Systematic Output Modification in a 2D User Interface Toolkit

W. Keith Edwards‡, Scott E. Hudson†, Joshua Marinacci, Roy Rodenstein*, Thomas Rodriguez††, Ian Smith**

‡ Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304
kedwards@parc.xerox.com

††JavaSoft
Sun Microsystems
10201 N. DeAnza Blvd.
Cupertino, CA 95014
never@eng.sun.com

†Human Computer Interaction Institute
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 1513-3891
hudson@cs.cmu.edu

*Graphics Visualization and Usability Center
College Of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
{iansmith, joshuam, royrod} @cc.gatech.edu

ABSTRACT

In this paper we present a simple but general set of techniques for modifying output in a 2D user interface toolkit. We use a combination of simple subclassing, wrapping, and collusion between parent and output objects to produce arbitrary sets of composable output transformations. The techniques described here allow rich output effects to be added to most, if not all, existing interactors in an application, without the knowledge of the interactors themselves. This paper explains how the approach works, discusses a number of example effects that have been built, and describes how the techniques presented here could be extended to work with other toolkits. We address issues of input by examining a number of extensions to the toolkit input subsystem to accommodate transformed graphical output. Our approach uses a set of "hooks" to undo output transformations when input is to be dispatched.

KEYWORDS: User Interface Toolkits, Output, Rendering, Interactors, Drawing Effects.

INTRODUCTION

Graphical user interfaces have, since their inception, relied on a fairly basic set of *interactors* (sometimes called *controls* or *widgets*) to effect change in applications. These interactors have typically been rendered on the screen using fairly simple output techniques—nearly all interactors visually consist of simple lines and boxes, often drawn using only a small number of colors. Most graphical interfaces, at least for "traditional" applications, have been wanting in graphical richness.

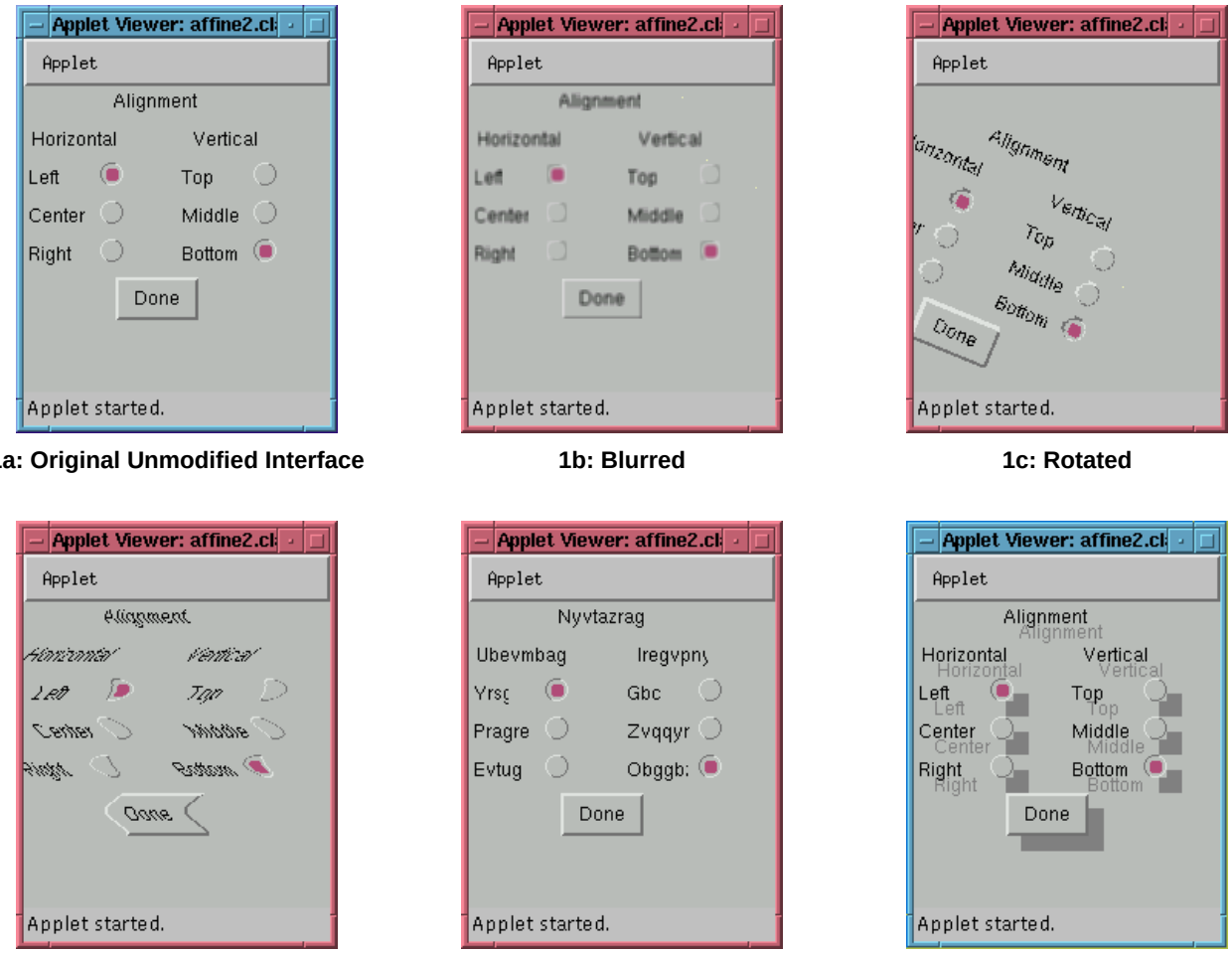
The reasons for the limited "vocabulary" of output in these interfaces is obvious—most early systems were constrained by computational power to produce only the most basic flat-looking 2D interfaces. More recently, most UI toolkit and application designers have decided that they are willing to trade computational cycles for a slightly richer look on the desktop. Hence we see 3D *looking* interactors, such as those found in Motif, Windows 95, and NextStep. The 3D look of these toolkits can, and has, been leveraged by designers to convey new visual cues in their interfaces. The dialog boxes in the OPEN LOOK system which appear to "fly off" of the desktop are an example of such a cue [13].

Our work is concerned with augmenting and extending the visual richness in user interface toolkits. In particular, we are interested in ways to create new *effects* that can be applied to objects in a graphical user interface. Unlike the move to 3D looking toolkits, where each interface interactor had to be re-coded to provide the new look, our focus is on effects that can be applied *transparently* to any existing object in an application's interface.

These effects form a new visual vocabulary that can be used by application writers to convey new cues and new information in their interfaces. These effects are visually rich, easy to create and use, and can be applied throughout an application.

Figure 1 shows an example of several transformations being applied to a simple graphical interface. All of these transformations have been implemented using our infrastructure for creating output effects; application writers who desire new effects can easily create them using this infrastructure. An application writer may choose to use these effects, and others, in any number of ways. For example, the

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.



1a: Original Unmodified Interface **1b: Blurred** **1c: Rotated**

1d: Sine Wave Mapped (Shimmer) **1e: Rot-13** **1f: Shadowed**

FIGURE 1: Examples of Output Modifications

presence of a modal dialog box may be indicated by blurring the inactive portion of an application. Scaling can be used throughout many applications, including drawing tools, interface builders, and document layout systems, to provide zooming. Shear and shimmer can be used to add novel animation effects to application interfaces and cast shadows can be used to enhance dragging interactions. This paper focuses on the architectural requirements, design, and implementation of a user interface toolkit that supports arbitrary output transformations throughout the interface. These transformations can be applied to user interface interactors (such as buttons, scrollbars, and the like), as well as other on-screen objects (such as drawn figures in a graphical editor, or text regions in a word processor).

Our work has several goals. First, it is essential that all transformations can be applied transparently. By transparency we mean that the output of on-screen objects can be modified without the objects themselves "knowing" about the modification. Interactors in the UI toolkit as well as applications can use new effects, even if the effect was written after the interactor or application was created.

Second, transformations can be composed, dynamically, at run-time. This requirement is essential since we do not wish to require that all possible transformations—and combinations of transformations—be explicitly known and implemented by the application writer or toolkit builder.

Third, we must be able to accommodate positional input in the face of output transformations. So, for example, if we scale an on-screen interactor to twice its size, we require that the interactor be able to receive and process input in its new dimensions. Such accommodation must be able to work for **arbitrary** output modifications, even ones that change the position and bounding box shape of the interactors.

The work presented here is discussed in the context of the subArctic user interface toolkit [8]. SubArctic is based on the Java [6] language, and is implemented entirely in Java—no native code outside the standard Java runtime environment is required. The techniques presented here are applicable to other interface toolkits, however, since subArctic assumes only a fairly simple model of the fundamental constructs in the user interface.

This paper is organized as follows. First, we present a brief overview of the output subsystems typically found in most object-oriented user interface toolkits. Next, we discuss our particular architectural approach to supporting arbitrary output transformations, given our goals of transparency, composition, and input accommodation. This discussion provides details of our particular implementation atop the Java Abstract Windowing Toolkit (AWT). Next, we examine several effects that we have developed as a part of the subArctic user interface toolkit. Some of these effects provide fairly simple output transformations; others implement a set of complex modifications that can significantly reduce the burden of application writers for certain tasks. These effects are presented both as an example of the kinds of modifications that are possible, and as examples of how particular transformations are implemented using our infrastructure. Finally, we discuss issues related to input that arise with our approach, and detail several approaches to solving various input problems. We close with a discussion of implementation details, conclusions, and future work.

OUTPUT OVERVIEW

Essentially all of the object oriented user interface toolkits in use today—both in the research community, such as Garnet [10], Amulet [11], Artkit [7], and Fresco [9], and in production systems, such as AWT [1] and the Be toolkit [2]—share two important properties.

- The runtime composition of the user interface is represented as a tree of objects.[1] We call the nodes of this tree *interactors*.
- The graphical operations on the display are encapsulated in an object that we call a *drawable*. The drawable object provides an interface which allows each interactor to display its output on the screen. The drawable provides access to the drawing surface, and also keeps track of the state of drawing such as the current foreground color, the current font, and so on.

In this work we will describe our output modifications in the context of the subArctic Java-based toolkit. SubArctic presents the same architectural model of interactor trees and drawables as the toolkits mentioned above, and thus our discussion of output modifications is easily applicable to them as well.

Typically in these systems, a graphical user interface is rendered to the screen through a traversal of the interactor tree. The traversal begins at the root of the tree, and a drawable object is recursively passed down the tree. At each node of the tree, interactors use the drawable's API (see Figure 2) to render their desired image onto the display. Each interior (parent) node in the tree is responsible for passing the drawable on to its children so that they will have the opportunity to draw themselves using it. The parent may modify the drawable before it is passed to each child, typically by translating the coordinate system used in the drawable to one "rooted" at the child and clipped to the child's bounding box.

1. The Fresco toolkit allows this structure to be a directed acyclic graph rather than a proper tree but this modification does not change our discussion so we will assume that each toolkit uses a tree.

```
draw_line(int x1, int y1, int x2,
    int y2);
draw_rect(int x1, int y1, int x2,
    int y2);
draw_circle(int center_x, int
    center_y, int radius);
set_foregound(Color fg);
set_font(Font f);
```

FIGURE 2: A Portion of the Drawable API from subArctic

For this discussion, we have assumed that the same drawable is passed to each interactor in the tree. We shall see shortly, however, that this is not strictly required.

Note that in a system such as this, interior nodes of the tree can produce various stacking orders of their children and themselves by imposing an ordering for the traversal and drawing of themselves and their children. For example, an interior node that wanted its output to be "above" (drawn on top of) the output of its children could simply traverse its children, causing their drawing to be performed before doing any drawing of its own.

Readers familiar with object oriented programming techniques will quickly see that this abstraction of the drawing API into an object allows for effective subclassing of drawable objects. For example, subclasses of drawable objects could be (and are) used to allow the same programmatic interface to drawing on the screen and drawing on a printer. We will exploit this property quite heavily in our techniques.

The AWT Graphics Object

SubArctic is built on top of the AWT toolkit [1] which comes standard with any Java implementation. AWT uses its *Graphics* object—the underlying object on top of which subArctic's drawable object is built—as a means to facilitate platform-independence, a major goal of Java. AWT applications use the Graphics object's API to provide drawing operations, but never use the Graphics object directly—only subclasses of it. A particular AWT implementation on a given platform will provide a platform-specific implementation of the Graphics object that is supplied to the application at runtime for drawing. This technique allows the same application code to work on widely different platforms such as UNIX systems running X, Windows, and the Macintosh.

Although this platform independence is clearly beneficial, there are drawbacks to the AWT Graphics object. Most notably, AWT applications normally do not create their own

new types (subclasses) of Graphics objects. In fact, it is not clear that the AWT design even considers that user level code would want to subclass the Graphics object. For this reason, the subArctic drawable is "wrapped around" an AWT Graphics object [5]. By wrapping, we mean that the subArctic drawable forwards most calls on its API to the underlying Graphics object but also the subArctic drawable allows user code to subclass and override its behavior.

OUR APPROACH TO OUTPUT MODIFICATION

The architectural approach subArctic takes to output modification is the use of subclassing of drawable objects in conjunction with wrapping. A toolkit must support both of these techniques to achieve maximum flexibility of the output system.

As an example, consider the possibility that a drawable subclass has been written for rendering images to a printer. This subclass may not be known to the application programmer; certainly the application programmer should not be *required* to know about this subclass to write his or her application. Requiring a priori knowledge such as this defeats one of the points of subclassing, namely that application code can transparently use subclasses without modification. Subclassing provides an effective means for adding new behaviors while retaining existing APIs in a way that is transparent to application code.

But now suppose that the application writer would like to provide a drawable class that performs scaling, perhaps to implement a zoom function in a drawing application. Ideally, we would like for the scaling drawable and the printing drawable to be usable in conjunction. That is, we should be able to "mix and match" drawing functionality. If the writer of the scalable drawable is also the creator of the printing drawable, then he or she could compose these statically through subclassing. This method is inflexible, however, and does not lend itself well to easy, on-the-fly composition of drawable effects. Further, it promotes a combinatorial explosion in the drawable class hierarchy (scaled drawables, printing drawables, scaled-printing drawables, and so on, each specialized at compile-time for a given set of tasks).

Our approach is to combine the use of subclassing for static extensibility of drawables with the use of wrapping for dynamic, or run-time composition of effects. In this model, behaviors that are entirely new and not expressible in terms of existing drawing behaviors are implemented through simple subclassing. An example of this type of behavior is a printing drawable. The low-level details of how to talk to a given printer are not decomposable into terms of any existing functionality in the drawable class. Thus, entirely new behavior such as this would be created through subclassing drawable to provide a new class that implements printing.

In contrast, behaviors which can be expressed in terms of existing drawable operations are implemented through wrapping. Unlike simple subclass relationships, which are fixed at the time the application is written, wrapping can be accomplished at runtime. An example of a behavior that can be decomposed into primitive drawable operations, and hence is a suitable candidate for the wrapping pattern, is rotation. A rotate drawable can implement its operations in terms of an "inner" drawable that it calls on to perform the actual output. The rotate drawable would be instantiated with a reference to the drawable it is wrapping, and would forward method calls to it, modifying them appropriate to implement its particular effect. Wrapping can obviously be chained at runtime—users can create wrappers of wrappers to compose effects—and can be created using any drawable subclass as an inner drawable—so they can wrap "primitive" drawables such as printing drawables and so forth.

In most drawing effects, these techniques are used side-by-side. That is, a new drawable subclass will be created for a particular behavior (subclassing being used to provide type safety and polymorphism), but in most cases, the subclass will wrap another drawable and forward method calls on to it. Only operations that are not decomposable (such as printing) avoid the wrapping technique.

Note that the subArctic drawable class itself is a subclass of the AWT Graphics object, so it can be used anywhere an existing Graphics object is used, even in pure AWT (non-subArctic) code.

A third technique is used in situations where a particular output effect may need to happen *after* some complete series of drawing operations has happened. An example of this type of effect is blur. A blur performs an average of the pixel values surrounding a given pixel to compute a new value for it. Effects such as this cannot easily be computed "on the fly" as rendering is done—typically having the entire, completed image available is required for blurring.

Such "deferred drawing" effects are accomplished through collusion between a drawable that implements the effects, and a parent interactor that sets up the context and extent of the effect. As an example, blurring might be accomplished by a combination of a blur drawable and a blur parent. These two classes would collude to blur any interactors contained in the blur parent. To perform this effect, the blur parent would first create an off-screen image, and cause its children to render themselves into it (off-screen rendering is accomplished simply by creating a drawable that draws into an off-screen image). Next, this image is rendered on the screen using the blur drawable. The end effect is that the interactor tree rooted at the blur parent is rendered blurred.

This collusion can be used anytime that drawing effects can be performed only after a complete set of drawing operations has been finished. The disadvantage of this technique is that it requires modification of the interactor hierarchy by the application writer to implement the effect—the colluding parent interactor must be inserted at the appropriate point in the hierarchy.

In summary, our approach can be characterized by three main features:

- The ability to subclass drawable objects to provide new primitive output behaviors.
- The ability to wrap drawable objects inside other drawables to allow functionality to be composed.

- The ability for drawables and parent interactors to collude to perform deferred drawing.

EXAMPLES OF SIMPLE OUTPUT MODIFICATIONS

In this section we will briefly explain several of the output transformations we have implemented. These transformations are implemented as new drawable classes that either subclass or wrap other drawables. The effects are composable and—more importantly—can be used with any existing interactor transparently.

Shadow

The shadow drawable was the first drawable subclass we created. This drawable has a simple effect—it forces the color of any drawing done using it to be gray. Further, it prevents user code from changing the drawing color to anything other than gray. The goal of this drawable was to leverage the pre-existing drawing code in interactors to create their own "drop shadows" by simply substituting the shadow drawable.

To support interactors that draw using images, the shadow drawable supports both a fast and slow image handling mode. In "fast mode," the drawable simply copies a gray rectangle with the same bounds as the image to the screen. In "slow mode," only the non-transparent pixels of the images are rendered in gray. Slow mode allows images that use transparency to cast shadows with "holes" in them, albeit at the price of compute-intensive drawing.

To achieve effects like the one seen in Figure 1F, we introduce a new interior node into the interactor tree, called a "shadow parent." This parent works with a shadow drawable to create the shadow effect. When the drawing traversal reaches the shadow parent, the parent does not immediately draw its children. Instead, it performs an "extra" drawing traversal over its children, using the shadow drawable to produce images of their shadows slightly offset from the normal position of the children. Finally, it allows "normal" drawing to proceed. The end result is that shadows are drawn slightly offset and underneath any output drawn by the children.

Importantly, and like all of our drawing effects, this effect can be used with any interactor, and requires no modification to any existing interactor code.

Blur

Like the pair of shadow parent and shadow drawable, we have a blur parent and blur drawable that work together to perform a simple "averaging" blur on the blur parent's children. An averaging blur is one in which a given pixel is compared to its neighbors, the values of all nearby pixels summed and the result averaged to get the new pixel value.

Our blur parent creates this effect as a "post processing" step after the all of its children's drawing is completed. This is done by allowing the normal drawing traversal of the children to proceed as usual, but with the parent substituting a drawable that renders to an off-screen image. This image is then rendered on-screen using the blur drawable, wrapped around the original drawable.

Rot-13

Although this drawable is of questionable utility as shown in Figure 1E[1], its ability to transform drawn text makes it of interest here. The rot-13 drawable is a wrapper around another drawable and whose only function is to perform the rot-13 transformation on text drawn with it. All other output drawing operations are not changed and are passed through unmodified to the wrapped drawable.

The rot-13 transformation is one in which all the letters of the alphabet are shifted by 13 positions, wrapping around the end when necessary. For example, the rot-13 of 'A' is 'N.' Numbers, punctuation marks, and other symbols are not affected.

The image shown in Figure 1E is interesting because it shows a difficulty that can arise when a drawable is performing a drawing modification without telling the interactor. In a proportional font like the one used in Figure 1E, the size of a string to be drawn can change substantially when the content is changed—even in a simple transformation like rot-13. We have imagined that it might be possible for "language drawables" to be implemented which can render their display in different languages, but this sizing problem seems to make the construction of such a drawable quite a difficult task.

The ability to capture and modify text, as explored by the rot-13 drawable, can be used in other ways. We have constructed a drawable very similar to the rot-13 drawable that can take all output drawn with it and record it to a file and/or send the text to a speech synthesizer. This ability to "trap" all textual output has been used in the Mercator system to build an offscreen model of text on the screen for blind users [12].

Shimmer

The shimmer effect that is shown in Figure 1D is actually a snapshot of an animation. A given frame of this animation is created by shifting each horizontal line of pixels to the right an amount which is controlled by a sine wave. The vertical position of the row of pixels determines at which point along the sine wave the given row is placed. One can imagine the graph of a sine wave placed vertically down the left hand side of the image and the image shifted by an amount corresponding the amplitude of the sine wave at that point.

For the animation, the input parameter of the sine wave is gradually modified and the whole interface appears to shimmer. This shimmer effect is very similar to the "flashback sequence" marker that is common in movies and television.

This effect is accomplished by a parent and drawable using the deferred drawing approach outlined above. After the parent's children have rendered their image, the entire image is post-processed to form the shimmer.

1. We have used the rot-13 drawable to implement a lens which can be moved over the interface to see any text on the interface through the rot-13 function. This is much more useful!

Affine Transformations

The screen shot shown in Figure 1C represents taking the interface in Figure 1A and applying a rotation transform. This transformation is applied by using a special subclass of drawable, the affine drawable. The affine drawable can be parameterized by any reversible transformation on X,Y coordinates. Simple uses of the affine drawable include scaling, shearing, rotation, and other "normal" two-dimensional transformations.[1]

The affine drawable can be implemented in two basic ways. The most obvious approach is to allow the user code to "draw" into an offscreen image and then perform the appropriate transformation on each pixel. While this approach is general, it can be slow if the size of the offscreen image is large. The second approach is to subclass and intercept each function in the drawable API and transform the coordinates that are supplied *before* any drawing is done. This approach offers a substantial speed improvement (since only a few points need to be transformed) but has several drawbacks. First, one of the drawable APIs allows user code to simply copy an offscreen image onto the drawing surface. This implies that affine operations on images will be required anyway. Slightly more subtly, some of the drawable APIs don't work as expected when a transformation is being applied. For example, drawing a circle is not correct behavior for the draw_circle() function if there is a scale transformation in use that is not the same scale in the X dimension as it is in Y dimension. In this example, the affine drawable would need to realize to convert API calls from draw_circle() to draw_ellipse().

For these and other similar reasons, we have chosen to use a hybrid approach. On API calls that can be transformed without image manipulations in all cases we do so, otherwise we resort to manipulating the images themselves.

Our work here has a relationship to, but differs substantially from, the concept of glyphs found in Interviews[3] and Fresco[14]. Glyphs do not store their own size or position— they derive (compute) these quantities when needed. Further, glyphs compute their X and Y position with respect to their parent, so it is implicit that a parent can affect the computations of its children's position or size. (In [14] these properties of are used to implement a technique for mixing visual transformations with conventional UI layout techniques, which is complementary to the techniques presented here.) In our case, interactors are allowed to cache their own local copies of their state variables, so the system must conspire with particular parents to implement the output transformations. In our discussion of the input side of these transformations, we will use an approach that is more similar to glyphs.

1. Although this drawable will work correctly for any *image* drawn with it under any reversible X,Y transform it will not work as expected if polygons or lines are drawn with it and the transformation is not affine. This is because we modify only the endpoints in an API call like *draw_line()* we do not attempt to modify each point along the line.

A MORE COMPLEX OUTPUT MODIFICATION

Other applications built using subArctic can support more complex output modifications. The Timewarp collaborative toolkit [4] makes use of a number of rich drawing effects. For example, onscreen objects that are involved in "conflicts"—operations where the interpretation of the result is ambiguous—are drawn blurred or transparently to indicate their special status.

One Timewarp application, an office furniture layout tool, uses the subArctic output modification system to create a "pseudo-3D" interface. This interface leverages the existing 2D infrastructure in subArctic to create an axonometric view on an office. This view is essentially 3D without perspective; all parallel lines stay parallel.

Figure 3 shows an example from this application. Here we see a room layout containing several pieces of furniture. In this application, if two pieces of furniture are in conflict—they occupy the same space at the same time—the offending pieces of furniture are displayed as partially transparent.

FIGURE 3: An Example of a Complex Output Modification

In this example, all of the objects in the office—including tables, chairs, walls, and so on—are subArctic interactors, as are the "typical" interactors found in the interface—scrollbars, buttons, and the like. The axonometric interactors change the semantics of position and size to work in their pseudo-3D world. These interactors maintain a "floor location" and "floor size," which indicate the "virtual" position of the furniture on the floor. But to work in the subArctic 2D world, they also maintain "real" location and size information, which indicates the absolute location of the interactor in the subArctic interface.

Essentially, the “real” coordinates represent the bounding box of the projection of the furniture, while the “floor” coordinates represent the bounding box of the plan of the furniture.

Floor coordinates and dimensions are universally used to position these interactors, and to detect spatial conflicts among them. subArctic coordinates and dimensions are computed “on the fly” as needed, by a special axonometric drawable. The drawing code for the furniture interactors knows how to draw a given piece of furniture. For a chair, for example, the code understands how to draw a back, seat, and four legs. It colludes with the axonometric drawable to position, scale, and project the drawing operations it performs on it.

When the drawing for a scene is started, the top-level parent of the scene creates an axonometric drawable, and sets scaling and projection angle values for it. These values will hold for all children drawn in the scene. Next, when each child interactor draws, it begins by establishing its floor bounding box with the axonometric drawable. Next, as it draws the features of its particular piece of furniture, the axonometric drawable transforms the coordinates and angle of drawing, based on floor bounding box information, to create the projected drawing at the angle and size specified by the overall scene.

This pseudo-3D effect was created easily and efficiently in a toolkit that was primarily written for 2D interfaces.

Likewise, the transparency effect used for conflicts is accomplished by using a special transparent drawable subclass. This drawable can easily be used to render any object in the interface with a given opacity value. Interactors in the interface are unaware of this transformation.

Both of these modifications are examples of the kinds of output effects that applications can easily achieve using the flexible drawing approach outlined here.

INPUT WITH OUTPUT MODIFICATION

The primary challenge with modifying the output of interactors—principally via parents modifying their childrens’ output—is that the interactor being modified is unaware of these changes and thus may make assumptions about its size or position that are not valid. (Again, this problem does not occur with glyphs, since glyphs partially delegate the computation of their area and position to their parents.) We have implemented one solution to this problem and have devised two other possible approaches that may be useful. It should be clear that the problems of interactors having invalid local notions about their position and size is only a problem for positional input (such as mouse clicks) and is not relevant for focus-based input (such as keyboard input).

This problem primarily occurs in situations where a parent interactor knows about a transformation being applied to its childrens’ output. To address this case, we have implemented a parent/child solution that is related to Linton’s glyphs. In subArctic, direct access to instance variables is generally not allowed, but rather access is always through a method (function) call. This insulation provides us with an opportunity to add a “hook” to several key methods so that these methods could be “updated” to understand that output is being transformed. When a parent object is using an output transformation, it provides each of its children with a “hook object” that is the reverse of the output transformation in use. Methods that are used in the calculation of interactor boundary or position call through the hook object before the normal computation is performed. This mechanism gives the hook object an opportunity to manipulate the coordinate systems appropriately to “reverse” the effects of the output transformation. Effectively, we have allowed the parent object to become involved in the size and position calculation of its children.

An alternative to this approach is to manipulate the coordinates of input events explicitly with a tree walk. Under this scheme, before any input is provided to an interactor the X and Y coordinates of the input are “walked” down the tree allowing each level to perform a transformation on the values. Since most interactors don’t modify the coordinate systems of their children, in the common case this transformation would simply be the identity transformation. However, a parent that was manipulating its childrens’ output could reverse the effects of its transformation by modifying the coordinates as needed.

The reason we did not choose this approach was the possibility that an interactor might cache the coordinate values for use at a later time. If some parent interactor decided to change its transformation (without telling the interactor doing the caching) after the cached values were stored, the cached coordinates would no longer correct In practice, it is not yet clear how often this case will occur.

Another alternative we considered was a layer of translation between the input dispatch system and the interactors that receive the input. Under this approach, input would no longer be dispatched directly to interactors, but would be indirected through a table of “input filters.” When an interactor was to receive a particular type of input, the table entry for that type of input would be consulted and the input would be dispatched to the filter found in the table. The handler then is given the responsibility for transmitting the input the interactor. This would permit users to introduce new filters into the table that were aware of the output transformations and to perform the correction on the input values “early” in the process—before the interactor tree even became aware of the input.

Our primary objection to this approach is that for each new input protocol [7] the user code must be concerned with all possible output modifications and adjusted accordingly. In practice, we have found that subArctic’s users do construct new input protocols for their own uses, and we were concerned about requiring additional effort for them to work with custom input protocols.

IMPLEMENATION STATUS

All of the simple techniques shown in Figure 1 are implemented and have either already been released as part of the toolkit or will be available soon in forthcoming releases of subArctic. Most new drawable effects are fairly simple to implement. As an example the shadow drawable is 500 lines of Java code and the blur interactor is 160 lines of code. One of the authors (Marinacci) estimates that the blur interactor took 5-6 hours to successfully implement and many simple drawable modifications can probably be done in substantially less time.

The more advanced technique for axonometric drawing discussed in the text is currently being integrated with the toolkit. The original implementation of this technique was not done with a drawable.

FUTURE WORK

Our approach to modifying output has at its core the difficulty that interactors "know" their position and size. While this is good for performance, since computing a position or size requires simply retrieving a stored value, it has created many challenges for our system. Glyphs are at the other end of this design spectrum of caching versus computing position—glyphs are always required to compute their size and position. While this requirement makes output modifications easier, performance can be a question. There may be a middle ground in this spectrum, an approach we are investigating with an object we are calling a *semiglyph*. A semiglyph is an object that could have some number *N* of sizes and positions. Its "current" size and position would be selected from this set, or perhaps even computed on-the-fly, by a parent. This technique incurs some storage overhead, but gives objects the capability to perform complex output manipulations transparently.

CONCLUSIONS

In this paper, we have presented our approach to adding complex output modifications to a 2D user interface toolkit. The techniques described here allow a set of complex, arbitrary output transformations to be created and composed. Further, drawing effects can be added after the fact to existing interactor libraries and applications.

We have also discussed our approach to adapting positional input to work correctly in the face of output modifications that may transform an interactor's position, size, or bounding shape.

We believe that the techniques described here provide a convenient and systematic way to provide more realistic displays and effects to any interactor.

The strongest evidence of the utility of this approach may be found in the fact that many people outside of the subArctic research group have constructed output effects using this approach. From their experiences, we believe that the using the output techniques presented here does not require an understanding of the entire toolkit—the handling of output, and the construction of new output effects, is largely separable from the rest of toolkit. As an example, the affine drawable presented here was one of the authors' (Marinacci) first project using subArctic

REFERENCES

[1] *Abstract Window Toolkit API* Available from http://www.javasoft.com/products/JDK/1.1/docs/api/packages.html

[2] *BeOS Interface Kit* Available from http://www.be.com/documentation/be_book/InterfaceKit/ikit.html

[3] Calder, Paul, and Linton, Mark. "Glyphs: Flyweight Objects for User Interfaces," in *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST'90), Snowbird, Utah: ACM, pp. 92-100.

[4] Edwards, W.K., and Mynatt, E.D., "Timewarp: Techniques for Autonomous Collaboration." *Proceedings of the ACM Conference on Human Factors in Computing Systems* (CHI'97), Atlanta, GA: ACM, pp. 218-225.

[5] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995. Reading, Mass.

[6] Gosling, James. *The Java Programming Language.* SunSoft Press, 1996.

[7] Henry, Tyson., Hudson, Scott., and Gary Newell., "Integrating Snapping And Gesture in a User Interface Toolkit," *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST'90), Snowbird, Utah: ACM, pp. 112-122.

[8] Hudson, Scott, and Smith, Ian. "Ultra-Lightweight Constraints." *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST'96), Seattle, WA, 1996.

[9] Linton, M., and Price, C., "Building Distributed User Interfaces with Fresco," *Proceedings of the Seventh X Technical Conference,* Boston, Mass., January 1993, pp. 77-87.

[10] Myers, Brad A., "A New Model for Handling Input," *ACM Transactions on Information Systems*, 8, 3 (July 1990), pp. 289-320.

[11] Myers, B.A., McDaniel, R., Miller, R., Ferrency, A., Doane, P., Faulring, A., Borison, E., Mickish, A., and Klimovitski, A., "The Amulet Environment: New Models for Effective User Interface Software Development," Carnegie Mellon University Technical Report CMU-CS-96-189, November, 1996.

[12] Mynatt, E.D., and Edwards W.K., "Mapping GUIs To Auditory Interfaces," in *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST'92), Hilton Head, South Carolina: ACM, pp. 61-70.