Debugging Lenses: A New Class of Transparent Tools for User Interface Debugging

Scott E. Hudson
Human Computer Interaction Institute
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213-3891
E-mail: hudson@cs.cmu.edu

Roy Rodenstein
Ian Smith
Graphics Visualization and Usability Center
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
E-mail: {royrod, iansmith}@cc.gatech.edu

ABSTRACT

The visual and event driven nature of modern user interfaces, while a boon to users, can also make them more difficult to debug than conventional programs. This is because only the very surface representation of interactive objects — their final visual appearance — is visible to the programmer on the screen. The remaining "programming details" of the object remain hidden. If the appearance or behavior of an object is incorrect, often few clues are visible to indicate the cause. One must usually turn to text oriented debugging techniques (debuggers or simply print statements) which are separate from the interface, and often cumbersome to use with event-driven control flow.

This paper describes a new class of techniques designed to aid in the debugging of user interfaces by making more of the invisible, visible. This class of techniques: *debugging lenses*, makes use of transparent lens interaction techniques to show debugging information. It is designed to work *in situ* — in the context of a running interface, without stopping or interfering with that interface. This paper describes and motivates the class of techniques, gives a number of specific examples of debugging lenses, and describes their implementation in the subArctic user interface toolkit.

KEYWORDS: Interactive Debugging, Lens Interaction Techniques, Dynamic Queries, Context-Based Rendering, User Interface Toolkits, subArctic, Java™.

1. MOTIVATION AND BACKGROUND

Writing complex user interfaces can be difficult. For example, even experienced programmers make mistakes in specifying complex constraints [Myers91]. The difficulty of debugging such problems can be exacerbated by the graphical nature of the interface — objects and relationships within the interface are seen only in their final rendered form. Hence, the great usefulness of the visual interface to the end user can come at the price to the programmer. If the appearance or behavior of an object is incorrect, this can sometimes be readily seen, but often few clues are visible to indicate the cause. For example, if an object does not appear on the screen when it should, it could have been given a very small size, it could have not been added to the interactor tree, it could be marked as non-visible, it could be positioned off screen or outside clipping bounds, or it could be obscured by another object. However, none of these causes presents visual cues, and so the programmer must typically resort to text oriented techniques (debuggers or simply print statements) which are separate from the flow of the interface, and often cumbersome to use with event-driven control flow.

This paper introduces a new class of debugging aids designed to be used in concert with a running interface, without stopping it or interfering with its normal operation. These techniques are based on the use of *Magic Lenses*™ [Bier93, Ston94, Bier94]. Magic Lenses are transparent interface elements which are designed to be moved over other interface elements and modify the display of those elements in some way. In general lenses can modify the appearance of objects in arbitrary ways — some lenses add information to the display, some remove all but selected information, and some make more arbitrary changes to the display. In general, lenses can be seen as a form of focus plus context visualization technique [Furn86, Mack91, Rao94] which displays information relevant to a particular task (in our case a debugging task) in the context of a "normal" display.

In general, lenses can reduce screen clutter and increase the usefulness of specialized or task specific information by focusing a greater level of detail on particular screen areas. Further, because

This work was supported in part by a grant from the Intel Corporation, and in part by the National Science Foundation under grants IRI-9500942 and CDA-9501637.

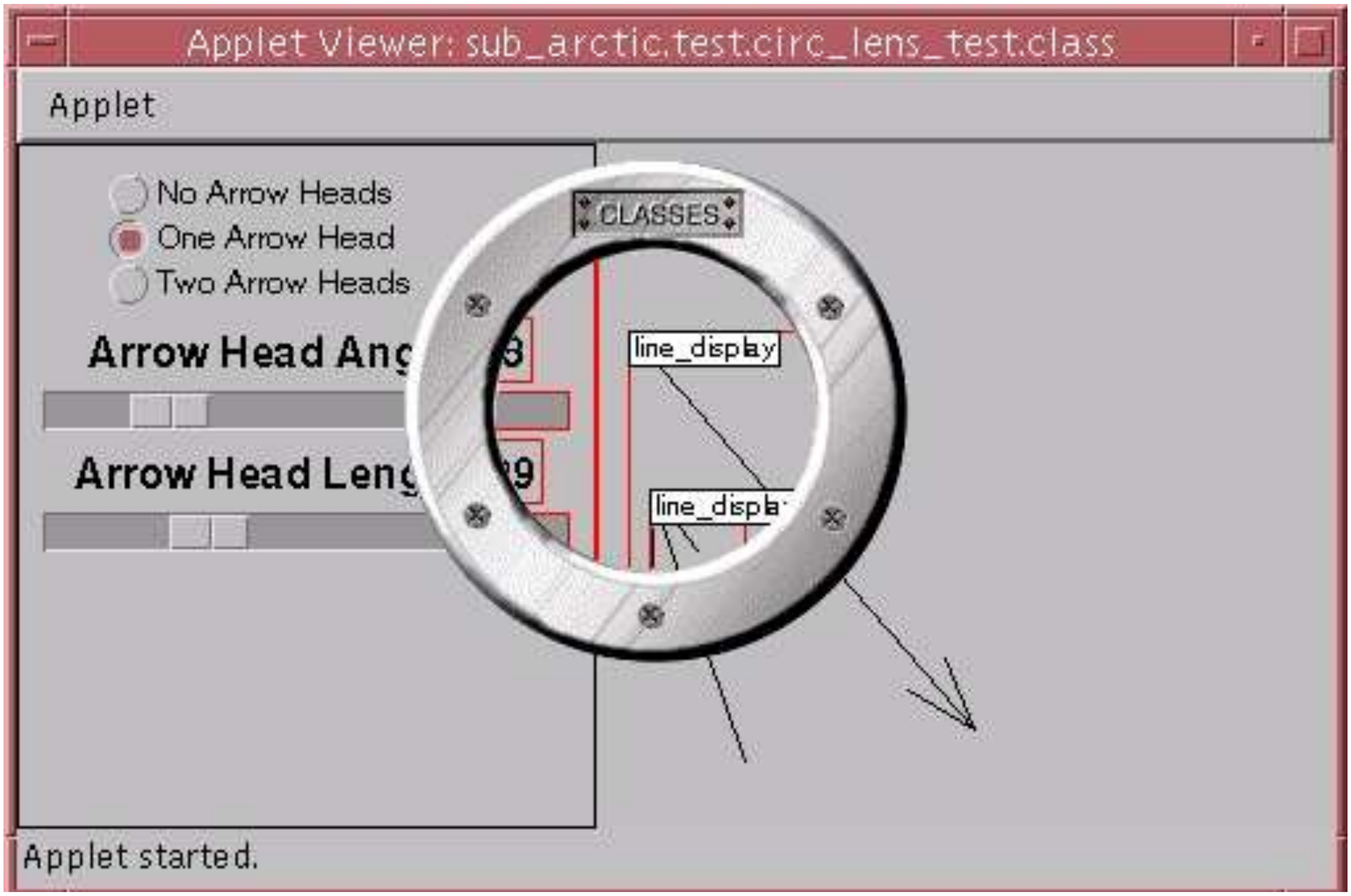

Figure 1. A simple circular debugging lens which displays a bounding box around underlying interactors, as well as a name tag with their class name.

they are transparent they do not obscure the underlying content, they can display information in situ — in the underlying area, and in the existing context — so that the user is not required to make the switch to an extraneous mental context, nor to divert attention to a different screen area in order to see the results. Lenses also allow click-through interaction with the underlying content (via what were called *toolglasses* by the original inventors). Finally, lenses can display naturally graphical information which would not be practical to display as text. For example, an application described in [Edwa97] (and built with the subArctic lens infrastructure) shows the face and contact information of the person who last modified part of a shared document. Similarly, Figure 1, provides a graphical display of the bounding box of each interface element under the lens.

This paper describes a new class of debugging tools based on lenses. Debugging lenses operate over the top of a running interface. This allows them to preserve code integrity and interactive flow by obviating the addition of debugging print statements, or control flow interruptions.

Debugging lenses are easy for the programmer to use. In the subArctic toolkit [Huds96, Huds97], they can be added to interface with only a one-word change to the source code. Debugging lenses update dynamically, so the programmer can be sure that the information displayed is up to date; and lenses' click-through interaction and transparency permit debugging lenses to be used fully in context, directly over the running interface of the real application.

The remainder of this paper contains a discussion of debugging lenses. Section 2 describes the technique in general, while Section 3 considers several examples of debugging lenses that we have implemented. Section 4 provides a description of the architecture and implementation of our debugging lenses, and finally, Section 5 provides some brief conclusions.

2. THE TECHNIQUE

As indicated above, debugging lenses possess the general capability to modify the appearance of the underlying interface, add information, or focus on particular aspects of debugging. Figure 1 shows a basic circular debugging lens which draws a small

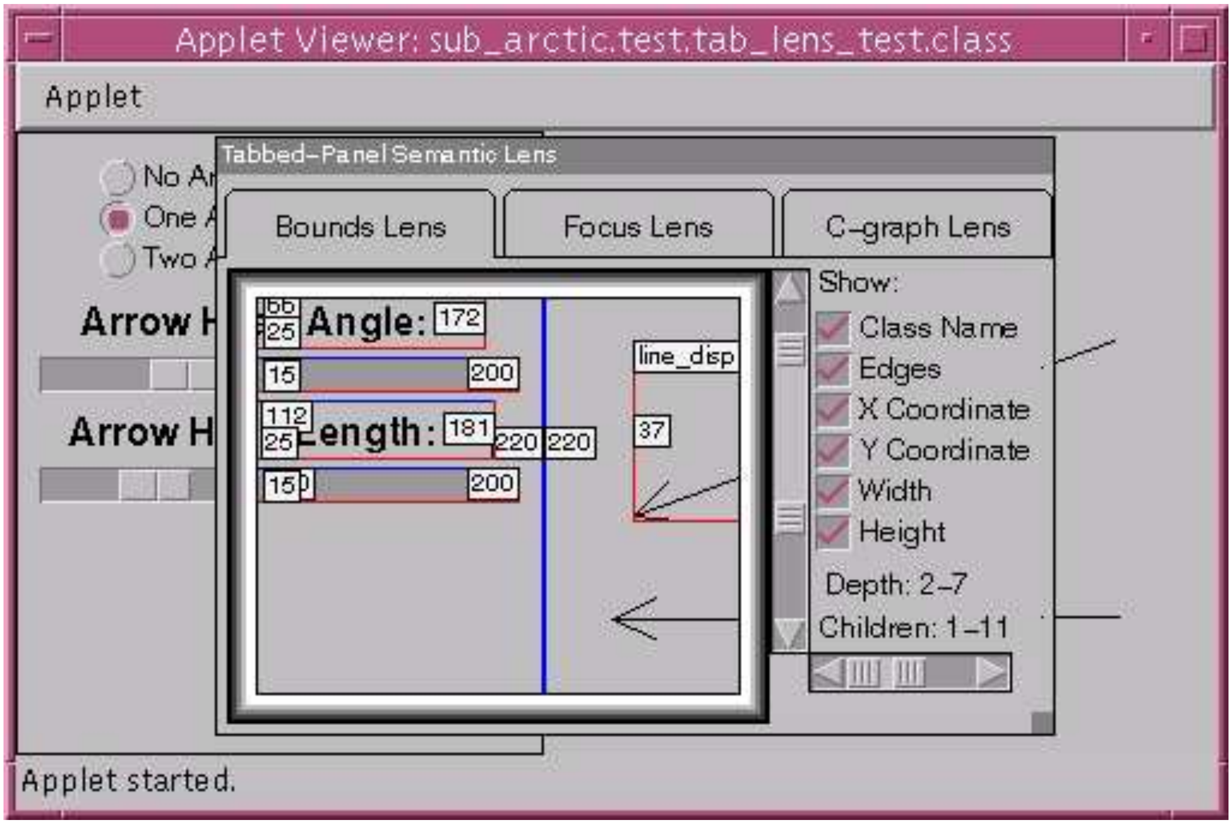

Figure 2. The depth- and child-number-bound lens. Note that the arrow at bottom is not in the selected range and does not display debugging information.

name tag showing underlying interactors' class name, as well as a bounding box showing its spatial extent. As the lens is moved its display updates dynamically to encompass new objects underneath it, while the area the lens was previously over is displayed in its normal form.

Debugging lenses require extra drawing, and hence could conceivably cause performance problems. However, the area of additional drawing is limited in size, and our initial experience with them has not uncovered significant performance problems. In almost all cases interface responsiveness has not been noticeably slower when debugging lenses are used. Maintaining responsiveness aids in the realistic testing of the interface through the lens.

Debugging lenses are highly extensible. Their implementation uses a flexible lens and layer infrastructure provided by the subArctic user interface toolkit. This allows them to be easily combined with existing or novel interactors. For example, taking advantage of the fact that debugging lenses provide a convenient on-screen platform for debugging interactions, the lens in Figure 2 goes beyond the basic lens from Figure 1 to include a side-mounted tool palette, allowing for extended user control. Whereas the first lens draws only a small tag with the class name of each interactor and a bounding box around it, the second lens' controls let the user turn various additional information displays on and off, such as interactors' x and y coordinates, width and height, class name, and bounding box, with constrained edges shown in blue and unconstrained edges shown in red.

A sample use case for these lenses would be if one has mistakenly created an interactor (either at initialization time or while the program is running) of zero height and/or width. In this case the lenses would show the class name tag for the interactor, alerting us to its existence and location, as well as to the problem, its zero size. As another typical example, if the interactor were placed into an incorrect stacking layer, which would in most cases hide it from view, the programmer might wonder whether the code had left out the interactor completely, whether its position or size were incorrect, or whether a constraint error was causing it to be invisible, among other possibilities. Debugging lenses would reveal the rogue interactor's presence and show any or all of the standard information about it.

An additional capability of the lens shown in Figure 2 allows it to provide more selected information. The depth and child-number range

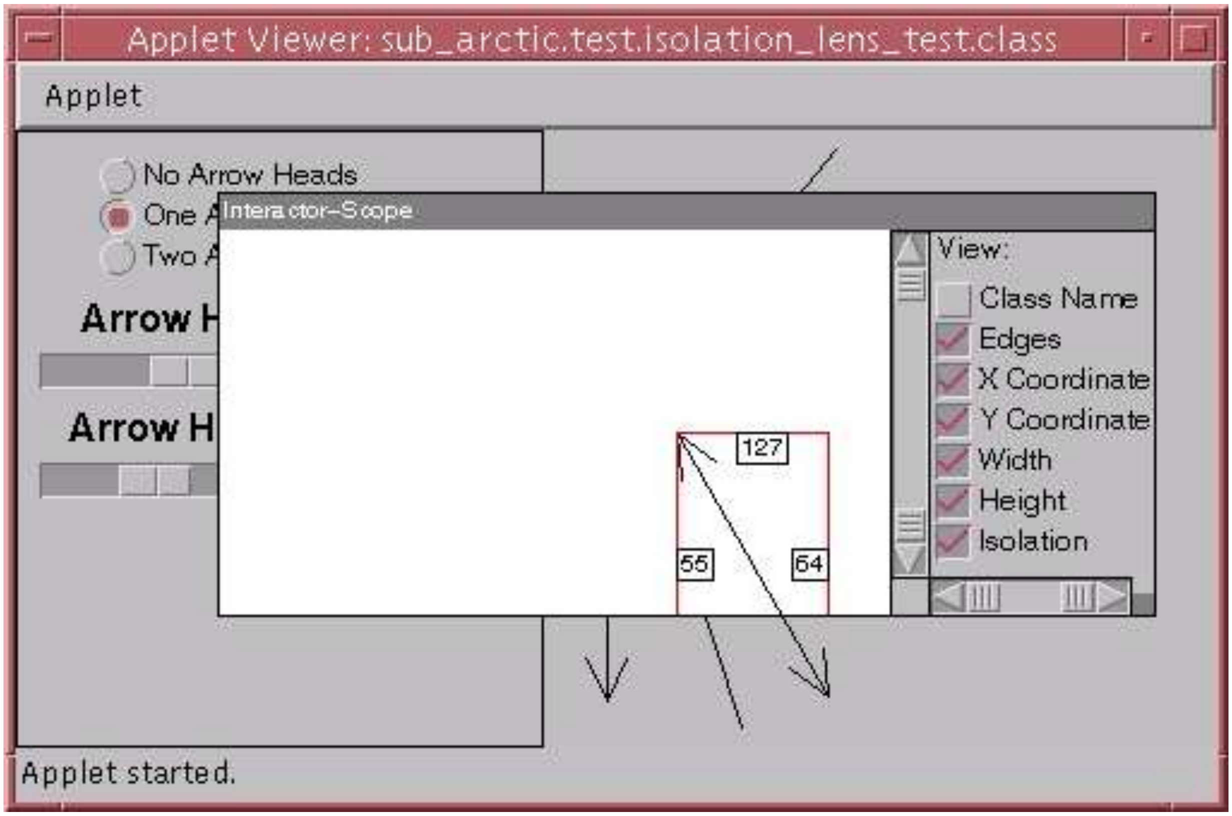

Figure 3. The isolation lens. Note that several arrows overlap, but only the one we have focused on using crosshairs (see grabber lens in Figure 5) is displayed in the lens.

sliders of the lens (mounted on the left and bottom of its tool palette) can further be used to isolate interface elements and study them individually. The sliders may be used to select interactors at a particular range of depths in the interface's interactor tree, or a range of interactors selected by ordinal number within a child list. The lens would then display its added information only for those interactors falling within the specified ranges. In this manner, a single interactor or a subset of interactors in the application may be selectively included or excluded. This allows better visibility of objects of particular interest in situations where interface elements are densely packed or occlude one another.

Movement of the depth and child-number sliders cause the lens image to be updated dynamically. As a result, these controls provide a form of dynamic query [Ahlb92, Ahlb94, Fish95]. This, combined with the ability to select display components, as well as resize, and reposition the lens, provides a very flexible tool for focusing on the specific information needed for a specific debugging situation. The lenses' ability to let user interaction pass through to the underlying interface also makes them further suitable for use in situ. Being able to see information directly on the interface and interact normally with it is extremely useful in symptom detection, problem diagnosis, test-case generation, and demonstration of repeatability.

3. EXAMPLES

In Figures 1 and 2 we have seen a version of the original debugging lens display that was distributed with public releases of the subArctic toolkit (and now in use by a number of users outside our group), as well as the most common lens used in our current development release. These illustrate many of the basic concepts of debugging lenses, and have been effective debugging aids in practical use (by our group and others). In this section we consider several more examples of this class of technique in order to illustrate more of the possibilities of the design space.

In addition to the dynamic query capabilities shown in Figure 2, a second method that debugging lenses can employ to empower the user in the interface debugging process is limiting a lens's focus at the atomic level of specific, selectable interactors. Figure 3 shows an isolation lens whose focus is limited to a particular set of interactors chosen by the user.

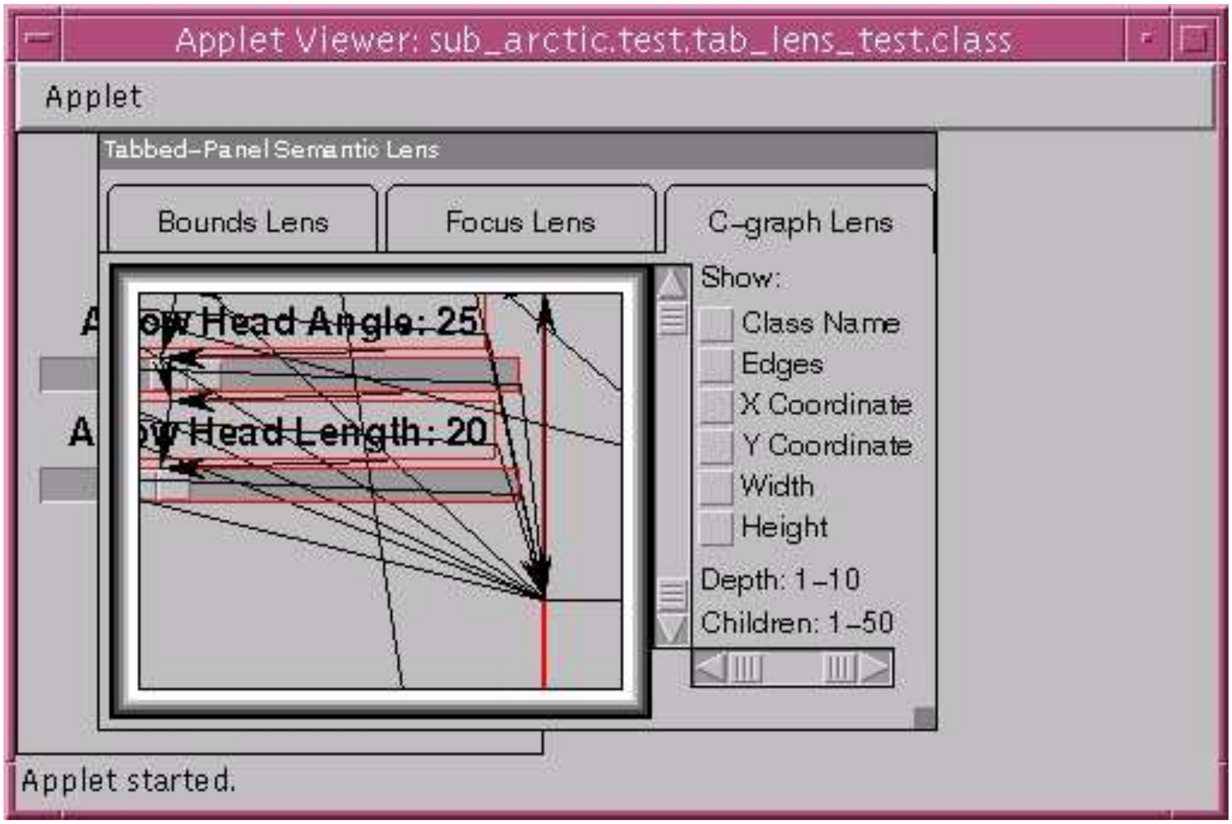

Figure 4. The constraint-graph lens, showing constraint sources and targets.

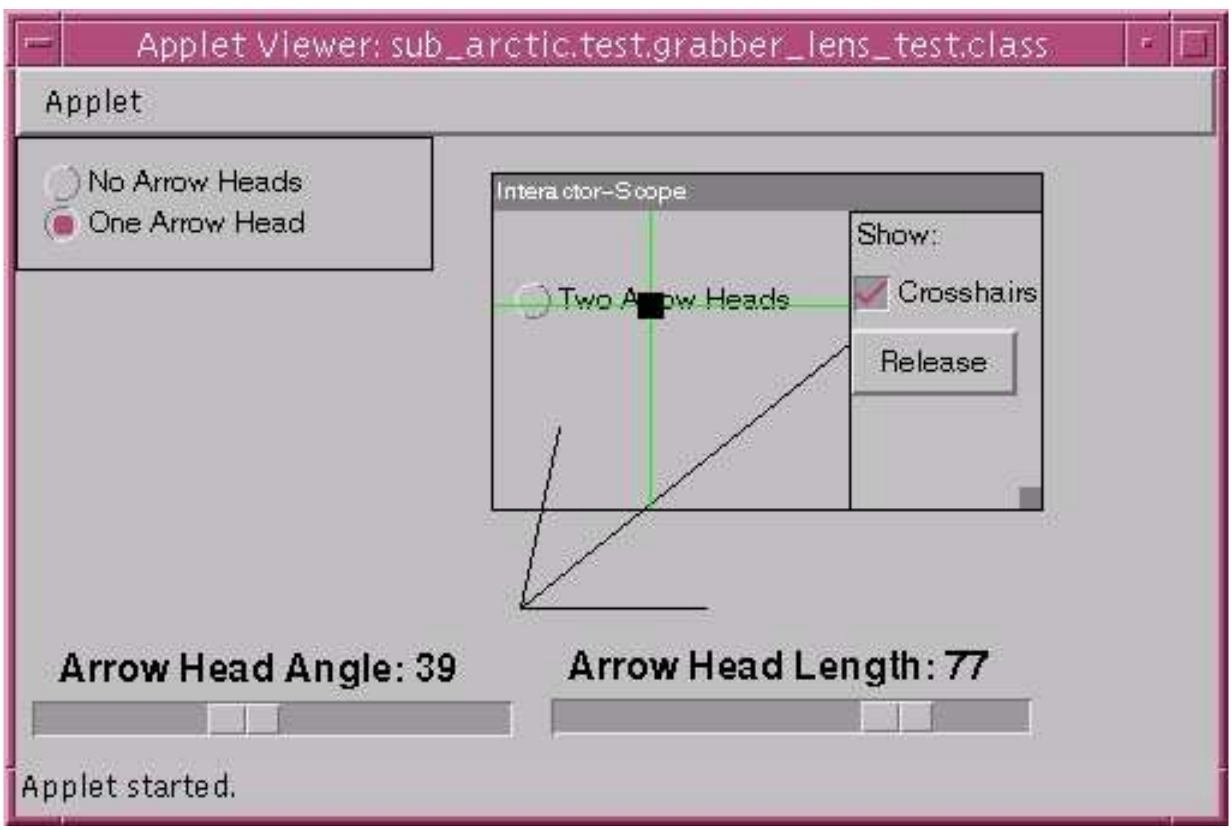

Figure 5. The grabber lens allows you to grab interface elements using the crosshairs, and reposition them.

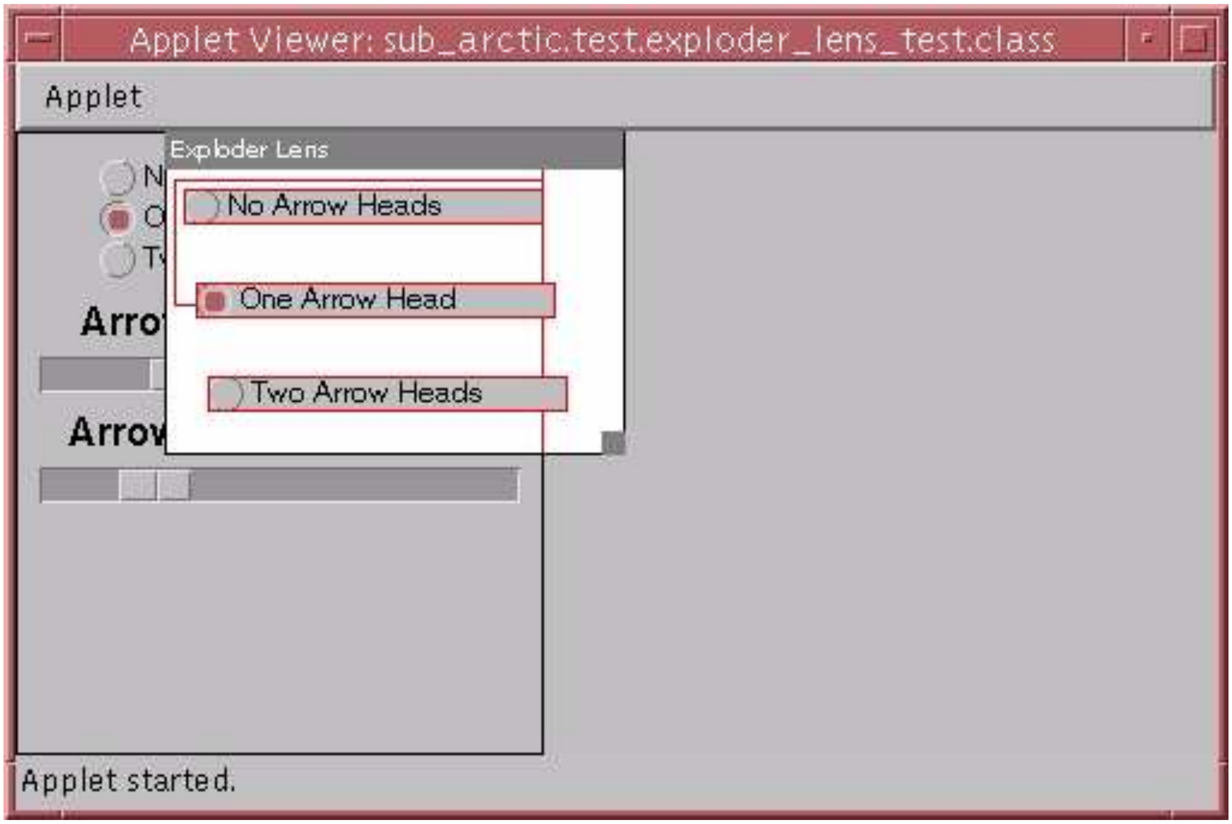

Figure 6. The child-exploder lens spreads out children to reduce clutter in the lens view of the interface.

These interactors are drawn in their normal position and the background of the lens is opaque, visually isolating the interactors in the focus set from the rest of the interface in order to avoid clutter, and allow control of the lens's display of debugging information for the set. The drawing isolation can be toggled on and off, so that context with the rest of the underlying interface can be reestablished at any point, without losing the focus set of interactors.

A further advantage of this focus model is that, once set, the lens can display information about interactors not only through drawing over the interface, but also by using the anchored palette to its side to display information. This information remains current whether the lens's transparent area is over a focused-on interactor or not. If at some point during runtime the interactor that has the focus disappears, this lens will make it clear whether the interactor has become of zero size, has been hidden under an opaque interactor, or has actually been removed from the interface.

As indicated above, debugging lenses are easily extensible. Figure 2 above shows an example of this, an interactor that places three lenses into a tabbed-folder parent. This allows the user to switch between the different capabilities provided by each lens and use the most appropriate one at all times; each lens preserves its state, so no context or effort is lost in switching between lenses.

Figure 4 shows another tab of this tabbed-panel interactor, which contains a constraint-graph lens. This lens displays arrows depicting constraints on interactors, such as the horizontal arrow spanning the width of the parent panel on the left side. This lens is useful for visualizing constraints, and although somewhat cluttered in this full view, can also be focused using the range sliders to obtain a more targeted view.

A different type of debugging lens appears in Figure 5. This is a grabber lens. It can pick up interface elements under the crosshairs, reposition them, and drop them back into the interface. This can be useful both for debugging of an interface's visual design and for testing of interactors in different locations and under different parents. This can bring to light certain problems that a static interface might not indicate so readily.

Finally, Figure 6 shows a child-exploder lens. This lens explodes the representation of a parent interactor's children (that is, a subtree, or interior node, in the interactor tree. The children are spread out, that is, the vertical and horizontal space between them is increased, so that a tight layout or overlapping interactors can be viewed with minimal interference. This is an example of a specialized lens that can be applied in particular situations to handle particular visualization needs.

Although a number of different, and very useful examples have been shown here, it should also be clear that these particular techniques are only the beginning of a larger tool suite that can be developed within the framework of debugging lenses.

4. ARCHITECTURE AND IMPLEMENTATION

User interfaces written using the subArctic toolkit utilize a structure typical in user interface construction: they maintain a 'root' interactor and build an n-ary tree of child interactors with this root interactor at the top as shown in Figure 7. Rendering of the user interface is performed through a recursive traversal of the interactor tree, passing a *drawable* object down the tree starting from the root. The drawable object maintains current drawing state (such as the current clipping rectangle) and provides operations for producing output on a drawing surface. Each interactor is responsible for producing its own output. The non-leaf interactors draw themselves then continue the traversal of their children, while leaf interactors simply drawing themselves and return.

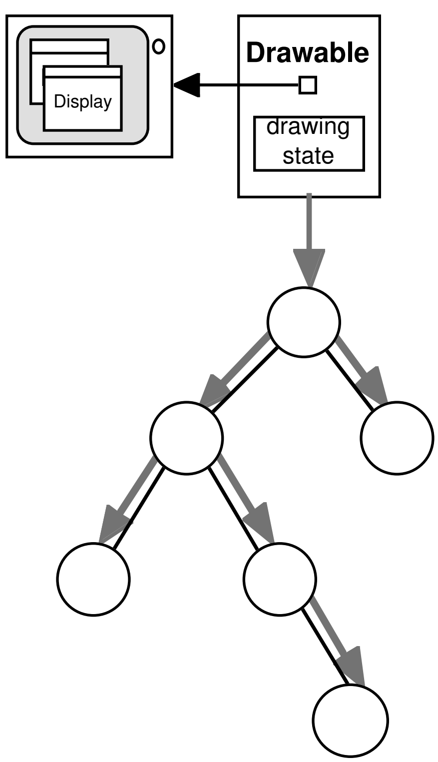

Figure 7. A representation of the typical structure of a user interface. Drawing of the interface is performed as a recursive traversal of the interactor tree passing a drawable object which maintains drawing state and provides access to a drawing surface.

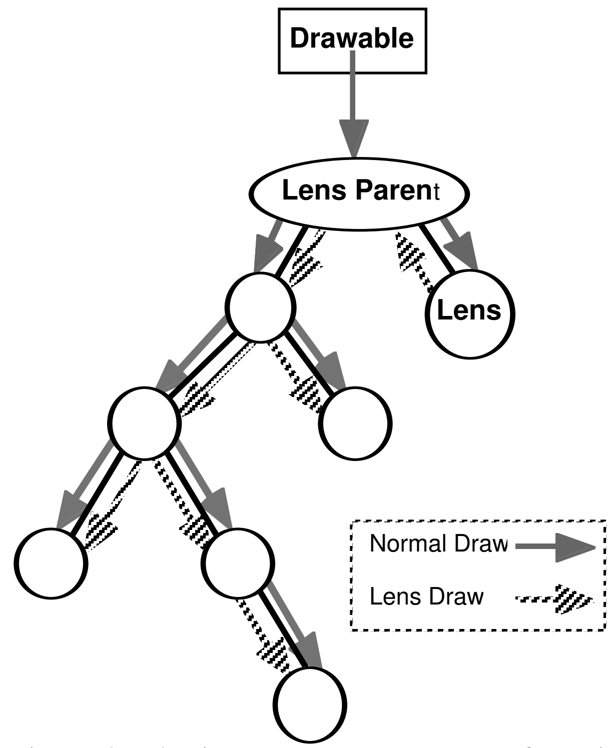

Figure 8. The interactor tree as set up for using lenses. The lens's redraw causes a second drawing pass through the interactor tree.

To implement lenses in subArctic, a special lens parent is inserted above the normal top interactor, as depicted in Figure 8. This parent ensures that damage resulting from modifications to interactors appearing under the lens is communicated to the lens so that it may redraw itself. This ensures that the most current state is always displayed. No modifications beyond this change of the root interactor are necessary to utilize lenses; indeed, a one-word change in the main source file of a program (e.g., changing "interactor_applet" to "debug_interactor_applet") allows the programmer to take advantage of lenses to debug their user interface. The use of the special lens parent is invisible to the program, and no other code need be modified. When used in this manner, debugging lenses may be shown or hidden by the user through a special keypress-mouse click combination. In the current system, this brings up the tabbed-panel lens shown in Figures 2 and 4 (which is currently being extended to include additional lens types).

The lens object placed under the lens parent will receive a redraw request as a part of the normal recursive redraw process. This lens object acts on this request by doing a second specialized drawing traversal starting at its parent (taking care not to recursively draw itself with this traversal). For the most common case of additive lenses, the lens drawing is simply done over the top of the existing object drawing. For lenses that completely replace the area under them (e.g., the isolation lens shown in Figure 3), the lens simply clears its background then performs the drawing traversal.

The drawing traversal performed by the lens is implemented using a general traversal mechanism provide by the subArctic toolkit. This traversal mechanism works on the basis of a parameterized top-down tree walk. The walk is controlled by a predicate object which determines at each node whether the recursive traversal should end at that point, or continue. The predicate operates both on the basis of the node it visits, and using a special state object passed down the traversal. This state object typically contains a drawable giving access to a properly clipped portion of the screen, and encodes things like the current tree level and child number, as well as the ranges that drawing should take place at.

The predicate first tests the object position to insure that its drawing would not be totally discarded by clipping (i.e., does a trivial reject test), then performs semantic tests such as verifying that the interactor is within the user requested ranges. If the predicate indicates that a node should be visited, a special action object is invoked with both the visited interactor and the state object passed as parameters. This action object performs the drawing action for the lens.

In order to step deeper into the traversal the state object is transformed from a state suitable for the parent object into a state object suitable for use by a candidate child interactor (this might for example involve incrementing the current tree depth). Finally, the traversal process is repeated recursively with the newly transformed state information.

The code for the standard bounds lens (a standalone version of the lens shown in Figure 2) is implemented in just over 1300 lines of Java code. Additional lenses are of similar size (although subclassing can reduce their size in some cases).

5. CONCLUSIONS

The debugging lenses introduced in this paper provide a new class of debugging tool for user interfaces. These tools can significantly help debugging by making information about interface objects which would have been invisible, visible to the programmer. Further they allow this information to be presented in a selective and focused manner that provides data of interest in the context of the interface. Finally, debugging lenses have the advantage that they can operate in situ on any interface, with minimal disruption to the workings of the interface.

One limitation of the current lens infrastructure is that, while permitting composition of lenses via overlap, it does not have particular capabilities to attempt to intelligently arbitrate this composition. We are currently working on a new and simplified lens infrastructure which will allow for more semantically meaningful and sophisticated compositing techniques.

REFERENCES

[Ahlb92] Ahlberg, C., Williamson, C., Shneiderman, B., "Dynamic Queries for Information Exploration: An Implementation and Evaluation", *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems*, 1992, pp. 619-626.

[Ahlb94] Ahlberg, C., Shneiderman, B., "Visual Information Seeking: Tight coupling of Dynamic Query Filters with Starfield Displays", *Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems*, 1994, pp. 313-317.

[Bier93] Bier, E.A., Stone, M.C., Pier, K., Buxton, W., DeRose, T.D., "Toolglass and Magic Lenses: The See-Through Interface", *Proceedings of ACM SIGGRAPH '93 Conference on Computer Graphics and Interactive Techniques*, 1993, pp. 73-80.

[Bier94] Bier, E., Stone, M.C., Fishkin, K., Buxton, W., Baudel, T., "A Taxonomy of See-Through Tools", *Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems*, 1994, pp. 358-364.

[Edwa97] Edwards, K., Mynatt, B., "Timewarp: Techniques for Autonomous Collaboration," to appear in *Proceedings of ACM CHI'97 Conference on Human Factors in Computing Systems*, 1997.

[Fish95] Fishkin, K., Stone, M., "Enhanced Dynamic Queries via Movable

Filters”, *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, 1995, pp. 415-420.

[Furn86] Furnas, G. W., "Generalized Fisheye Views", *Proceedings of ACM CHI'86 Conference on Human Factors in Computing Systems*, 1986, pp. 16-23.

[Huds96] Hudson, S., Smith, I., “Ultra-Lightweight Constraints”, *Proceedings of the ACM Symposium on User Interface Software and Technology*, 1996, pp. 147-155.

[Huds97] Hudson, S., Smith, I., “The subArctic User Interface Toolkit Home Page",, Web document available at: http://www.cc.gatech.edu/gvu/ui/sub_arctic

[Macki91] Mackinlay, J. D., Robertson, G. G., Card, S., K., "The Perspective Wall: Detail and Context Smoothly Integrated", *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*, 1991, pp. 173-179.

[Myer91] Myers, B., “Graphical Techniques In a Spreadsheet For Specifying User Interfaces”, *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems,*, 1991, pp. 243-249.

[Rao94] Rao, R., Card, S. K., "The Table Lens: Merging Graphical and Symbolic Representations in an Interactive Focus+Context Visualization for Tabular Information", *Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems,* 1994, pp. 318-322.

[Ston94] Stone, M.C., Fishkin, K., Bier, E., “The Movable Filter as a User Interface Tool”, *Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems*, 1994, pp. 306-312.