reference objects to be highlighted. The architecture is demonstrated using a multibookmark agent framework and several example applications.

THE MULTIAGENT POINTING PROBLEM

This work takes as its fundamental assumption the importance of transforming the single-user interactive environment into one where multiple agents interact with the user in the same work context. We use the term agents to refer to any human or process that acts independently in the same work context as the primary user. Agents may be other people, spell checkers, design critics [3], grammar checkers, assistance wizards, scripting by example or other tools that can offer assistance in the interactive workspace.

A second basic assumption of our work is that multiagent activity must be pervasive throughout the interactive environment and not confined to specific applications. This is true because 1)users will need assistance (human or software) in all aspects of their endeavors not just special "cooperative places," and 2) the mechanisms for conversing with other agents need to be uniform across the environment so that the burden of cooperation does not become so high that it is worthless.

CHI 98 Los Angeles CA USA
Copyright 1998 0-89791-975-0/98/ 4..$5.00

Achieving a multiagent interactive environment is too large a challenge for this paper. A fundamental stepping stone towards this goal, however, is the ability for agents to point into a visual space without interfering with the user's ability to interact in that space. Whenever two people discuss anything that has a physical manifestation, whether document, image, automobile part, or dog, they regularly gesture and point. The ability to draw visual attention to the particular aspect being discussed is fundamental. It is this ability of a software-based agent to point (draw the user's attention) into the visual space of any application in the environment that is the key contribution of this paper.

This pointing problem manifests itself in many forms in existing software. It occurs when a spreadsheet wants to show all cells on which some formula is dependent, when a search agent shows all instances of the word "foo" in a drawing, when Widgets By Example [7] needs to show the objects referenced in an inferred constraint, when a design critic must indicate all objects violating design rules, or when a change management tool must show recently modified objects.

An important design assumption in our work is that pointing by other agents needs to attract attention rather than coerce it. The behavior of tools such as Find or Spell Check, which commandeer the user's selection mechanism as a means of drawing the user's attention, is unacceptable. The user must be in control of how much attention other agents can demand.

Our approach

Our attack on this problem has two aspects. The first is the visual design task of developing a highlighting or pointing mechanism that will visually attract a user's attention without interfering with the user's work, and without requiring any application to change the design of its information display. The second problem is to algorithmically modify an application's display so as to present the highlight to the user without forcing application programmers to embed the highlighting scheme into the code of every application. The pointing or highlighting scheme must also support multiple agents working simultaneously in consort with the user.

VISUAL DESIGN OF HIGHLIGHTING

The problem with designing a highlighting scheme that works across all applications is that any visual cue that one might choose could conflict with visual cues coded into a given application. We were very unwilling to restrict the visual design space of applications in order to accommodate the highlighting. The problem is to draw the user's attention to an area without obscuring the application information already there and without having the highlight be mistaken for application content.

A second requirement is that the user must be able to control the strength of a highlighting scheme. Unlike spell checkers, which function by dominating the interaction, we wanted the user to be aware or ignore the activities of independent agents as desired. This means that highlighted as well as unhighlighted portions of the display must be visible and usable at the user's discretion.

A third requirement is that multiple simultaneous agents must be supported. If there is a grammar checker, a spell checker, and a change management tool active in the environment, it must be possible for each to draw the user's attention and the user must be able to discriminate among them visually.

The last requirement is that the highlighting must involve a simple direct algorithm that is easily implemented as part of the application drawing architecture. The drawing of the highlight cannot impede the performance of the application.

The process

Our approach to this design problem was to take screen dumps of as many existing application as we could find. For each screen dump we devised a set of possible objects to be highlighted. This formed our visual test-bed against which we could test the effectiveness of our ideas. We tested and discarded a number of ideas:

- *Bolding or thickening of selected objects.* This distorts the nature of the selected objects.
- *Blurring of unselected information.* This is quite effective in visually highlighting but reduced clarity.
- *Make the selected objects wiggle.* Motion is very difficult to ignore and therefore too demanding for our purposes.

Basic highlight

We concluded that there was no acceptable highlighting that would be visually orthogonal to every application's information display. We instead focused on "bending the color space" of unhighlighted items so that they would visually recede relative to the highlighted objects. Our approach is a simple blending of all unhighlighted colors with some neutral color. Blending unhighlighted items with a neutral color reduces their contrast and detail, making the highlighted items more visually prominent and unchanged.

Choice of the neutral blending color is somewhat application-dependent. For most applications a light gray is very effective. For applications which are monochrome, a pastel color is effective because the unselected items assume a hue that contrasts with gray levels of the selected items.

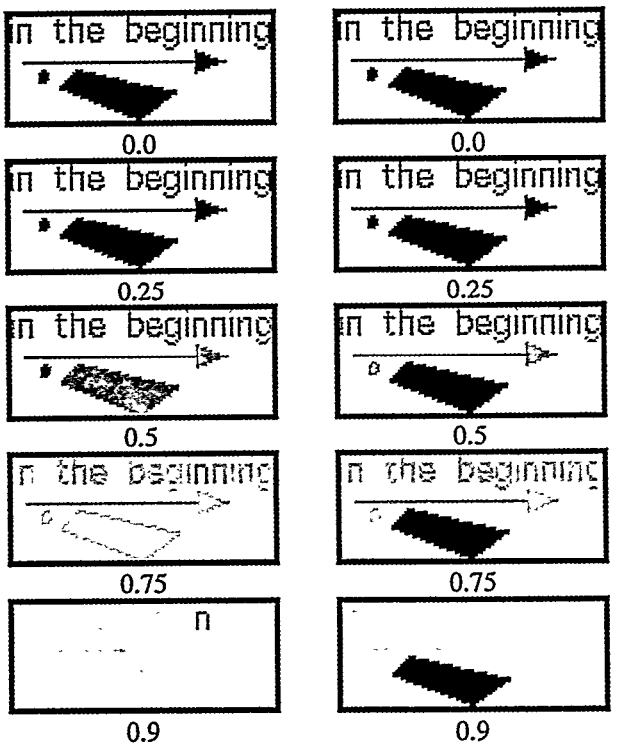

Figure 1 - Varying the highlight intensity

The parameters to our function are the *blending color* and the *highlight intensity*. Highlight intensity is a user controlled parameter ranging from 0.0 (no highlighting) to 1.0 (only selected objects can be seen). Our experiments showed that users must have control of the highlighting. First, because user goals may vary, and secondly, because various application objects have different levels of visual strength. Compare the highlighting intensities in the right column of figure 1 to those in the left. Note that the polygon on the right stands out as a highlight much sooner than the "n" on the left.

The effect of the highlighted polygon is much more striking on a large display than in the small images of figure 1. The "n" in a large image must show very high highlight intensity before it is visible at all. Note, however, that high highlighting intensity makes the unhighlighted objects much harder to work with. The intensity parameter gives the user the necessary control to adjust to the task at hand.

Given the two parameters (BC and HI) and a color for an unhighlighted object(OC) the formula for the blended color is BC•HI + OC•(1-HI).

Multiagent highlight

The blending method serves to bring highlighted objects into the foreground. This does not deal with the problem of discriminating among multiple agents. Solutions to this problem will depend on whether there are 2-3 agents, 10-20 agents, or 100+ agents. Our conclusion is that other than outright labeling of every highlighted item with the identity of the responsible agent, we could not visually encode more than about 5 agents at a time. We concluded that there would need to be a legend somewhere that related the visual encoding to information about the actual agents, be they people or software. Such a legend could group together many agents in a single encoding and all the encodings would be under user control.

The problem of visual conflict between the agent identification and the visual encoding of application information is mitigated by the way in which the blending algorithm works. By blending colors of all unhighlighted objects towards a single color, the color space of the unhighlighted area is sharply restricted. (See figure 1.) By placing our agent encoding in the unhighlighted area we can use a portion of the color space that is distant from the blending color and thus separate the agent indicators from application information. As with the simple highlight the visual contrast of the agent indicators with application data will increase as the highlight intensity is increased.

Our agent encoding technique computes a rectilinear region for the selected area of each agent, expands that region by a small number of pixels, and then draws that region's border using the color associated with the agent. By expanding the region before drawing it, we push the agent encoding away from the highlighted objects so that it does not interfere with them and out into the blended areas where the visual contrast will be better. If the blending color is a gray (which we have found effective in most cases), encoding agents with color that have above average saturation works very well.

One of the problems with multiagent pointing is that more than one agent may point at the same object. The agent indicators (in our case the borders) may overlap and obscure each other. We have accommodated this by expanding each agent's region by a different number of pixels. This prevents the overlap and preserves the encoding information. The end result is shown in figure 2.

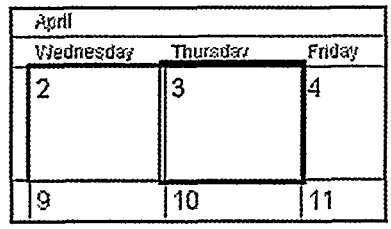

Figure 2 - Multiagent Selection

THE NATURE OF POINTING

The purpose of this work is to allow multiple agents to draw a user's attention to various items in the workspace. In most real applications this has both a local and a global aspect, due to the fact that real applications typically occupy a visual space much larger than is available on a computer screen. Local pointing involves highlighting those currently visible objects to which the agents want to draw attention. The highlighting techniques described above deal with the local pointing problem.

The global pointing problem involves drawing the user's attention to objects that are not currently visible. Global pointing depends on the organizational model for the information and how the user will navigate the information space. The simplest of these is the infinite 2D surface with scrolling as the navigation technique. Techniques for pointing in this model have already been developed including scroll-bar variants [5,6] and radar views [4]. A software architecture for integrating these techniques with any application is discussed later in this paper.

Other information organizations include the zooming model of Pad++ [1]. Hierarchic or linked organizations are also used as in the Macintosh Finder or the workbook sheets in the new versions of Microsoft Excel. The zooming model shares most of the features of the scrolling model with the exception of objects that are "zoomed" to smaller than a pixel. Zoomed objects that are outside the currently visible range can be dealt with just as with other "scrolled-away" objects. By assigning any selected object a minimum region of 1 pixel and then expanding that region by several pixels, even objects that are zoomed into invisibility can be highlighted.

In hierarchic organizations, there is always a visible object that serves as a surrogate for a group of hidden objects. In the Finder, for example, the surrogate for a hidden directory is the folder icon. In the Excel workbook there is a tab across the bottom for each hidden sheet. If objects inside of a hidden group are selected by an agent, the surrogate object can be highlighted using the techniques already described. For example, a directory folder can be highlighted if any of its contents are being referenced by an agent. The user thus has sufficient information to locate the highlighted material.

All of the above models for information can use the blended highlight with other global pointing widgets to provide a complete pointing mechanism for multiple agents. The one exception to this is the searchable space. This is a space such as a database that has no visual organization. Queries are created and results are returned but there never is any global geometry assigned to the entire space of information. No pointing technique is useful in this case. In such cases agents must describe possible queries that would return the selected information. Such techniques are important but outside the scope of this work.

A notable hybrid is the World Wide Web. For the web as a whole, there is no global visualization. The WWW in a global sense is a searchable rather than a navigable space. On the other hand, once a page is selected a local traversal tree of direct and indirect links from that page can then be treated using the techniques described above. Agents will then have an organization into which they can point and techniques for doing the pointing.

IMPLEMENTATION OF POINTING

There are several problems in actually implementing these pointing techniques on top of any application in the interactive environment. Our goal is to require only minimal changes to the applications and that these simple changes suffice for a wide range of agents and interactive services.

The first and most important problem is to actually define what a pointer is. For pointers to have any real meaning they must be defined in terms of the semantic structure of the application. For our highlighting techniques to work this semantic structure must be mapped onto the objects drawn on the screen by the application. Having defined the nature of a pointer we need algorithms to implement the highlighting without requiring a rewrite of all applications. Lastly we need to extract geometry information about the highlighted regions so that agent indicators and other global pointing widgets can be implemented.

Semantic / Surface Mapping

Every interactive application has some form of the architecture shown in figure 3. In most such applications this architecture is replicated for each independent view of the model, or views of different models.

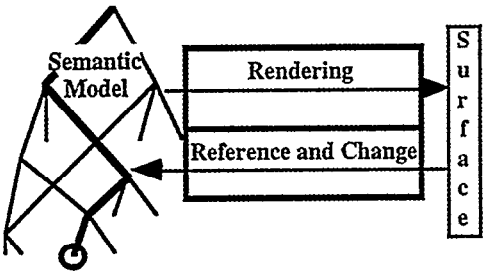

Figure 3 - Application Architecture

The goal of the application is to present and manipulate the information in the semantic model. For a given view, there is code that renders some portion of the semantic model onto an infinite 2D surface. Scrolling or other navigation techniques are used to present this surface on the limited screen space available to the application. As part of this rendering operation, the application usually traverses the semantic model in a semirecursive fashion, extracting information from the model and transforming it into drawing primitives on the surface.

The user in turn generates inputs in terms of the images presented on the surface. The application must translate these surface-based inputs into references to objects in the semantic model and changes to that model. Frequently such manipulations are defined in terms of a selected object or objects (as indicated by the circle in figure 3). These selected objects are the raw material for our pointers.

The problem is to define application-independent references to such selected objects so that external agents can save, manipulate and highlight the referenced objects. When the rendering code traverses the semantic model, it

generates a tree traversal of the information regardless of the actual structure of the model. Different views will take different traversals and thus generate different trees. Our approach is to capture the structure of this traversal tree during the rendering process so as to create a map between this traversal tree and geometric objects on the surface.

As the model is being traversed, the rendering software moves from object to object by following links of various sorts. Each such link can be identified by a textual name or an integer index. Such identifiers need only have meaning to the application itself.

Defining pointers

Note that pointing, highlighting and selecting operations are all defined at the surface where the user can perceive them. Because of this surface orientation we use the renderer's traversal tree as our model description rather than the full generality of the semantic model's structure. Based on this, a given object can be identified by the path from the root of the traversal tree to the desired object (as shown by the darkened line in figure 3). A simple object pointer, therefore, is a sequence of integers or strings defining the links between objects in the rendering traversal. Exactly what these strings or indices are and how they relate to the underlying structure of the application is unimportant, provided they are used consistently by the application. This approach for defining pointers is similar to our earlier work on semantic telepointers [8].

In many cases an agent will need to reference more than one object. A spell checker, for example, must reference all misspelled words. We extended our referencing mechanism to combine the paths of all selected objects into a single tree by combining paths with common prefixes into the same subtree. Some nodes in the tree can also represent ranges of indices rather than individual nodes for each index. This would be appropriate for the 3 consecutive days highlighted in the calendar application in figure 2.

Capturing drawing calls

Having a representation for pointers that is application-independent, our general tools need access to the drawing behavior of the application. We exploit a technique pioneered in [2] of using specialized drawing classes. In most modern windowing systems, the actual mechanisms for drawing into a window are isolated from the application by an abstract interface. This may be object-oriented as in the case of Java's `Graphics` class or SubArctic's `drawable` class, or it may be pseudo-object-oriented as in the Microsoft Windows device context.

By substituting our own class for the standard drawing class, we capture all of the calls that the application's rendering code may generate. This provides us with complete control over the application's drawing behavior without interfering with the application's normal implementation. We must, however, impose our first requirement on the application - it must be capable of rendering its entire 2D surface rather than just the portion in the visible window. We need access to the entire surface so that we can extract the global pointing information for showing where scrolled-away items can be found. The inefficiencies of requiring an application to render everything rather than just the portion visible in the window can be mitigated by visible bounds techniques. If the application will interrogate the clipping rectangle, it can optimize what sections of the surface should be drawn. By setting the clipping rectangle to the size of the whole surface we can capture the entire drawing area. By invoking an application's Redraw or Paint method with one of our specialized drawing classes, we can capture any information about the application's visual presentation while imposing minimal changes on the application's architecture.

Capturing the semantic map

Simply capturing drawing calls is not sufficient to implement the highlighting techniques because they contain no information about which drawn objects are mapped to which semantic objects in the rendering traversal tree.

To accommodate this need we have added three calls to the `drawable` class. They are:

```
GroupStart(String Name, String Type)
GroupStart(int Index, String Type)
GroupEnd()
```

As part of its rendering algorithm, an application is expected to call `GroupStart` before drawing information from a particular object. The object can be identified by a string name or by an index depending on the `GroupStart` method used. In addition to identifying the object, a string type for the object can also be supplied. The type information is not used by the highlighting mechanism but is helpful for other surface-based techniques that we are developing. After an object is drawn, `GroupEnd` is called. The calls to `GroupStart` and `GroupEnd` can be nested to any depth. Example pseudo-code for drawing the calendar application on a surface S is shown in figure 5.

```
For each month M
{  S.GroupStart(M.name,"Month");
   S.draw the month name
   S.draw the days of the week
   For each day D in M
   {  S.GroupStart(D.date,"Day");
      S.draw day rectangle
      S.draw day border
      S.draw the date
      S.GroupEnd();
   }
   S.GroupEnd();
}
```

Figure 5 - Example Calendar Rendering Code

These calls provide all of the information necessary to capture the rendering traversal tree. Including `GroupStart` and `GroupEnd` calls in the rendering code is the second major implementation requirement that we impose on an application. We do not consider this excessive, however, because such calls are relatively straightforward to add to existing rendering code with reasonable structure. It is far easier, for example, to include the `GroupStart/GroupEnd` calls than to implement Cut/Copy/Paste or OLE objects. These calls provide a link between surface geometry and the underlying model structure. A major part of our future research will leverage this surface/model mapping to support more powerful agent behavior than simple pointing.

Implementing highlighting

Based on the machinery described above, we can define a special `HighlightSurface` class which is a subclass of `drawable`. (We are using SubArctic. Similar techniques will work in most other object-oriented windowing systems.) The `HighlightSurface` class passes all drawing routines on to `drawable` with the exception of the calls to set colors and the `GroupStart` and `GroupEnd` calls. The purpose of `HighlightSurface` is to draw the application information using blended highlights. The global pointing widgets and the agent identifiers are handled separately. In order to draw correctly, the `HighlightSurface` class must have a reference to all objects being highlighted, a blending color, and a highlight intensity.

Whenever the application redraws itself as part of its interactive behavior, it is given a `HighlightSurface` instead of a `drawable`. At each `GroupStart` call the surface will compare the name or index to the reference tree to determine if this group is selected or not. The selected state and position in the reference tree before `GroupStart` was called are pushed onto a stack. If the group is selected, then the current draw color, as requested by the application, is passed on to the `drawable`. If the group is not selected the draw color is blended with the blend color as described above. The resulting blended color is passed to the `drawable` instead of the application's requested color. When `GroupEnd` is called the stack is popped and the color settings restored to what they should be given the selected state of the enclosing group. The application rendering code never knows that highlighting has been done. This technique produces the drawings in figure 1. This blended highlight technique has very minimal impact on the drawing speed of the application.

Capturing the selected region

In order to draw the agent identifiers and to implement the global pointing widgets, we need to compute a geometric region on the surface for all selected objects identified by a given reference. Note that a reference may point at multiple objects, and that a given object may appear on the surface in multiple places.

As with blended highlights, we create a subclass of `drawable` called `SelectedRegion`. The `SelectedRegion` class does not pass any of its calls on to `drawable`. As with `HighlightSurface`, the `SelectedRegion` must be given a reference tree. To compute the region geometry corresponding to the reference tree, the Paint or Redraw method on the application is called with a `SelectedRegion` for its `drawable`. The clipping rectangle associated with `SelectedRegion` is the size of the whole surface. The `SelectedRegion` initializes a `Region` object to empty. The `Region` object collects the bounding region for all selected groups. Each `GroupStart` is compared with the reference tree to determine selectedness. Whenever a drawing call is encountered within a selected group, that drawing primitive's bounding region is unioned into the `Region` object. Any drawing call encountered in an unselected group is ignored. At the end of this process the `Region` object contains the bounding region for all selected groups.

Executing the redraw code for an entire surface just to compute the bounding region for a single object can be very inefficient. For this reason the `GroupStart` method returns `true` if the named or indexed group is of interest and `false` otherwise. When drawing into the `HighlightSurface`, `GroupStart` always returns `true` because all groups need to be drawn. The `SelectedRegion`, however, returns `false` for any group that does not contain selected objects. The pseudo-code of figure 5 can be optimized to that shown in figure 6. This structure will minimize the cost of computing regions.

```
For each month M
{  if (S.GroupStart(M.name,
          "Month"))
   {
      S.draw the month name
      S.draw the days of the week
      For each day D in M
      {  if(S.GroupStart(D.date,
               "Day"))
         {
            S.draw day rectangle
            S.draw day border
            S.draw the date
         }
         S.GroupEnd();
      }
   }
   S.GroupEnd();
}
```

Figure 6 - Optimized Rendering Code

One problem with computed highlight regions is that they may move as the application's data is changed. We can track such changes in the same way that display changes are handled. When application data changes, the rendering system will damage the changed area causing the windowing system to request that those areas be repainted. Similarly such damage information can be used to recompute reference regions so that they are up to date.

Once a region has been computed for a given agent, the region is expanded by a specified number of pixels and its border drawn in the agent's color to produce the agent indicator. This is done for all agents after the highlight drawing has been done. Again the application knows nothing about any of this being done other than the successive calls to its drawing routine.

BOOKMARKS FRAME DEMO

In order to test the efficacy of our architecture we assigned students in our lab to implement 4 different applications using the standard SubArctic toolkit. They were: a drawing application, an appointment calendar, a tabular data presentation and editing tool, and a map-based planning application. Those implementing the applications did not know about the highlighting algorithms.

We then built a simple framework that supports multiple bookmarks, shown in figure 7. The essence of the framework is that any object selected by any application embedded in the framework can be saved as a bookmark or added to an existing bookmark. The multiple bookmarks serve as surrogates for multiple agents, which we did not implement.

The test was to take the existing applications and embed them in the bookmark framework, thus adding the multibookmark pointing feature to any application. Because we require some modifications to application code, we wanted to test how long it took to modify an application to work within the framework. Once the framework itself was debugged, each application took less than one man-day to embedding in the framework. The last application took less than 30 minutes to convert.

The framework required that the application implement four new methods:

```
interactor getSurfaceInteractor();
Rectangle getViewableArea();
TRef getModelRef();
void scrollSurfaceInteractor
        (int x, int y);
```

Because we were embedding arbitrary applications into the framework, the first two methods are required so that the framework can separate controls and buttons from the working surface of the application. We decided that applications would use their own selection mechanisms which the bookmark facility exploits by requesting a reference to the currently selected object. The fourth method allows global pointing widgets to force the application to scroll to a particular area in response to user inputs. Such coercion only occurs at the user's request.

The bookmark framework provides controls for the blend color and highlight intensity. The user has direct control over the highlight intensity using a scroll bar. The bookmark framework substitutes a `HighlightSurface` whenever the application is drawn and uses the blend color and intensity to provide the highlighting. When necessary, a `SelectedRegion` is used to compute the region for each bookmark. Each bookmark border is then drawn clipped to the viewable area rectangle. This implement the highlighting required for the bookmarks.

In addition the bookmark frame provides three global pointing widgets. By projecting the bookmark regions along X and Y, we can define widgets that provide highlights along the vertical and horizontal axis. Greenberg reports that these are not very effective for collaborative use; however, the vertical version can implement the "wear marks" techniques of Hill and Hollan [5]. We find that when the total surface is very narrow in one dimension, the global pointing widget of the other dimension is very effective. In addition to these projected widgets there is a rectangular "radar view" [4] which represents the entire surface area in miniature with the highlight regions drawn in their agent identification

colors. Clicking on any of the global pointing widgets will force the application to scroll so as to make the indicated area visible. This supports navigating directly to highlighted areas. All of these global pointing widgets use the selected regions calculated from the drawing surface. These widgets are also independent of bookmarks and can be used by any agent needing global pointing.

SUMMARY

We have defined a mechanism for visually highlighting any object or objects in any application using our blended highlight technique. We also identify the responsible agent using our colored region borders. Selected regions calculated from the drawn images also drive the global pointing widgets. The heart of our algorithms are in the model/surface mapping information provided by the `GroupStart` and `GroupEnd` calls. These deliver semantic information to the surface where external agents can support the end user's work.

REFERENCES

1. Bederson, B. and Hollan, J. Pad++ A Zooming Graphical Interface For Exploring Alternate Interface Physics. *UIST '94,* (Nov 1994), 17-26.
2. Edwards, W. K., Hudson, S. E., Marinacci, J., Rodenstein, R., Rodriguez, T., and Smith, I. Systematic Output Modification in a 2D User Interface Toolkit. *UIST '97,* (Oct 1997).
3. Fischer, G., McCall, R., and Morch, A., Design Environments for Constructive and Argumentative Design. *CHI '89,* (May 1989), 269-279
4. Gutwin, C., Roseman, M., and Greenberg, S. A Usability Study of Awareness Widgets in a Shared Workspace Groupware System. *CSCW '96,* (Nov 1996), 258-267).
5. Hill, W. C., and Hollan, J. D. Edit Wear and Read Wear. *Human Factors in Computing Systems (CHI '92),*(May 1992), 3-9.
6. Olsen, D. R., Bookmarks: An Enhanced Scroll Bar. *ACM Transactions on Graphics.* (July 1992).
7. Olsen, D. R., Ahlstrom, B., Kohlert, D., Building Geometry-based Widgets by Example. *CHI '95,* (May 1995).
8. Olsen, D. R., and Rodham, K. Smart Telepointers: Maintaining Telepointer Consistency in the Presence of User Customization. *ACM Transactions on Graphics* (July 1994)

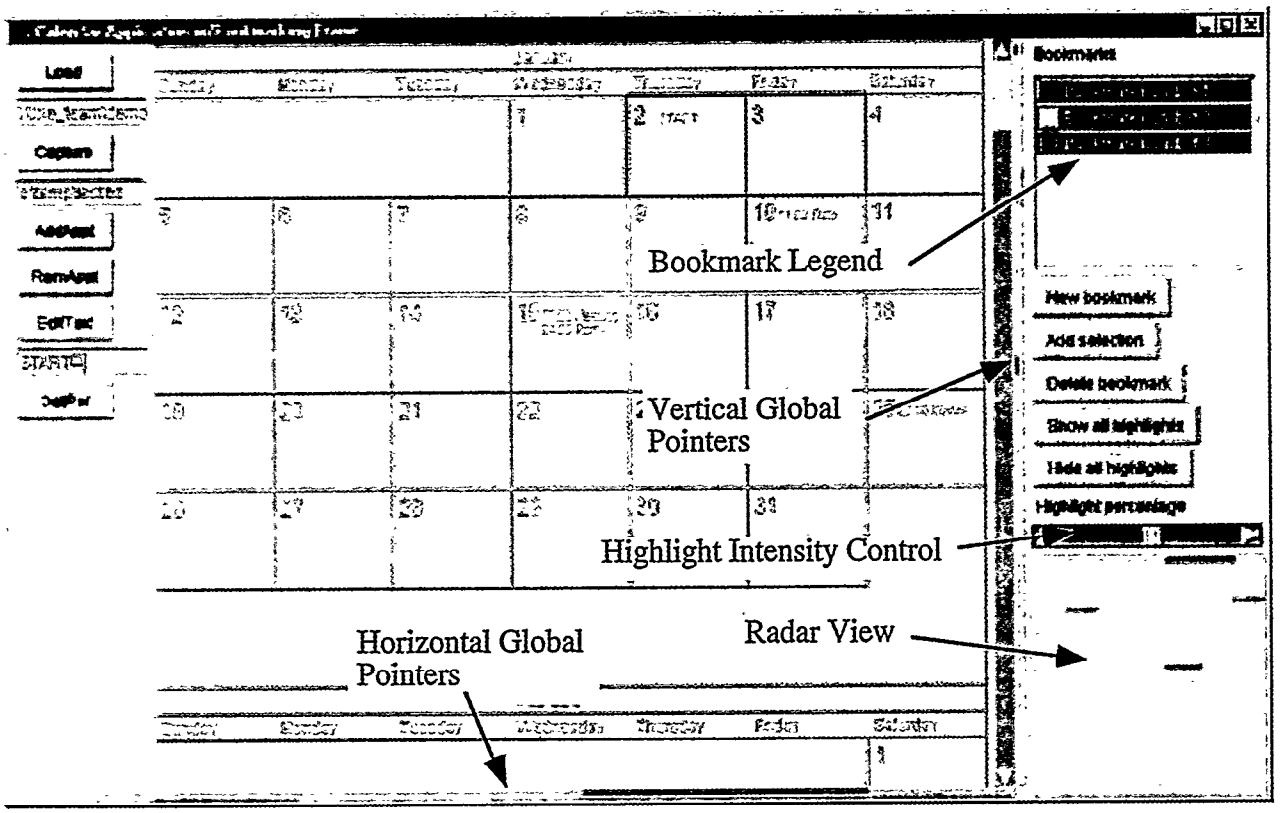

Figure 7 - Bookmark Framework