Supporting Dynamic Downloadable Appearances in an Extensible User Interface Toolkit

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

Ian Smith
Graphics, Visualization, and Usability Center, and
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, 30332-0280
iansmith@cc.gatech.edu

ABSTRACT

Most consumer products, from automobiles to breakfast cereals, pay significant attention to the visual appearance they present to the consumer. Designers of these products normally create custom appearances that reflect things such as the functionality or purpose of the product, the market they are trying to reach, and the image that the company creating the product is trying to create. As graphical user interfaces begin to fully penetrate the consumer market, we expect that similar customization of appearance will and should become part of every day practice in user interface design as well. This paper describes new user interface toolkit techniques designed to support dynamic, even downloadable, appearance changes for graphical user interfaces. The long term goal of this work is to create a system of *styles* which is analogous to current systems of fonts. That is, to provide a system for applying a style of visual appearance to an interface independent of the content of the interface, and for allowing such styles to be developed at least partially independent of specific user interface components, even in many cases supporting custom interactive components that did not exist when a style was created.

Keywords: User Interface Toolkits, User Interface Appearance, Look and Feel, Constraint Systems, Web-Based Interfaces, Java™.

INTRODUCTION

With the recent emergence of the Internet as a mass market, there are new demands for improved visual appearance for interactive programs. On the web, for example, we now see significant time and effort applied to visual design. Almost all commercial or organizational web sites now attempt to use their visual appearance to create an ambiance or style which reflects their organization or products. However, because of the limitations of existing tools, most of this effort is concentrated in click sensitive static images — a medium which is only barely interactive in nature. To employ more interactive components, a web page designer is currently forced to use a fixed style which may bear no relationship to their other visual design goals, and which may even detract from them.

This paper considers new user interface toolkit techniques for allowing the appearance of interactive objects to be easily modified. The long term goal of this work is to allow user interface appearance (but not necessarily "feel" or method of operation) to be changed in a manner much like changing the font for a body of text. Like the practice of using fonts to convey subtle meaning and ambiance that accentuates the specific message being conveyed by the words of the text, we hope to allow the look of an interface to be changed to convey ambiance and general impression without (necessarily) changing the functionality or operation of the interface.

Specifically, an infrastructure for dynamically applying styles to interfaces is described. This infrastructure has been implemented in the Java™-based subArctic user interface toolkit [Hud96a, Hud96b], and allows for the creation of *style* objects which can be applied to user interfaces after they have been created. Such style objects can be created independent of particular interfaces and, in a web-based environment, can even be downloaded separately from the interface itself. Further, the style objects described here are designed to be robust across custom objects. They can provide support for the appearance of both interactive objects (what we will henceforth call *interactors*) from the system supplied library, as well many custom interactors created using toolkit extension mechanisms — even providing some significant

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.

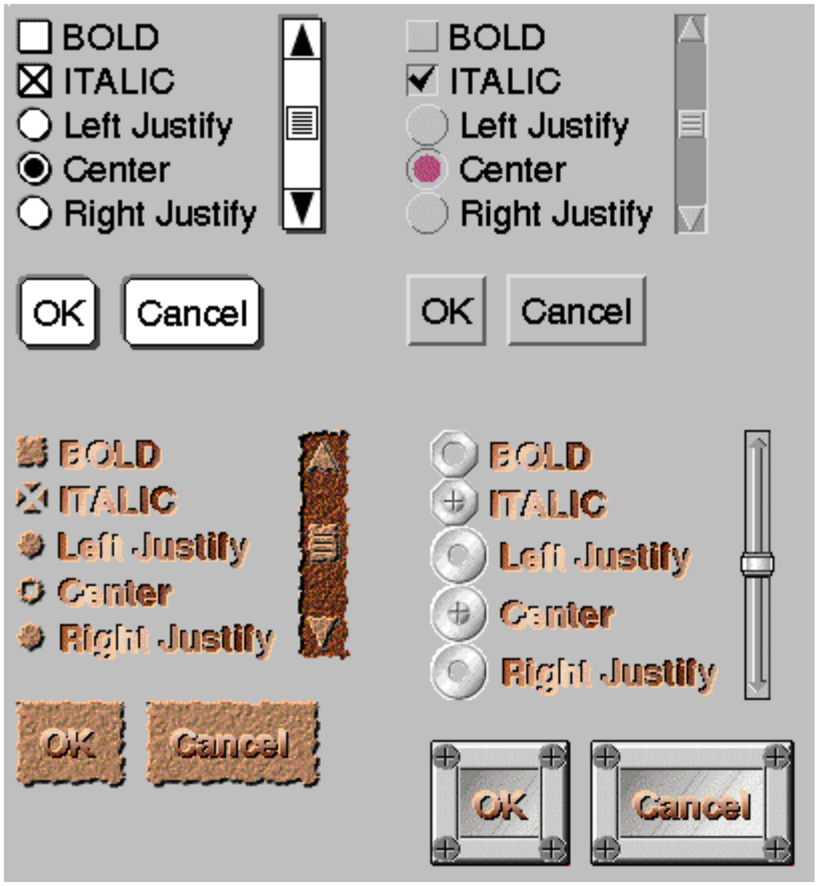

Figure 1. A Small Sample Interface Rendered in Four Different Styles.
A color version of this figure is available on the web at:
http://www.cc.gatech.edu/gvu/people/faculty/hudson/papers/style_system/style_sampler.gif

support for interactors which did not even exist when a style was created.

Figure 1 shows an example of several styles that have been applied to a sample set of interactors. The style at the top left is called DUMPY and is designed to be plain — reminiscent of the early black and white Macintosh interactors. The style on the top right is MOTIF-LIKE and is designed to closely resemble the Motif toolkit look. The styles at the bottom are designed to illustrate less conventional looks. The one on the left is called ADOBE-COPPER and, when displayed in color, it looks somewhat like adobe bricks with copper lettering. The final style is called INDUSTRIAL.

These examples give an idea of some of the kinds of appearances which can be supported by the system. It is important to note, however, that the styles presented here were created by the authors, who are definitely not graphic designers. As a result they are designed to illustrate the operation of the system, and to show some of the range of expression available, rather than necessarily to be visually appealing or highly functional.

A number of widely used existing toolkits allow some simple levels of customization of appearance, even by end users. Others have previously offered the ability to more substantially change appearance by supplying new drawing code, or even simply new images (see for example [Mye90]), and the area of platform independent, or *virtual*, toolkits (see for example [Roc92, Chi93, Gos96]) has worked to allow a single interface to be instantiated under several different platform specific toolkits with different look and feel. However, each of these systems have worked under the assumption of a relatively fixed library of interactive components and (with the possible exception of the Xt toolkit family [McC88]) have not specifically considered separation of style information from interactive objects, much less the dynamic application of styles to objects, or the notion of downloading a style object.

The work presented here builds on these previous techniques, extending them in several ways. These include improved support for both programmers and style designers, as well as new techniques to allow support to extend beyond a single fixed library of interactors.

In the next section, we will consider the basic constructs and mechanisms used to create appearances in the subArctic style system. Next an example will be considered. This will be followed by a description of the names used to designate style resources within the system, and then a description of how custom interactors are supported by the system. This will in turn be followed by a description of how style objects are implemented and how dynamic restyling of images is performed. Finally, system implementation and future work will be considered, along with some conclusions.

CREATING APPEARANCE

For a fixed library of fixed sized interactors, a great deal of appearance customization can be accomplished simply by basing interactor appearance on a set of images. In this situation, each type of interactor would expect to be provided with a particular type of image or set of images. For example a vertical slider might expect to be provided with "up button", "down button", "thumb" and "background pattern" images (along with versions of each corresponding to a depressed appearance and a disabled appearance). A style would then consist of a collection of named images in a lookup table. Under this design, dynamic style changes could be accomplished by switching to a new lookup table, informing each interactor that it may need to reconstitute its appearance, and if necessary, computing a new layout for the interface based on new sizes.

Use of images for appearance has a significant advantage for high production value interfaces. Images can be created and modified without programming using a variety of high quality interactive drawing programs now in wide use (for example, Adobe Photoshop [Ado96] or Fractal Design Painter [Fra95a]). Hence using images as the primary controller of appearance makes them fully accessible to graphics design professionals and makes available a wide range of easy to apply visual effects (e.g., via drawing program "plug-ins") that can provide a significant visual richness at low cost.

While use of display techniques such as realistic 3D rendering and new media such as video and audio are important for increasing the richness and expressiveness of interfaces, we also believe that there has yet to be a full exploitation of the richness of more conventional 2D graphical media — the richness that we currently see in the clickable image maps of web pages, but typically do not have in interactive programs. An image lookup approach to appearance can open up this richness.

Unfortunately, a pure image lookup approach to styles is not normally sufficient. In particular, it cannot handle objects whose size or contents is determined at runtime. For example, container objects which form a backdrop to other displayed objects must resize to fit those objects. Similarly, buttons, menus, and labels must normally be able to accommodate textual information supplied at run time (and in any case we would not want to require a new image to be drawn "by hand" for every button label). Consequently, an at least partially programmatic approach to interactor appearance is needed.

The subArctic style system presented here, uses an approach which is based partially on image lookup, but augmented with a simple programmatic mechanism. This mechanism is designed to be capable enough to support many different styles and effects, but simple enough that it can eventually be encapsulated in a tool only slightly more complex than a typical drawing program. The mechanism provides five basic constructs: *manipulation affordances, text renderers, blanks*, *look templates*, and *post processors*.

Manipulation affordances are individual components — normally images — that are designed to serve some specific purpose or suggest to the user the possibility of a specific kind of action [Nor90, Gav91a]. For example, many systems use a small down pointing triangle to indicate the presence of a pulldown menu. As a result many styles might provide such an image as a menu-pulldown manipulation affordance. Other examples include the "up button", "thumb" and "down button" parts of a slider, which are each designed to provide the visual cues that the object afford a particular type of action at a particular location.

For example, the image used for a slider thumb often has small simulated ridges. These ridges simulate the *knurling* often found on physical knobs and other surfaces in order to increase their friction with the fingertips when grasped. Although few people think explicitly about the function of knurling, it none the less affords grasp by the fingers because many graspable surfaces use it, and hence its appearance reminds us of those surfaces.

As described below, each style provides a library of named manipulation affordance images for a range of different common user actions. Further, as described later, these named affordances are taxonomized in a way that helps them to be robust and reusable.

Figure 2. Text from a Gradient Text Renderer

Text renderers are objects which are capable of turning a string into an image. The simplest text renderer encapsulates a font, size, and text style, and uses a single drawing command to render text in that font on demand. More complex text renderers

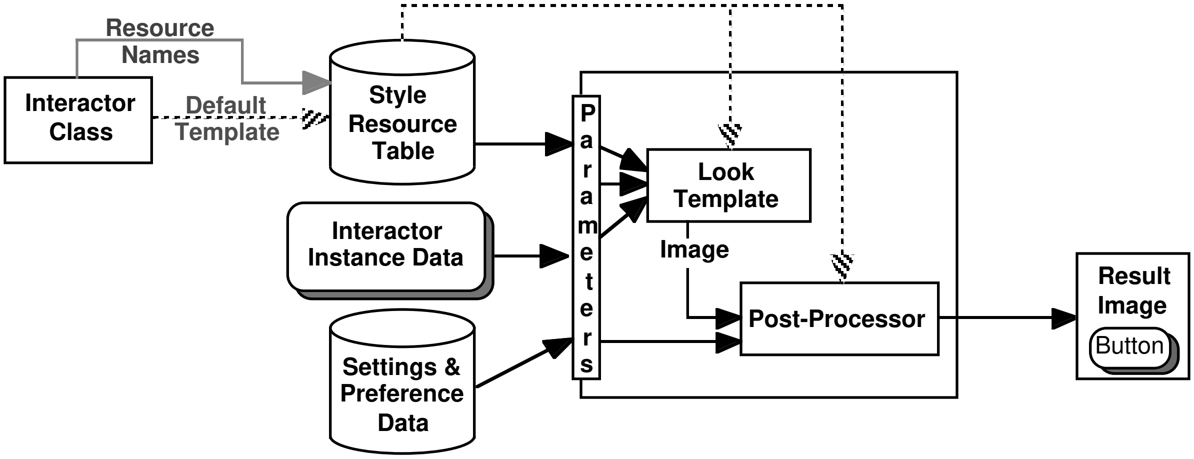

Figure 5. Composing Appearances

may do more sophisticated rendering. For example, the subArctic style system supplies text renderers for drawing shadowed text and for coloring text with gradients and patterns. The text shown in Figure 2, for example, is drawn using a gradient text renderer which (when displayed in color) simulates a copper appearance†. The subArctic system provides a small standard library of parameterized text renderers. In addition, particular styles can also introduce new text renderers as needed.

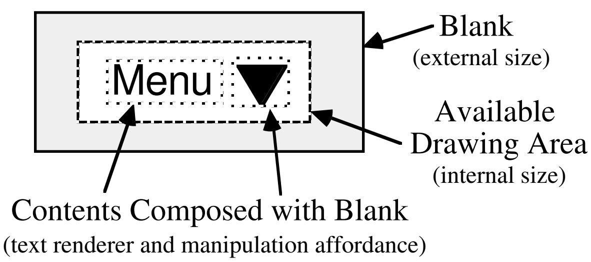

Figure 3. A Blank with Composed Components

Blanks are resizable graphical elements that are normally used as the backdrop upon which manipulation affordances and text are placed. For example the appearance of a button without its text label, would normally be supplied by a blank. Blanks may be created based on either an *internal* or an *external* size as illustrated in Figure 3. Internal size indicates the space available to the objects going inside the blank — for example, the text label in the case of the button blank. External size indicates the overall size of the resulting image, including any border at the edges.

The subArctic style system provides a small library of standard parameterized blank types. This includes several types of blanks that can be constructed from a series of images. As illustrated in Figure 4, one of the more commonly used blanks can be constructed from nine images — four fixed size corner images, four pattern images for the top, bottom, left, and right which are repeated horizontally or vertically to fill the respective edge borders, and one image which is tiled horizontally and vertically to fill the center area.

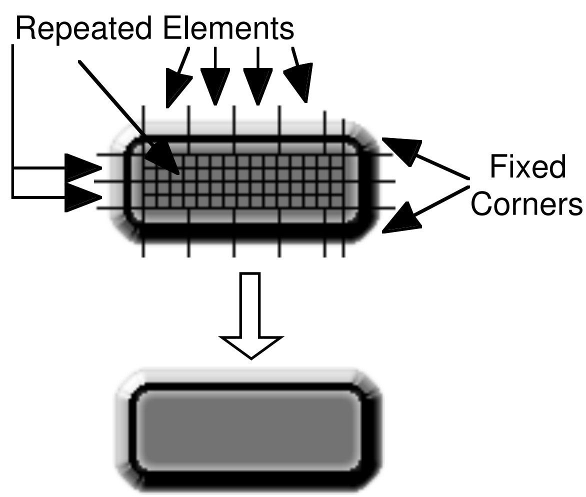

Figure 4. A Blank Created From Nine Images

Look templates are the primary image creation mechanism for the subArctic style system. As illustrated in Figure 5, each template expects a series of parameters. These parameters come from several sources and can include several different types of information. Among these parameter types are the objects to be composed (manipulation affordance

† The gradients used here are smooth interpolations between a sequence of colors, not simply two colors. With properly designed color sequences this technique can simulate a fairly wide range of materials and effects [Fra95b].

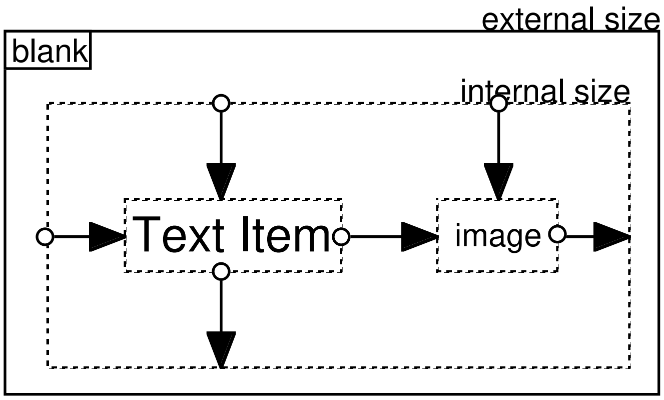

Figure 6. Example Constraints

images, text items, blanks, and even the images resulting from other templates), as well as objects such as text strings, colors, and integers that are used to specify final runtime details. Individual parameter values can be specified either concretely as simple values, or symbolically as names which can be looked up in the current style to obtain a value.

A look template provides a plan for creating a final image based on its parameters. The plan consists of two parts: an initialization of a series of *composition parts* (an object that will appear in the final image result — typically an image, text item, or blank), and a series of constraints for specifying the position and/or size of each part.

Initialization of each composition part is specified by indicating a type of composition object and the list of parameters and constants to be passed to its initialization routine.

The constraints used by look templates are simple one-way constraints [Mye90. Hud90, Hud91] taken from a fixed set of layout oriented constraints supporting things such as offsets, proportionality, and centering. (These constraints are similar to those described in [Hud96a]). Each constraint controls one coordinate of the bounding box of one composition part. Constraint equations may use values from parameters, as well as the coordinates from the bounding boxes of other composition parts, to compute their final values (cyclic constraints are considered an error and cause a run-time exception to be thrown). For constraints involving the coordinates of a blank, either the external or internal dimensions of the blank may be controlled and/or be used as input to other constraints.

Although at present the constraints of a look template are provided programmatically, they could also be easily specified using an a simple "arrow" notation similar to that found in the author's previous Apogee [Hud89] and Pluto [Hud90, Moh93] systems. This is illustrated in mockup form in Figure 6. Based on past experience and preliminary user tests with previous implementations of this notation, we believe it will make constraints of this form suitable for use by many non-programming graphic designers.

A template is instantiated by gathering together a series of appropriate parameters, using those parameters to instantiate each composition part, then executing each constraint equation (in topological order) to establish the final size and position of each part. Parameter values come from several sources. A number of parameters typically come from the style object. These style resources typically include the manipulation affordance images, text renderers, and blanks that control the primary appearance of the final image, as well as spacing and color values that may be associated with the style. Additional parameters can also be taken from a set of defaults or global preferences. Finally, parameter values often come from the interactor object requesting the stylized image (e.g., for the text string needed to create the label of a button).

Once a look template has been instantiated and laid out, it can easily be drawn simply by drawing each of its composition parts at their proper size and location. The resulting image is then optionally manipulated by a *post-processing object* provided by the style in order to add final effects to the image.

Post Processors are objects which accept an image, process its pixels in some way, and return an new image. As illustrated earlier in Figure 5, post processors are used in the subArctic style system to apply systematic visual effects to images. Their most common use is in darkening, "graying out", or blurring images to indicate a disabled status. They could also be used to systematically brighten or highlight images to indicate selection, or to apply other systematic special effects such as embossing.

AN EXAMPLE

As an example of using the constructs of the subArctic style system to create the appearance of an interactor, this section considers the example of a `menu_button` object. To construct the images needed to depict it in normal, depressed, and disabled states, the initialization code (*constructor*) of the `menu_button` makes requests of a style object (normally the global current style which is found via the `style_manager` object). The constructor calls the `build_image_set()` method of the style object, passing the name of the image set being requested, along with the text string forming the label of the button.

As illustrated in Figure 7, this style object provides a look template, along with a background blank, a text renderer, and the image for a "menu pulldown"

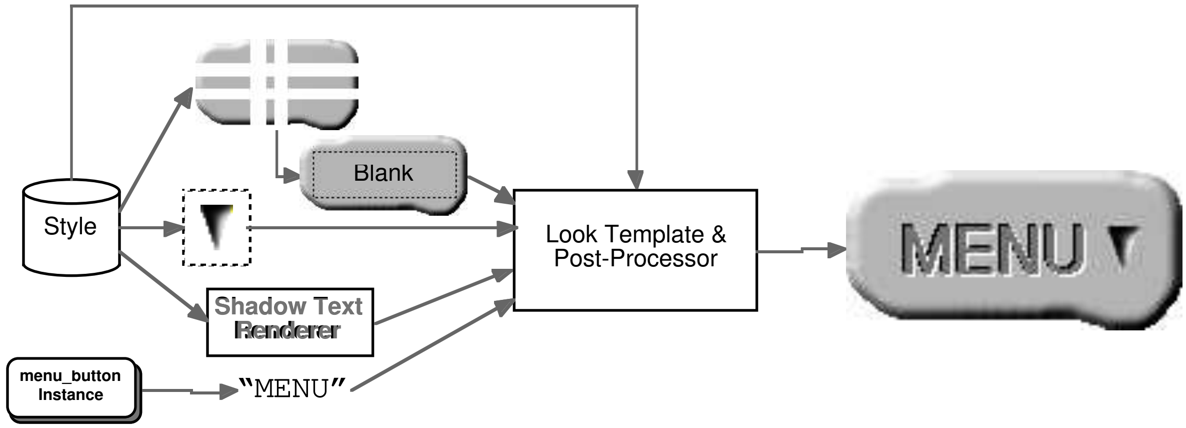

Figure 7. Example Image Layout

affordance. In this case, the background blank is constructed from images (also supplied by the style). The look template constraints (previously illustrated in Figure 6) place the text item and affordance image within the blank and define the internal size of the blank to be large enough to hold the items, plus a small border. The final result (for the normal appearance) is shown at the left of Figure 7. The style applies a similar process to create images for the depressed and disabled appearances (using a related blank with reversed highlights and a "dimming" post processor, respectively), then returns all three images in an array. Finally the `menu_button` object uses this array of images to draw its appearance as needed.

Note that the interactor object using images created in this way does not know or care about the exact appearance of the images it uses. For interactor objects made up of individual manipulatable parts whose size or position is dynamic (such as a scrollbar with a sliding thumb), the style system can also be used to produce individual components, and the interactor can then compose them "manually" to produce a drawn final result rather than a static image.

STYLE RESOURCE NAMES

Style objects are primarily lookup tables. They map the names of various resource objects to the instances of those objects appropriate to the style. Because quite a few different names are needed to fully implement the appearance of a complete toolkit, a structured naming convention is used. Under this convention, style resource names have the form: *`package.class.name:variant`*

The first two parts of the name contain the fully qualified class name of the interactor class which "owns" the name — that is which uses the resource. This follows the Java convention where fully qualified class names are of the form *`package.class`*. The use of packages provides a simple mechanism for segregating the Java programming name space, and using the same mechanism here eliminates cross-package name collisions (since such collisions would have to have been previously resolved in order to properly compile the program). Classes in the standard subArctic library use the shorthand package name "`#std`" to increase brevity and avoid conflicts for resources which are global in nature.

The third part of a resource name is simply the resource identifier. This normally indicates the function of the resource. The final part of a resource name indicates one variant among several related resources. For example, the names:

```
#std.button.label_look:out
#std.button.label_look:in
#std.button.label_look:disabled
#std.button.label_look:set
```

name the look templates used to create labels (in normal, depressed, and disabled form) for the `button` interactor class found in the standard toolkit library, along with a resource consisting of an array containing these three templates.

SUPPORTING CUSTOM INTERACTORS

While the mechanism described above is quite capable of supplying images for styles with a wide range of different appearances, as described thus far, it will only work with a fixed library of interactors which has been established in advance of style creation. A central goal of the subArctic toolkit is to support the easy creation of new interactors. However, supporting new interactor classes with existing styles is a very difficult challenge. In some cases, this may simply not be possible because the new class introduces a completely new functionality which requires completely new images. Fortunately, in many cases the basic actions of new interactor classes (such as "clicking" or "dragging") are related in functionality to the basic actions of existing classes, and may be able to reuse manipulation affordance images. The subArctic style system includes facilities to try to maximize support for custom interactors, and provide fallbacks to make it more likely that new interactor classes that do not have specific support built into a particular style, may still be supported by that style (perhaps in a somewhat degraded form).

To do this, the subArctic style system uses the concept of manipulation affordances. Each manipulation affordance image provided by a style is classified as to its specific purpose. In addition these purposes are taxonomized and generalized into a hierarchy of manipulation affordances.

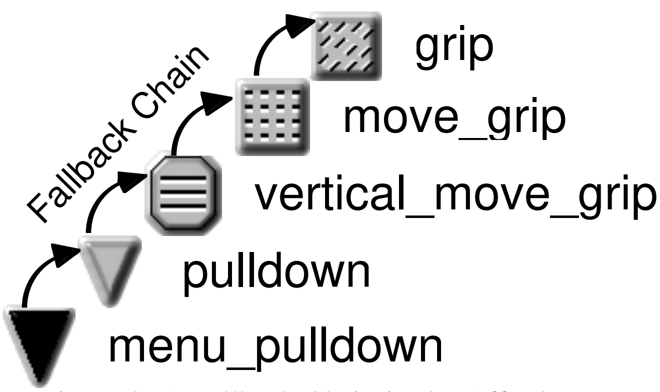

Figure 8. A Fallback Chain in the Affordance Hierarchy

Figure 8 shows a partial example of this. The menu_pulldown affordance used in our previous examples has a very specific purpose — to indicate the presence of a pulldown menu. A slightly more general affordance is that of a pulldown for any purpose. This in turn is an instance of a vertical_move_grip — an affordance for "grasping" then moving (dragging) vertically. This in turn is a special case of a move_grip — an affordance for grasping and moving in some direction. This finally is an instance of a grip affordance — one which is designed to be grasped in some way.

Figure 8 shows one path in the affordance hierarchy along with the image resources that a hypothetical style might supply for those names. Examples of other parts of the hierarchy include vertical_slider_thumb and vertical_resize_grip both of which are specializations of vertical_move_grip, as well as corner_resize_grip which is a specialization of move_grip. The full affordance hierarchy consists of a classified set of affordance trees. Each name used by the style system is placed in this affordance hierarchy.

Currently, root affordances are provided for the following categories: grip, press, text_display, box, backdrop, separator, font, color, and color_scheme. This set of affordance trees is thought to cover the interactive objects found in most GUI toolkits. However, the full affordance hierarchy is still under development, and additional experience will be needed before a full assessment of coverage by this set is possible.

New interactor classes may introduce new affordances to suit their specific needs and register the name of that affordance with the style system. However, each such name must also have a *fallback link* which indicates another affordance in the hierarchy which serves a more general purpose. This effectively places each custom affordance within the affordance hierarchy.

Whenever any affordance name is requested from a particular style, the style returns the "best" resource available. In particular, if the specific resource being requested is not available, then the fallback chain (corresponding to a path up the affordance hierarchy tree) for that affordance is consulted to find a resource that the style actually contains. The system provides a set of required standard affordance names which must be supported by every style. As a result, a request for any affordance which has been properly linked into the affordance hierarchy will always be fulfilled. Further, the style will always return the resource which is semantically closest to the requested resource (where *semantic closeness* is defined by the affordance hierarchy).

Although custom affordances may in some cases not be closely related to any standard affordance, this scheme of fallbacks provides significant robustness and allows for graceful degradation. Since resource requests are always fulfilled, custom interactors are guaranteed to be at least operable under any style. Further, the interactor designer is normally able to judge how closely matching the closest standard affordance is and see what this looks like in a variety of existing styles.

In cases where the closest standard affordance may be an awkward match, the interactor designer has at least two additional alternatives. As described below, styles can easily be extended via inheritance. This allows the designer to easily extend several existing styles to work with their new interactor. As a final alternative for cases where existing fallback affordances are not acceptable substitutes, it is also possible for a custom interactor to override the fallback mechanism and supply a default resource which will be used for all styles which do not explicitly provide a particular resource.

STYLE OBJECTS

As indicated above, style objects are primarily collections of images, look templates, and other resources, indexed by a hierarchical set of names. However, in order to make the creation of styles as easy as possible, the system allows styles to be constructed based on other styles. In particular, a form of delegation-based multiple inheritance is supported. Each style may declare a list of *super-styles* which are consulted if a particular named resource is not found locally within a style. This list of super-styles is consulted in order. When looking for a particular named resource, the local lookup table of the style is consulted first. If the named resource is not found, then a complete search of the first style in the super-style list (implicitly including its super-styles) is performed. If the resource is not found there, then the second super-style is consulted, and so forth. This scheme makes it easy to make small additions or changes to styles, and allows partial or *mixin* styles to be created that can be easily composed to create full styles.

All styles directly or indirectly inherit from a special style called `root_style`. This style provides a resource for all required standard names and in general provides a set of guaranteed final fallbacks for all styles. In order to increase the modularity of the system, each interactor class in the standard library is responsible for both registering the names it owns, and adding final fallback resources for those names to the root style. In addition, custom interactors may add resources to the root style if they wish to require use of a given default resource in preference to any available fallback resources.

Note, that the resource name fallback system described in the previous section interacts with style inheritance[†]. A complete search is performed as follows: first a full search of the style is performed (implicitly including all its super-styles) looking for the exact name requested. If this search fails, then the first fallback name is searched for, etc. until a suitable fallback resource is found.

While this full search process may seem somewhat complex, in actual practice it does not cause undue difficulties because the two primary users of the system — the style author, and the interactor implementor — are each almost exclusive concerned with one aspect of the system. Style authors are normally only concerned with the style inheritance part of the system, and simply consider what resources are needed to cover the set of required and optional resource names. Those who develop interactors, on the other hand, typically do so in a style independent fashion, concerning themselves only with the name fallback system and, if they are considered at all, treating styles as monolithic in nature.

[†]In fact the name fallback system is itself really another form of inheritance. We choose to use different terminology for it primarily to avoid some of the confusion that results from two simultaneous inheritance hierarchies.

A final issue concerning style objects is the capability to download them. In the subArctic system style objects are currently each implemented by a new Java subclass with resource loading performed by static code and tables. This scheme allows styles to be downloaded with the normal Java dynamic class loading facilities. We are currently investigating the use of object serialization capabilities [Rig96] to allow pre-initialized style objects to be placed on persistent storage and reloaded on demand.

RESTYLING STYLE CONSTRUCTED IMAGES

Another goal of the subArctic style system has been to allow dynamic style switching — that is to allow the style applied to an interface (or to an individual interactor) to be changed while the interface is running. This is useful for end user customization, and other occasions that involve modifications to live running interfaces. However, perhaps more interestingly, this capability is useful for migratory interfaces. In this situation we would like to be able to move a running interface from, for example, a color desktop machine, to a black and white laptop, or to a hand-held PDA. In these cases it would be extremely useful to change styles so that a style designed to look good in black and white, or a highly compact style, could be used.

On the other hand, the ability to change styles dynamically is not always of central importance, and in the overall design of the style system we wish to keep the style system as simple as possible so that programmers see it as a resource to make their job easier, rather than as another thing to be implemented or conformed to. As a result, a very simple API for restyling is needed.

To achieve this, a design was constructed whereby any image created by the style system could be automatically restyled by another style. To restyle, the interactor programmer needs only to re-request non-image resources from the new style, pass each of its images to a new style for restyling, and then recompute its size. It need not, for example, explicitly store things such as the text string used to generate a label. (Recomputation of the overall user interface layout is handled in subArctic by means of constraints, so that part of the restyling process also happens very automatically, as a side effect of changing the size of the objects involved.)

To implement restylable images, the system uses a small "trick" of object oriented programming. Rather than return images or arrays of images from style operations, a special subclass of image is returned instead. This subclass contains additional bookkeeping attached by the style system. Since this object is a subclass of the normal toolkit image class, the interactor can draw it or perform any other image operation on it as normal. However, when asked to restyle the image object, the style system can use the runtime type query mechanism of Java (the `instanceof` operator) to detect that it is actually an instance of the special `styled_image` class used by the style system. (If the programmer has substituted a custom image, this can be detected as well, in which case the original image is returned unmodified).

The bookkeeping attached to each image object includes all the resource names (including things such as the look template name and the names of any parameters supplied symbolically), and specific parameter values originally used to create the image. These names can then be looked up in the new style and combined with any specific parameter values to recreate the equivalent of the image under a new style. By placing the bookkeeping necessary to recreate an image in "hidden" data within the image we can effectively keep most of the work of restyling an image inside the style system and very little extra work is need from the programmer.

Once a system for storing information about the nature of images is in place, it is possible to use that information in other ways as well. For example, it is possible to extract the set of manipulation affordances used in constructing the display of a particular interactor. Since manipulation affordances are categorized as to purpose, this information can be used to extract information about the interactions supported by the interactor without requiring that information to be explicitly provided by the programmer.

We are currently exploring techniques for using this and other hidden or implicit information to obtain some of the benefits of model based user interface development environments [Piy93, Sze93] without requiring the explicit construction of a model. For example, we are currently exploring the use of this information to transform interfaces from the GUI domain into the audio domain (using techniques similar to the Mercator system [Myn92]). Here the challenge is again to deal intelligently with custom interactors that go beyond a fixed library. For these audio translations, we may be able to use embedded information about the affordances of an object's display to synthesize an audio icon [Gav91b] that conveys some or all of those affordances in the audio domain.

IMPLEMENTATION AND FUTURE WORK

The style system described in this paper has been implemented in approximately 8000 lines of Java code (not including individual style objects) as a part of the subArctic user interface toolkit. At the time of this writing, some of the toolkit library is still being converted to use the style system and the full manipulation affordance hierarchy is still being enumerated (on the basis of the components of the subArctic interactor library as well as a survey of other toolkits). Although we cannot at this time fully assess the coverage that will be provided by the affordance hierarchy, we believe that, because there is a relatively small set of basic operations in a graphical interface (such as clicking and dragging) it will converge over time to provide coverage for nearly all interactive objects in a GUI setting.

A major area of planned future work is in the creation of the style designers interface. In this area we plan to provide an interface for conveniently "cutting" small image components from large images (that would typically be created in a commercial drawing or painting program), a graphical layout constraint editor (similar to the one illustrated in Figure 6, and described in [Hud90]), and an environment for exploring the appearance of objects drawn with partially completed styles in the context of example interfaces.

Additional areas of future work include adding serialization [Rig96] support for storing and downloading style objects as data instead of code, as well as a number of new text renderer, post processor and blank implementations.

CONCLUSIONS

This paper has described the design and implementation of the subArctic style system. This system provides an infrastructure for the dynamic application of a coordinated set of interactor appearances in order to implement a consistent and rich look for an interface — a style. These techniques are designed to be easy to use for both the programmer and (eventually) the style author, with the eventual goal of allowing appearances to be applied to interfaces nearly as easily as fonts are currently applied to text. Further, the system provides the capability to support many custom interactors, even in cases where interactor classes were created after the styles that support them.

REFERENCES

[Ado96] Adobe Systems Inc., Adobe Photoshop™ Version 4.0 User Guide for Macintosh and Windows, San Jose CA, 1996.

[Chi93] Chimera, R., Barr, J., Brunecky, M., Pausch, R., Rappaport, A., Platform Independent User Interface Builders: Where are we Headed?, *Proceedings of the ACM Symposium on User Interface Software and Technology*, pp. 235-236, November 1993.

[Fra95a] Fractal Design Corp., Fractal Design Painter™ 4 User Guide, Aptos, CA, 1995.

[Fra95b] Frank, S., Kai's Power Tools™ 3: Explorer's Guide, MetaTools Inc., Carpinteria. CA. 1995.

[Gav91a] Gaver, W., Technology Affordances, *Proceedings of the ACM CHI '91*

Conference on Human Factors in Computing Systems, May 1991, pp. 79-84.

[Gav91b] Gaver, W., Smith, R. O' Shea, T., Effective Sounds in Complex Systems: The ARKola Simulation, *Proceedings of the ACM CHI '91 Conference on Human Factors in Computing Systems,* May 1991, pp. 85-90.

[Gos96] Gosling, J., Yellin, F., The Java Application Programming Interface, Vol 2: Window Toolkit & Applets, Addison-Wesley, NY, 1996.

[Hud89] Hudson, S., Graphical Specification of Flexible User Interface Displays, *Proceedings of the ACM Symposium on User Interface Software and Technology*, pp. 105-114, November, 1989.

[Hud90] Hudson, S. , Mohamed, S., Interactive Specification of Flexible User Interface Displays, *ACM Transactions on Information Systems*, v8, n3, pp. 269-288, July 1990.

[Hud91] Hudson, S., Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update, *ACM Transactions on Programming Languages and Systems*, v13, n3, pp. 315-341, July 1991.

[Hud96a] Hudson, S., Smith, I., Ultra-Lightweight Constraints, *Proceedings of the ACM Symposium on User Interface Software and Technology*, pp. 147-155, November 1996.

[Hud96b] Hudson, S., Smith, I., SubArctic UI Toolkit User's Guide, WWW document http://www.cc.gatech.edu/gvu/ui/sub_arctic/sub_arctic/doc/users_manual.html, 1996.

[McC88] McCormack, J., An Overview of the X Toolkit, *Proceedings of the ACM Symposium on User Interface Software and Technology*, Oct. 1988, pp. 46-55.

[Moh93] Mohamed, S., End-User Specification of Interactive Displays, Ph.D. Dissertation, University of Arizona, Department of Computer Science, 1993.

[Mye90] Myers, B., Guise, D., Dannenberg, R., Vander Zanden, B., Kosbie, D., Pervin, E., Mickish, A., and Marchal, P., Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces, *IEEE Computer,* 23(11), Nov. 1990, pp. 71-85.

[Myn92] Mynatt, E., Edwards, K., Mapping GUIs to Auditory Interfaces, *Proceedings of the ACM Symposium on User Interface Software and Technology*, November 1992, pp. 61-70.

[Nor90] Norman, D., The Design of Everyday Things. Doubleday, New York, 1990.

[Piy93] Piyawadee, S., Foley, J., Griffith, T., A Second Generation User Interface Design Environment: The Model and the Runtime Architecture, *Proceedings of the ACM INTERCHI '93 Conference on Human Factors in Computing Systems*, Amsterdam, May 1993, pp. 375-382.

[Rig96] Riggs, R., Waldo, J., Wollrath, A., Bharat, K., Pickling State in the Java™ System, Computing Systems, 9(4), Fall 1996, pp. 265-290.

[Roc92] Rochkind, M., An Extensible Virtual Toolkit (XVT) for Portable GUI Applications, *Proceedings of COMPCON '92*, Spring, 1992, pp. 485-494.

[Sze93] Szekely, P., Ping, L., Neches, R., Beyond Interface Builders: Model-Based Interface Tools, *Proceedings of the ACM INTERCHI '93 Conference on Human Factors in Computing Systems*, Amsterdam, May 1993, pp. 383-390.