Toolglass and Magic Lenses: The See-Through Interface

Eric A. Bier, Maureen C. Stone, Ken Pier, William Buxton†, Tony D. DeRose‡
Xerox PARC, 3333 Coyote Hill Road, Palo Alto, CA 94304
†University of Toronto, ‡University of Washington

Abstract

Toolglass™ widgets are new user interface tools that can appear, as though on a transparent sheet of glass, between an application and a traditional cursor. They can be positioned with one hand while the other positions the cursor. The widgets provide a rich and concise vocabulary for operating on application objects. These widgets may incorporate visual filters, called *Magic Lens™ filters*, that modify the presentation of application objects to reveal hidden information, to enhance data of interest, or to suppress distracting information. Together, these tools form a *see-through interface* that offers many advantages over traditional controls. They provide a new style of interaction that better exploits the user's everyday skills. They can reduce steps, cursor motion, and errors. Many widgets can be provided in a user interface, by designers and by users, without requiring dedicated screen space. In addition, lenses provide rich context-dependent feedback and the ability to view details and context simultaneously. Our widgets and lenses can be combined to form operation and viewing macros, and can be used over multiple applications.

CR Categories and Subject Descriptors: I.3.6 [**Computer Graphics**]: Methodology and Techniques–interaction techniques; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces–interaction styles; I.3.3 [**Computer Graphics**]: Picture/Image Generation–viewing algorithms; I.3.4 [**Computer Graphics**]: Graphics Utilities–graphics editors

Key Words: multi-hand, button, lens, viewing filter, control panel, menu, transparent, macro

1. Introduction

We introduce a new style of graphical user interface, called the *see-through interface*. The see-through interface includes semi-transparent interactive tools, called *Toolglass™ widgets*, that are used in an application work area. They appear on a virtual sheet of transparent glass, called a *Toolglass sheet*, between the application and a traditional cursor. These widgets may provide a customized view of the application underneath them, using viewing filters called *Magic Lens™* filters. Each lens is a screen region together with an operator, such as ''magnification'' or ''render in wireframe,'' performed on objects viewed in the region. The user positions a Toolglass sheet over desired objects and then points through the widgets and lenses. These tools create *spatial modes* that can replace temporal modes in user interface systems.

Two hands can be used to operate the see-through interface. The user can position the sheet with the non-dominant hand, using a device such as a trackball or touchpad, at the same time as the dominant hand positions a cursor (e.g., with a mouse or stylus). Thus, the user can line up a widget, a cursor, and an application object in a single two-handed gesture.

A set of simple widgets called *click-through buttons* is shown in figure 1. These buttons can be used to change the color of objects below them. The user positions the widget in the vicinity and indicates precisely which object to color by clicking through the button with the cursor over that object, as shown in figure 1(b). The buttons in figure 1(c) change the outline colors of objects. In addition, these buttons include a filter that shows only outlines, suppressing filled areas. This filter both reminds the user that these buttons do not affect filled areas and allows the user to change the color of outlines that were obscured.
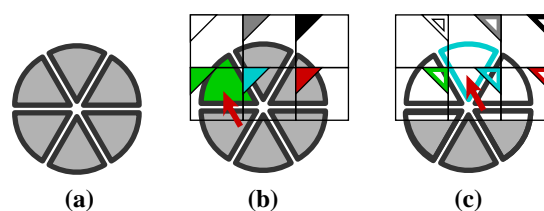

Figure 1. Click-through buttons. (a) Six wedge objects. (b) Clicking through a green fill-color button. (c) Clicking through a cyan outline-color button.

Many widgets can be placed on a single sheet, as shown in figure 2. The user can switch from one command or viewing mode to another simply by repositioning the sheet.
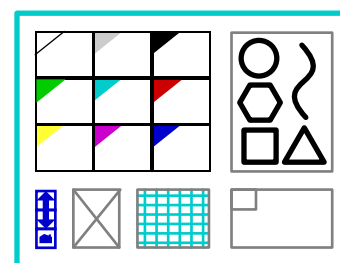
Figure 2. A sheet of widgets. Clockwise from upper left: color palette, shape palette, clipboard, grid, delete button, and buttons that navigate to additional widgets.

Widgets and lenses can be composed by overlapping them, allowing a large number of specialized tools to be created from a small basic set. Figure 3 shows an outline color palette over a magnifying lens, which makes it easy to point to individual edges.
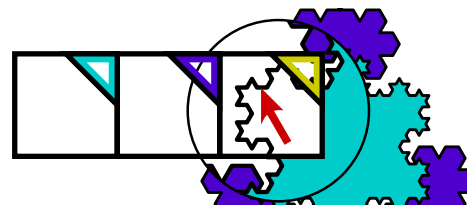
Figure 3. An outline color palette over a magnifying lens.

The see-through interface has been implemented in the Multi-Device Multi-User Multi-Editor (MMM) framework[5] in the Cedar

programming language and environment,[24] running on the SunOS UNIX™↑compatible operating system on Sun Microsystems SPARCstations and other computers. The Gargoyle graphics editor,[20] as integrated into MMM, serves as a complex application on which to test our interface. We use a standard mouse for the dominant hand and a MicroSpeed FastTRAP™ trackball for the non-dominant hand. The trackball includes three buttons and a thumbwheel, which can be used to supply additional parameters to the interface.

The remainder of this paper is organized as follows. The next section describes related work. Section 3 describes some examples of the tools we have developed. Section 4 discusses general techniques for using the see-through interface. Section 5 discusses some advantages of this approach. Section 6 describes our implementation. Sections 7 and 8 present our conclusions and plans for future work.

Except for figures 12 and 16, all of the figures in this paper reflect current capabilities of our software.

2. Related Work

The components of the see-through interface combine work in four areas: simultaneous use of two hands, movable tools, transparent tools, and viewing filters. In this section, we describe related work in these four areas.

Multi-Handed Interfaces

Several authors have studied interfaces that interpret continuous gestures of both hands. In Krueger's VIDEOPLACEs system,[15] the position and motion of both of a participant's hands, as seen by a video camera, determine the behavior of a variety of on-screen objects, including animated creatures and B-spline curves. Buxton and Myers discovered that users naturally overlap the use of both hands, when this is possible, and that, even when the two hands are used sequentially, there is still a performance advantage over single-hand use.[7,8]

Other work characterizes the situations under which people successfully perform two-handed tasks. Guiard presents evidence that people are well-adapted to tasks where the non-dominant hand coarsely positions a context and the dominant hand performs detailed work in that context.[4] Similarly, Kabbash presents evidence that a user's non-dominant hand performs as well or better than the dominant hand on coarse positioning tasks.[13]

Our system takes full advantage of a user's two-handed skills; the non-dominant hand sets up a context by coarsely positioning the sheet, and the dominant hand acts in that context, pointing precisely at objects through the sheet.

Movable Tools

Menus that pop up at the cursor position are movable tools in the work area. However, such a menu's position is determined by the cursor position before it appears, making it difficult to position it relative to application objects.

Several existing systems provide menus that can be positioned in the same work area as application objects. For example, MacDraw ''tear-off menus'' allow a pull-down menu to be positioned in the work area and repositioned by clicking and dragging its header.[17] Unfortunately, moving these menus takes the cursor hand away from its task, and they must be moved whenever the user needs to see or manipulate objects under them.

Toolglass sheets can be positioned relative to application objects and moved without tying up the cursor.

Transparent Tools

Some existing systems that allow menus to be positioned over the work area make these menus transparent. For example, the Alto Markup system[18] displays a menu of modes when a mouse button goes down. Each menu item is drawn as an icon, with the space between icons transparent. Bartlett's transparent controls for interactive graphics use stipple patterns to get the effect of transparency in X Windows.[2]

While these systems allow the user to continue to see the underlying application while a menu is in place, they don't allow the user to interact with the application through the menu and they don't use filters to modify the view of the application, as does our interface.

Viewing Filters

Many existing window systems provide a pixel magnifier. Our Magic Lens filters generalize the lens metaphor to many representations other than pixels and to many operations other than magnification. Because they can access application-specific data structures, our lenses are able to perform qualitatively different viewing operations, including showing hidden information and showing information in a completely different format. Even when the operation is magnification, our lenses can produce results of superior quality, since they are not limited to processing data at screen resolution.

The concept of using a filter to change the way information is visualized in a complex system has been introduced before.[25,10,14] Recent image processing systems support compostition of overlapping filters.[23] However, none of these systems combine the filtered views with the metaphor of a movable viewing lens.

Other systems provide special-purpose lenses that provide more detailed views of state in complex diagrams. For example, a fisheye lens can enhance the presentation of complicated graphs.[21] The bifocal display[22] provides similar functionallity for viewing a large space of documents. The MasPar Profiler[3] uses a tool based on the magnifying lens metaphor to generate more detail (including numerical data) from a graphical display of a program.

Magic Lens filters combine viewing filters with interaction and composition in a much broader way than do previous systems. They are useful both as a component of the see-through interface and as a general-purpose visualization paradigm, in which the lenses become an integral part of the model being viewed.

3. Examples

This section shows several tools that demonstrate features of the see-through interface. Because we have implemented primarily in the graphical editing domain, most of these tools are tailored to that application. However, the see-through interface can be used in a wide variety of other application domains.

Shape and Property Palettes

Palettes are collections of objects or properties that can be added to a scene. Figure 1 showed two widgets that apply color to shapes. Similar tools can be designed to apply other graphical properties, such as type and line styles to an illustration, shading parameters to a 3D model, or initial values to a simulation. Figure 4 illustrates a widget containing graphical shapes that can be ''pushed through'' from the tool into the illustration below. In figure 4(a), the user has positioned a shape palette widget (shown in cyan) over an illustration (shown in magenta). When the user clicks on a shape on the tool, a copy of that shape is added to the illustration. The widget attaches the copied shape to the cursor for interactive dragging until the final shape position is achieved (figure 4(b)).

Figure 4. Shape palette. (a) Choosing a shape. (b) Placing the shape.

Figure 5 shows a design for a property palette for setting the face of text in a document. Each face (regular, bold, etc.) has an active region on the right side of the tool. Selecting the text displayed in this region changes its face.

regular	temporal modes and modes created by holding down a keyboard key with
italic	*spatial modes.* Because these spatial
bold	modes can be changed **directly** in the
bold italic	application work area, the cursor and
	the user's attention can remain on the

Figure 5. Font face palette. The word "directly" is being selected and changed to bold face.
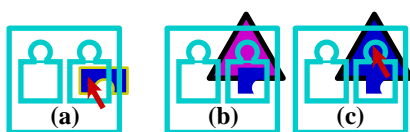
Clipboards

Clipboard widgets pick up shapes and properties from underlying objects, acting as visible instantiations of the copy and paste keys common in many applications. Clipboards can pick up entire objects or specific properties such as color, dash pattern or font. They can hold single or multiple copies of an object. The objects or properties captured on the clipboard can be copied from the clipboard by clicking on them, as in the palette tools.

Figure 6 shows a symmetry clipboard that picks up the shape that the user clicks on (figure 6(a)) and produces all of the rotations of that shape by multiples of 90 degrees (figure 6(b)). Moving the clipboard and clicking on it again, the user drops a translated copy of the resulting symmetrical shape (figure 6(c)). Clicking the small square in the upper left corner of the widget clears the widget so that new shapes can be clipped.
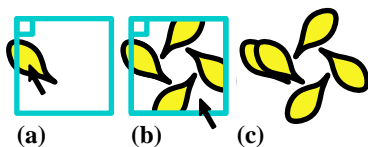
Figure 6. Symmetry clipboard. (a) Picking up an object. (b) Rotated copies appear. (c) The copies are moved and pasted.

Figure 7 shows an example of a type of clipboard that we call a *rubbing*. It picks up the fill color of an object when the user clicks on that object through the widget (figure 7(a)). The widget also picks up the shape of the object as a reminder of where the color came from (figure 7(b)). Many fill-color rubbings can be placed on a single sheet, allowing the user to store several colors and remember where they came from. The stored color is applied to new shapes when the user clicks on the applicator nib of the rubbing (figure 7(c)).


Figure 7. Fill-color rubbings. (a) Lifting a color. (b) Moving the clipboard. (c) Applying the color.

Besides implementing graphical cut and paste, clipboards provide a general mechanism for building customized libraries of shapes and properties.

Previewing Lenses

In graphical editing, a lens can be used to modify the visual properties of any graphical object, to provide a preview of what changing the property would look like. Properties include color, line thickness, dash patterns, typeface, arrowheads and drop shadows. A previewing lens can also be used to see what an illustration would look like under different circumstances; for example, showing a color illustration as it would be rendered on a black/white display or on a particular printer. Figure 8 shows a Celtic knotwork viewed through two lenses, one that adds drop shadows and one that shows the picture in black and white. The achromatic lens reveals that the drop shadows may be difficult to distinguish from the figure on a black/white display.
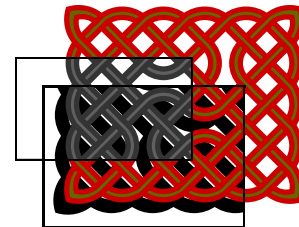

Figure 8. An achromatic lens over a drop shadow lens over a knotwork. (Knotwork by Andrew Glassner)

Previewing lenses can be parameterized. For example, the drop shadow lens has parameters to control the color and displacement of the shadow. These parameters can be included as graphical controls on the sheet near the lens, attached to input devices such as the thumbwheel, or set using other widgets.

Selection Tools

Selection is difficult in graphical editing when objects overlap or share a common edge. Our selection widgets address this problem by modifying the view and the interpretation of input actions. For example, figure 9 shows a widget that makes it easy to select a shape vertex even when it is obscured by other shapes. This tool contains a wire-frame lens that reveals all vertices by making shape interiors transparent. Mouse events are modified to snap to the nearest vertex.
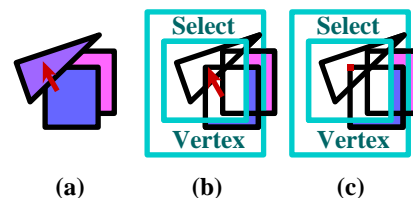
Figure 9. Vertex selection widget. (a) Shapes. (b) The widget is placed. (c) A selected vertex.
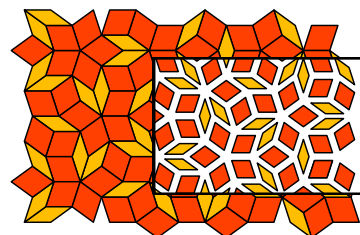

Figure 10. The local scaling lens. (Tiling by Doug Wyatt)

Figure 10 shows a lens that shrinks each object around its own centroid. This lens makes it easy to select an edge that is coincident with one or more other edges.

Grids

Figure 11 shows three widgets, each of which displays a different kind of grid. The leftmost two grids are rectangular with different spacings. The rightmost grid is hexagonal. Although each grid only appears when the lens is in place, the coordinates of the grid are bound to the scene, so that grid points do not move when the sheet moves. By clicking on the grid points and moving the widget, the user can draw precise shapes larger than the widget. If the sheet is moved by the non-dominant hand, the user can quickly switch between the grids during an editing motion.
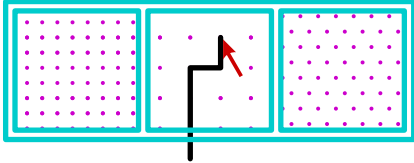
Figure 11. Three grid tools.

Visualization

Figure 12 illustrates the use of tools and lenses to measure Gaussian curvature in the context of a shaded rendering of a 3D model. The pseudo-color view indicates the sign and relative magnitude of the curvature,[9] and the evaluation tool displays the value at the point indicated.
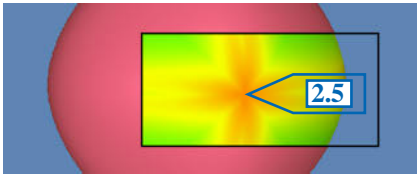
Figure 12. Gaussian curvature pseudo-color lens with overlaid tool to read the numeric value of the curvature. (Original images courtesy of Steve Mann)

4. Using the See-Through Interface

Widgets and lenses are most effective when supported by appropriate conventions specifying how to position, size, organize, and customize them. This section discusses a few of these issues.

Moving and Sizing the Sheet or the Application

A Toolglass sheet can be moved by clicking and dragging on its border with a mouse or by rolling the trackball. The sheet and all its widgets can stretch and shrink as a unit when the user works a a second controller such as a thumbwheel. With these moving and sizing controls, the user can center a widget on any application object and size the widget to cover any screen region. Large widgets can be used to minimize sheet motion when applying a widget to several objects. A widget that has been stretched to cover the entire work area effectively creates a command mode over the entire application.

By clicking a button on the trackball, the user can disconnect the trackball from the sheet and enable its use for scrolling and zooming a selected application area. If a sheet is over this application, the user can now move an application object to a widget instead of moving a widget to an object. This is a convenient way to use the see-through interface on illustrations that are too large to fit on the screen.

Managing Sheets

A typical application will have a large number of widgets in its interface. To avoid clutter, we need a way to organize these widgets and sheets. One approach is to put all of the widgets on a single sheet that can be navigated by scrolling and zooming. Perlin and Fox's paper in these proceedings[19] describes techniques for creating and navigating unlimited structures on a single sheet. A second approach is to have a master sheet that generates other sheets. Each of these sheets could generate more sheets, like hierarchical menus. A third technique, used in our prototype, is to allow a single sheet to show different sets of widgets at different times. The set to display can be selected in several ways: the user can click a special widget in the set, like the arrows in HyperCard,™[11] that jumps to another set. In addition, a master view provides a table of contents of the available sets allowing the user to jump to any one. To use different sets simultaneously, the user creates additional sheets.

Customizing Sheets

Because sheets can contain an unlimited number of widgets, they provide a valuable new substrate on which users can create their own customized widgets and widget sets. In effect, the sheets can provide a user interface *editor*, allowing users to move and copy existing widgets, compose macros by overlapping widgets, and snap widgets together in new configurations. Indeed, with the techniques described in this paper, one Toolglass sheet could even be used to edit another.

5. Advantages of See-Through Tools

In this section, we describe some advantages we see for using the see-through interface. Most of these advantages result from placing tools on overlapping layers and from the graphical nature of the interface.

In most applications, a control panel competes for screen space with the work area of the application. Toolglass sheets exist on a layer above the work area. With proper management of the sheets, they can provide an unlimited space for tools. The widgets in use can take up the entire work area. Then, they can be scrolled entirely off the screen to provide an unobstructed view of the application or space for a different set of widgets.

The see-through user interface can be used on tiny displays, such as notebook computers or personal digital assistants, that have little screen real estate for fixed-position control panels. It can also be used on wall-sized displays, where a fixed control panel might be physically out of reach from some screen positions. These tools can move with the user to stay close at hand.

A user interface layer over the desktop provides a natural place to locate application-independent tools, such as a clipboard that can copy material from one window to another.

These widgets can combine multiple task steps into a single step. For example, the vertex selection widget of figure 9 allows the user to turn on a viewing mode (wire-frame), turn on a command mode (selection), and point to an object in a single two-handed gesture.

Most user interfaces have temporal modes that can cause the same action to have different effects at different times. With our interface, modes are defined spatially by placing a widget and the cursor over the object to be operated on. Thus, the user can easily see what the current mode is (e.g., by the label on the widget) and how to get out of it (e.g., move the cursor out of the widget). In addition, each widget can provide customized feedback for its operation. For example, a widget that edits text in an illustration can include a lens that filters out all the objects except text. When several widgets are visible at once, the feedback in each one

76

serves a dual role. It helps the user make proper use of the widget and it helps the user choose the correct widget.

The visual nature of the see-through interface also allows users to construct personalized collections of widgets as described above.

6. Implementation

This section provides an overview of our implementation of the see-through interface.

Toolglass Sheets

We describe three Toolglass subsystems: one that handles simultaneous input from two pointing devices and updates the screen after multiple simultaneous changes, one that modifies pointing events as they pass through widgets, and one that modifies graphical output as it passes up through each widget.

Multi-Device Input and Screen Refresh

Our Toolglass software uses the MMM framework.[5] The see-through interface relies on the following features of MMM.

MMM takes events from multiple input devices, such as the mouse and trackball, keeps track of which device produced which event, and places all events on a single queue. It dequeues each event in order and determines to which application that event should be delivered. MMM applications are arranged in a hierarchy that indicates how they are nested on the screen. Each event is passed to the root application, which may pass the event on to one of its child applications, which may in turn pass the event on down the tree. Mouse events are generally delivered to the most deeply nested application whose screen region contains the mouse coordinates. However, when the user is dragging or rubberbanding an object in a particular application, all mouse coordinates go to that application until the dragging or rubberbanding is completed. Keyboard events go to the currently selected application.

To support Toolglass sheets, MMM's rules for handling trackball input were modified. When a sheet is movable, trackball and thumbwheel events go to the top-level application, which interprets them as commands to move or resize the sheet, respectively. When the sheet is not movable, the trackball and thumbwheel events are delivered to the selected application, which interprets them as commands to scroll or zoom that application.

Filtering Input Through Lenses and Widgets

(a) | **(b)**

Root Application

Toolglass Sheet | **Text Editor** | **Graphical Editor**
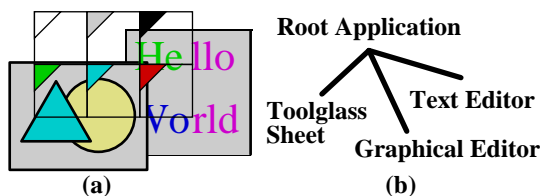
Figure 13. A simple hierarchy of applications

Ordinarily, MMM input events move strictly from the root application towards the leaf applications. However, to support the see-through interface, input events must be passed back up this tree. For example, figure 13(b) shows an application hierarchy. The left-to-right order at the lower level of this tree indicates the top-to-bottom order of applications on the screen. Input events are first delivered to the Toolglass sheet to determine if the user is interacting with a widget or lens. If so, the event is modified by the sheet. In any case, the event is returned to the root application, which either accepts the event itself or passes it on to the child applications that appear farther to the right in the tree.

The data structure that represents an MMM event is modified in three ways to support Toolglass sheets. First, an event is annotated with a representation of the parts of the application tree it has already visited. In figure 13, this prevents the root application from delivering the event to the sheet more than once. Second, an event is tagged with a command string to be interpreted when it reaches its final application. For example, a color palette click-through button annotates each mouse-click event with the command name ''FillColor'' followed by a color. Finally, if the widget contains a lens, the mouse coordinates of an event may be modified so the event will be correctly directed to the object that appears under the cursor through that lens.

(a)   **(b)**   **(c)**
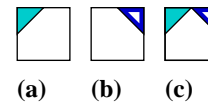
Figure 14. Composing color-changing widgets.

Widgets can be composed by overlapping them. When a stack of overlapped widgets receives input (e.g., a mouse click), the input event is passed top-to-bottom through the widgets. Each widget in turn modifies the command string that has been assembled so far. For example, a widget might concatenate an additional command onto the current command string. In figure 14, a widget that changes fill colors (figure 14(a)) is composed with a widget that changes line colors (figure 14(b)) to form a widget that changes both fill and line colors (figure 14(c)). If the line color widget is on top, then the command string would be ''LineColor blue'' after passing through this widget, and ''LineColor blue; FillColor cyan'' after both widgets.

Filtering Output Through Lenses and Widgets

Ordinarily, MMM output is composed from the leaf applications up. To support lenses, the normal screen refresh composition has been extended to allow information to flow down and across the tree as well as up. For example, if the widgets in figure 13 contain one or more lenses, and if any of those lenses is situated over the graphical editor, each lens must examine the contents of the graphical editor (which is the lens's sibling in the hierarchy) in order to draw itself.

In addition, to improve performance, MMM applications compute the rectangular bounding box of the regions that have recently changed, and propagate this box to the root application, which determines which screen pixels will need to be updated. Generally, this bounding box is passed up the tree, transformed along the way by the coordinate transformation between each application and the next one up the tree. However, lenses can modify the set of pixels that an operation affects. A magnifying lens, for example, generally increases the number of pixels affected. As a result, the bounding box must be passed to all lenses that affect it to determine the final bounding box.

Magic Lens Filters

A Magic Lens filter modifies the image displayed on a region of the screen, called the *viewing region*, by applying a *viewing filter* to objects in a model. The *input region* for the lens is defined by the viewing region and the viewing filter. It may be the same size as the viewing region, or different, as in the magnification lens. For a 3D model, the input region is a cone-shaped volume defined by the eye point and the viewing region. Input regions can be used to cull away all model objects except those needed to produce the lens image. Our current implementations do not perform this culling; as described below, there are advantages to lenses that operate on the entire model.

When several lenses are composed, the effect is as though the

model were passed sequentially through the stack of lenses from bottom to top, with each lens operating on the model in turn. In addition, when one lens has other lenses below it, it may modify how the boundaries of these other lenses are mapped onto the screen within its own boundary. The input region of a group of lenses taken as a whole can be computed by applying the inverses of the viewing filters to the lens boundaries themselves.

Our lenses depend on the implementation of Toolglass sheets to manage the size, shape and motion of their viewing regions. This section describes two strategies we have tried for implementing viewing filters: a procedural method that we call *recursive ambush*, and a declarative method that we call *model-in model-out*. We also describe a third method that promises to be convenient when applicable, called *reparameterize-and-clip*. Finally, we discuss issues that arise in the presence of multiple model types.

Recursive Ambush

In the recursive ambush method, the original model is described procedurally as a set of calls in a graphics language such as Interpress[12] or PostScript.[1] The lens is a new interpreter for the graphics language, with a different implementation for each graphics primitive. In most cases, the implementation of a given graphics primitive first performs some actions that carry out the modifying effect of the lens and then calls the previous implementation of the primitive. For example, a lens that modifies a picture such that all of its lines are drawn in red would modify the ''DrawLine'' primitive to set the color to red and then call the original ''DrawLine'' primitive.

When lenses are composed, the previous implementation may not be the original graphics language primitive, but another lens primitive that performs yet another modification, making composition recursive.

Recursive ambush lenses appear to have important advantages. Because they work at the graphics language level, they work across many applications. Because they work procedurally, they need not allocate storage. However, the other methods can also work at the graphics language level. In addition, recursive ambush lenses have three major disadvantages. First, making a new lens usually requires modifying many graphics language primitives. Second, debugging several composed lenses is difficult because the effects of several cooperating interpreters are hard to understand. Finally, performance deteriorates rapidly as lenses are composed because the result of each lens is computed many times; the number of computations doubles with the addition of each lens that overlaps all of the others.

Model-In Model-Out

In the model-in model-out (MIMO) method, we make a copy of the original model as the first step. This model might be the data structure of an editor, a representation of graphics language calls, an array of pixels or some other picture representation. The implementation walks through this data structure and modifies it in accordance with the desired behavior of the lens. When composed with other lenses, a MIMO lens takes each model that is produced by each lens under it, produces a modified version of that model, and associates it with the clipping region formed by intersecting its clipping region with that of the lens underneath. The resulting models are passed on to lenses above.

Although MIMO lenses must allocate storage, this investment pays off in several ways. First, during the rendering of a single image, each lens computes its output models only once, and then saves them for use by any lenses that are over it. In addition, if the computed model is based on the entire original model, then redrawing the picture after a lens moves is just a matter of changing clipping regions; no new model filtering is needed. In this case, each lens maintains a table of the models it has produced. The table is indexed by the models it has received as input and when they were last modified. The action of such a lens often consists of a single table lookup.

MIMO lenses have many other advantages. Given routines to copy and visit parts of the model, the incremental effort to write a MIMO lens is small. Many of our lenses for graphical editor data structures were written in under 20 minutes and consist of under 20 lines of code. Debugging composed lenses is easy because the intermediate steps can easily be viewed. Finally, MIMO lenses can perform a large class of filtering functions because they can access the input model in any order. In particular, they can compute their output using graphical search and replace,[16] as shown in figure 15 where each line segment is replaced by multiple line segments to create a ''snowflake'' pattern.
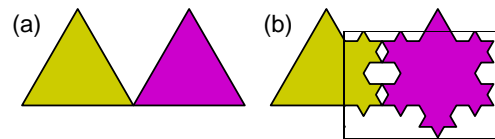
Figure 15. The snowflake lens. (a) Two triangles. (b) Snowflake lens over part of the scene.

An important variation of MIMO is to allow the output model to differ in type from the input model. For example, a lens might take a graphics language as input and produce pixels as output. In this case, the lens walks the original model, rather than copying it, and allocates data structures of the new model type.

Reparameterize and Clip

If the original image is being produced on the screen by a renderer with variable parameters, it is easy to implement lenses that show the effects of varying those parameters. To function, the lens modifies a renderer parameter and asks the renderer to redraw the model clipped to the boundary shape of the lens. For example, a lens showing the wireframe version of a 3D shaded model can be implemented this way.

Several reparameterize-and-clip lenses can be composed if the parameter changes made by these lenses are compatible. In the region of overlap, the renderer re-renders the original model after each of the overlapping lenses has made its changes to the renderer parameters. The flow of control and performance of a stack of these lenses is like that of MIMO lenses; a new output is computed for each input region received from lenses underneath. These lenses differ from MIMO in that each output is computed from the original model, and each output is always a rendering.

Multiple Model Types

In our discussion above, lenses are used to view a single type of model, such as a graphical editor data structure or a graphical language. In practice, multiple model types are often present, for two reasons. First, a lens can overlap multiple applications at the same time, where the applications have different model types, as shown above in figure 13. Second, a lens may overlap both an application and a lens, where the lens output and application model are of different types. For example, in figure 16, the wireframe lens converts from a 3D model to a 2D line drawing. The magnifier lens, which operates on 2D drawings, overlaps both the original image and the output of the wireframe lens. Rich illustrations can be produced by permitting lenses to overlap multiple model types in this way.

Supporting multiple model types requires *type conversion* and

78

type tolerance. When a lens that expects one type of model as input is moved over a model of a different type, the system may automatically convert the model to be of the type required; this is type conversion. For example, all of our applications produce Interpress graphics language calls as part of drawing themselves on the screen. When a lens that takes Interpress as input is positioned over one of these applications, that application converts its model to Interpress on demand for that lens.
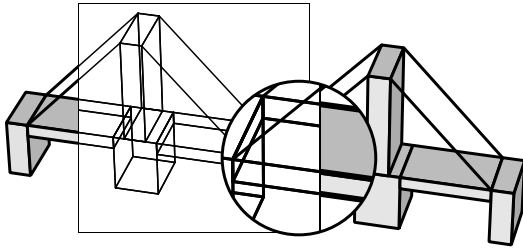
Figure 16. A bridge made of shaded, 3D blocks showing a 3D wireframe lens and a 2D magnifier.

Alternatively, when presented with a model it does not understand, a lens can simply pass that model through unchanged; this is type tolerance. For example, a lens that operates only on a graphics editor's data structures will only modify the image in the part of that lens's boundary that overlaps the graphics editor; other regions are unchanged.

Composing Widgets and Lenses

When a widget and a lens are composed, their functions combine. For example, consider a click-through button on top of a magnifying lens. Mouse events pass through the button, are annotated with a command, and then pass through the lens, which applies the inverse of its transformation to the mouse coordinates. During screen refresh, the widget adds its appearance to the output of the lens. If the lens is on top of the widget, input events are first transformed by the lens and then tested to see if they fall within the button or not; during refresh, the widget adds its appearance to the model, which is then acted on by the lens. A widget and lens can be very tightly coupled. For example, an editing tool could include a lens that displayed control points or editing handles implemented as widgets.

Performance

Our sheets and lenses are already fast enough to be useful on current hardware, but need to be faster for smooth motion. For example, using our prototype on a SPARCstation 10, we measured the time it takes to redraw the screen after moving a wireframe lens of size 70 by 70 pixels over the Penrose tiling of figure 10, containing 117 filled and outlined shapes. For the MIMO implementation of the lens, once it has cached its output scene, it takes an average of 300 milliseconds to repaint the scene, of which 120 milliseconds are spent drawing the lens interior. The same lens implemented using recursive ambush takes %15 longer to redraw the lens interior, which we attribute to the procedure call overhead of the recursive approach. Computing the filtered scene for the MIMO lens takes an average of 480 milliseconds for this example. This computation is performed whenever the illustration under the lens is changed or lens parameters are modified.

Figure 17. The Magic Lenses logo.

7. Conclusions

We have described a new style of user interface, the see-through interface, based on Toolglass widgets and Magic Lens filters. The see-through interface offers a new design space for user interfaces based on spatial rather than temporal modes and provides a natural medium for two-handed interaction. Because the interface is movable and overlays the application area, it takes no permanent screen space and can be conveniently adapted to a wide range of display sizes. Because the overlaid tools are selected and brought to the work area simply by moving the Toolglass sheet, the user's attention can remain focused on the work area. Because the operations and views are spatially defined, the user can work without changing the global context.

The see-through interface provides a new paradigm to support open software architecture. Because Toolglass sheets can be moved from one application to another, rather than being tied to a single application window, they provide an interface to the common functionality of several applications and may encourage more applications to provide common functionality. Similarly, Magic Lens filters that take standard graphics languages as input work over many applications.

In addition to their role in user interfaces, Magic Lens filters provide a new medium for computer graphics artists and a new tool for scientific visualization. When integrated into drawing tools, these filters will enable a new set of effects and will speed the production of traditional effects. Figure 17 shows a magnifying lens and a wireframe lens used to produce our Magic Lenses logo.

Integrated into scientific visualization tools, these filters can enhance understanding by providing filtered views of local regions of the data while leaving the rest of the view unchanged to provide context, as was shown in the visualization example in figure 12.

We hope the see-through interface will prove to be valuable in a wide variety of applications. While the examples in this paper stress applications in graphical editing, these tools can potentially be used in any screen-based application, including spreadsheets, text editors, multi-media editors, paint programs, solid modelers, circuit editors, scientific visualizers, or meeting support tools. Consider that most applications have some hidden state, such as the equations in a spreadsheet, the grouping of objects in a graphical editor, or the position of water pipes in an architectural model. A collection of widgets and lenses can be provided to view and edit this hidden state in a way that takes up no permanent screen space and requires no memorization of commands.

We believe that the see-through interface will increase productivity by reducing task steps and learning time, providing good graphical feedback, and allowing users to construct their own control panels and spatial modes.

8. Plans for Future Work

The see-through interface is a framework that can be used to create many new tools in many application domains. Exploring the current space of possibilities will take many people many years. Furthermore, this design space will be enlarged by future software and hardware. We will carry out some of this exploration ourselves, creating new widgets in different application domains, working out taxonomies for the tools we discover, designing new conventions for composing, editing, navigating, organizing and triggering these tools, combining them with existing user interface techniques, and testing them on users performing real work.

We are building two Toolglass widget toolkits. The first is a

traditional toolkit in which widgets are created through object-oriented programming. The second toolkit is based on our EmbeddedButtons project;[6] here, users draw new widgets and collections of widgets using a graphical editor and then apply behavior to these graphical forms, where the behavior is expressed in a user customization language.

We are designing new algorithms to increase the speed of these tools. It is clear that Magic Lens filters and, to a lesser extent, Toolglass widgets provide a new way to consume the graphics power of modern computers.

Finally, we are working to better understand how to model and implement general composition of widgets and lenses, especially those that work with multiple model and applications types.

References

1. Adobe Systems Incorporated. *PostScript® Language Reference Manual, second edition.* Addison-Wesley, 1990.

2. Bartlett, Joel F. Transparent Controls for Interactive Graphics. WRL Technical Note TN-30, Digital Equipment Corp., Palo Alto, CA. July 1992.

3. Beck, Kent, Becher, Jon, and Zaide, Liu. Integrating Profiling into Debugging. *Proceedings of the 1991 International Conference on Parallel Processing, Vol. II, Software,* August 1991, pp. II-284-II-285.

4. Guiard, Yves. Asymmetric Division of Labor in Human Skilled Bimanual Action: The Kinematic Chain as a Model. *The Journal of Motor Behavior, 19,* 4, (1987), pp. 486-517.

5. Bier, Eric A. and Freeman, Steve. MMM: A User Interface Architecture for Shared Editors on a Single Screen. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology* (Hilton Head, SC, November 11-13), ACM, New York, (1991), pp. 79-86.

6. Bier, Eric A., EmbeddedButtons: Supporting Buttons in Documents. *ACM Transactions on Information Systems*, *10,* 4, (1992), pp. 381-407.

7. Buxton, William and Myers, Brad A.. A Study in Two-Handed Input. *Proceedings of CHI '86* (Boston, MA, April 13-17), ACM, New York, (1986), pp. 321-326.

8. Buxton, William. There's More to Interaction Than Meets the Eye: Some Issues in Manual Input. *Readings in Human-Computer Interaction: A Multidisciplinary Approach.* (Ronald M. Baecker, William A.S. Buxton, editors). Morgan Kaufmann Publishers, Inc., San Mateo, CA. 1987.

9. Dill, John. An Application of Color Graphics to the Display of Surface Curvature. Proceedings of SIGGRAPH '81 (Dallas, Texas, August 3-7). *Computer Graphics, 15,* 3, (1981), pp. 153-161.

10. Goldberg, Adele and Robson, Dave, A Metaphor for User Interface Design, *Proceedings of the University of Hawaii Twelfth Annual Symposium on System Sciences,* Honolulu, January 4-6, (1979), pp.148-157.

11. Goodman, Danny. *The Complete HyperCard Handbook.* Bantam Books, 1987.

12. Harrington, Steven J. and Buckley, Robert R.. *Interpress, The Source Book.* Simon & Schuster, Inc. New York, NY. 1988.

13. Kabbash, Paul, MacKenzie, I. Scott, and Buxton, William. Human Performance Using Computer Input Devices in the Preferred and Non-preferred Hands. *Proceedings of InterCHI '93,* (Amsterdam, April 24-29), pp. 474-481.

14. Krasner, Glenn and Hope, Stephen, A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, *Journal of Object-Oriented Programming, 1,* 3, (1988), pp. 26-49.

15. Krueger, Myron W., Gionfriddo, Thomas, and Hinrichsen, Katrin. VIDEOPLACE – An Artificial Reality. *Proceedings of CHI '85* (San Francisco, April 14-18). ACM, New York, (1985), pp. 35-40.

16. Kurlander, David and Bier, Eric A.. Graphical Search and Replace. Proceedings of SIGGRAPH '88 (Atlanta, Georgia, August 1-5) *Computer Graphics*, *22,* 4, (1988), pp. 113-120.

17. *MacDraw Manual.* Apple Computer Inc. Cupertino, CA 95014, 1984.

18. Newman, William. *Markup User's Manual.* Alto User's Handbook, Xerox PARC technical report, (1979), pp. 85-96.

19. Perlin, Ken and Fox, David. Pad: An Alternative Approach to the Computer Interface. this proceedings.

20. Pier, Ken, Bier, Eric, and Stone, Maureen. An Introduction to Gargoyle: An Interactive Illustration Tool. *Proceedings of the Intl. Conf. on Electronic Publishing, Document Manipulation and Typography* (Nice, France, April). Cambridge Univ. Press, (1988), pp. 223-238.

21. Sarkar, Manojit and Brown, Marc H.. Graphical Fisheye Views of Graphs. *Proceedings of CHI '92,* (Monterey, CA, May 3-5, 1992) ACM, New York, (1992), pp. 83-91.

22. Spence, Robert and Apperley, Mark. Data Base Navigation: An Office Environment of the Professional. *Behaviour and Invormation Technology, 1,* 1, (1982), 43-54.

23. *ImageVision*, Silicon Graphics Inc., Mountain View, CA.

24. Swinehart, Daniel C., Zellweger, Polle T., Beach, Richard J., Hagmann, Robert B.. A Structural View of the Cedar Programming Environment. *ACM Transactions on Programming Languages and Systems*, *8,* 4, (1986), pp. 419-490.

25. Weyer, Stephen A. and Borning, Alan H., A Prototype Electronic Encyclopedia, *ACM Transactions on Office Systems, 3,* 1, (1985), pp. 63-88.