23

Presentation Architecture

In chapter 2 we discussed a presentation model where applications indicate regions of the screen that are out of date by invoking a damage() method on the window. The windowing system, for various reasons, can call redraw(Graphics g) on a widget to have it redraw itself. The Graphics object defines the interface to virtually any display surface including windows, printers, images or other display devices. This basic presentation architecture is the basis for almost all modern graphical user interfaces.

The basic damage/redraw architecture has a number of interesting variations that have become the basis for a variety of research systems. Some of these architectures are beginning to move into commercial products. In this chapter we will look at three such variations. The first is to use the Graphics object interface for very different purposes than its original intent. By creating special subclasses of Graphics a variety of features can be implemented that work across all applications. The second technique is to render windows onto in-memory images rather than on the screen and then use those images to redirect the interface in new ways. In particular we can adapt and change the nature of the user interface without requiring the application's cooperation. The last technique is to represent the presentation as a tree or graph of objects. The application modifies this tree and the damage/redraw mechanism is invisible.

Redirecting the *Graphics* object

In an object-oriented architecture the Graphics object or some similar class provides the universal abstraction of a drawing surface. Application views only know about this class and its methods. The view code is independent of what is actually done with the information in the Graphics object.

Magic Lenses

The first system to exploit the Graphics object was the Magic Lenses of Bier, et. al.[1] Magic lenses are special windows that can be dragged over the top of other windows and modify their display. They can do many things like produce an achromatic view such as a color blind person might see (figure 23.1), Separate

closely spaced objects for easier selection (figure 23.2), eliminate the fills from shapes so that shapes behind can be easily selected (figure 23.3), or magnify the underlying shapes (figure 23.4).

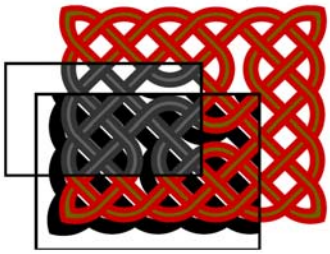

Figure 23.1 – Achromatic Magic Lens[1]

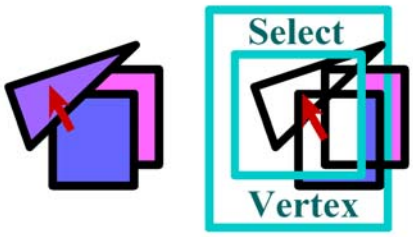

Figure 23.2 – X-Ray Magic Lens[1]

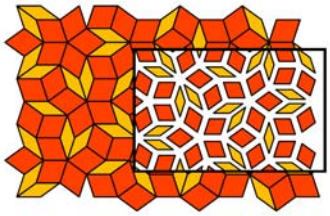

Figure 23.3 – Separator Magic Lens[1]

Figure 23.4 – Magnifying Lens

For the achromatic lens (figure 23.1) a window is placed over the original view. When the lens window receives a call to redraw(Graphics gDraw) it takes the Graphics object gDraw that it received and wraps it in a special subclass of Graphics to produce an object gLens. The lens then calls the redraw() method of the underlying window passing in gLens rather than gDraw. The implementation of gLens will pass through any method call to gDraw with the exception of the methods to set colors. When one of the methods to change the drawing color is called, the color is first modified to a saturation of zero and the new color is passed to gDraw. The underlying application believes that it is drawing on a normal Graphics object. However, what are actually being drawn are the modified colors. The underlying window still responds to redraw() calls in the same way but the lens window, being over the top, is what the user sees.

For the X-Ray lens (figure 23.2) a modified gLens object translates all method calls that draw filled shapes into the corresponding calls for unfilled shapes. Thus objects behind show through. For the separator lens (figure 23.3) a modified gLens object first computes the center of each shape to be drawn, scales the shape down a small amount about its center and then calls gDraw with the smaller shape. Each shape appears to draw away from its neighbors. The magnifying lens simply applies a scaling transformation to all draw calls before passing them on to gDraw. Creating a new type of lens simply involves creating a different wrapper class that is a subclass of Graphics but modifies the calls in some way before passing them on to the Graphics object that the lens received in its own redraw() call.

One of the side effects of this architecture is that lenses can be placed over other lens in a layering effect. For example a magnifying lens can be placed over an X-ray lens. When redraw() is called on the magnifying lens it wraps the Graphics object D in a magnifying object M and calls redraw(M) on the window below. The X-ray lens below takes the drawing object M and wraps it in a Graphics object X that implements the X-ray transformation and calls redraw(X) on the application

window below. When the application window draws a rectangle it will call X.drawRect(..). The implementation of X will strip out the fill information and call M.drawRect(..) because X is a wrapper for M. The magnify object M will scale the rectangle and call D.drawRect(..), which will draw an unfilled magnified rectangle in the window. All of the lenses and applications know nothing about what each other is doing. All communication is through redraw() and specialized subclasses of Graphics.

Edwards et. al.[2] added a number of other techniques to the repertoire of graphics object modification. They added drop shadows to objects using a two pass painting technique. They would first call redraw() with a Graphics object whose only color choice was gray and with a transformation down and to the left. This drew a shadow of all objects in the presentation. Then the original redraw() was called with the unmodified Graphics object so that the real content was drawn over the shadows. They also demonstrated how animation effects such as shimmering can be added by periodically recalling redraw() using an incrementally modified transformation.

Attachments

By definition the redraw() method will reveal all of the user-related information about an application because drawing on the screen is the primary mechanism for communicating to the user. Because all of this information is being passed through the Graphics object we can exploit that information for other purposes. This is the basis for the Attachments[3] system. Unlike the lens technique above, attachments required additional cooperation from the application in three limited ways. First the application's redraw() method should generate a drawing for all of the application rather than just the currently visible portion. The Graphics object may have a clip region that could be used to limit the redraw effort, but if the clip region was infinite, all of the information should be drawn. For example a word processor should be prepared to draw all of its pages in one long vertical presentation if requested.

The second attachment requirement was that the application should reveal its structure through four methods that were added to the Graphics object. These were:

```
groupStart(String name, String type)
groupStart(int index, String type)
groupStart(Interactor toolkitObject)
groupEnd()
```

The idea is that most redraw() methods are implemented to tour their model structures as a tree traversal. Whenever the application starts to draw some model object it calls one of the groupStart() methods to identify the object and its type. Objects are identified by name or index. The object types are any names that were meaningful to the application. Identifying an object by interactor supplies both identifier and type information. When an object's drawing is complete groupEnd() is called. When complex objects are drawn, the start/end associations are nested. These methods reveal a tree structure of how objects being drawn on the screen are associated together in meaningful application objects.

The third requirement was that the application reveal its current selection using a path mechanism that used the same names or indices found in the groupStart() methods. When the application selects an object this path representation indicates what is selected. When some attachment wants to modify the selection it sets the path. When the selection is modified the application should scroll so as to display that selection.

Based on these three requirements of applications (full world redraw, start/end grouping, and path selection) plus the information normally supplied by calling methods on the Graphics object, a variety of attachment tools can be created that augment any application.

The first example is a "bookmark" facility. The bookmark attachment allows a user to save the path of the currently selected object for future reference. A bookmark attachment might name these marks or save them in some other way, the applications do not care. When the user selects a bookmark, the selection is passed back to the application to scroll the selected object into view. A more interesting technique is to create a "radar view"[4] of the bookmarks. A radar view shows where the bookmarks are within the world space of the application. A radar view is generated by invoking the application's redraw() with a special Graphics object that knows about the bookmarks. By watching the groupStart/groupEnd calls the Graphics object knows when one of the bookmarked objects is being drawn. This subclass of Graphics does not actually draw anything. Instead it retains the bounding boxes for any drawing of a bookmarked object. From these bounding boxes a scaled down version of the application's entire world can be displayed with the bookmarked object locations highlighted. If the user selects one of the bookmarks in this space the selection of the application is moved to that location. Vertical and horizontal radar views are possible by discarding X or Y from the bounding boxes. These radar views can be used to augment scrollbars. This bookmark attachment and its radar views are independent of any particular application.

The bookmark attachment illustrates the need for attachments to highlight objects in an application's presentation to direct the user's attention. This is problematic because there are so many ways that applications may present their data. For example highlighting an object by drawing a red box around it is a good idea until the application has a red background or many red objects. The generalized pointing[5] technique was developed for attachments. When the redraw() of an application is called, a special Graphics object that knows about the bookmarks is used. Whenever objects are being drawn that are not part of a bookmark (as indicated by groupStart/groupEnd), their colors are blended with a neutral color such as light gray. Objects that are part of bookmarks or other selections are drawn in their normal colors. This technique deemphasizes unselected objects. The more background objects are blended to a neutral color, the more prominent the selected items. The less background blending the more visible the unselected objects and able to provide context. Figure 23.5 shows this technique in highlighting a map location without loosing context. Again the application program is unaware of this highlight mechanism because it is just a modification of the Graphics object.

Figure 23.5 – Blended highlighting

Using the above mechanisms a spell checker attachment can be written. To spell check any application the redraw() method is called with a Graphics object that ignores all calls except text drawing calls. The text is not drawn but is checked against a spelling dictionary. For any text not in the dictionary, the path of

groupStart identifiers is added to the bookmarks. When done, all of the objects that contain misspelled words can be highlighted using the previous techniques. A similar technique could be used to provide dictionary lookup or language translation for selected objects. The key is that applications reveal substantial information through redraw()/Graphics objects and that information can be usefully mined.

Image space architectures

All windowing systems must eventually render the application's presentation as an image on the screen. For object-oriented systems this rendering is encapsulated in the Graphics object. In most windowing systems the pixel representation is stored in the frame buffer of the display where it can be seen by the user. As compute power and RAM have increased it has become possible to separate the rendering of an application presentation into pixels and the compositing of those pixels onto the screen. The windowing system can first render the presentation into an image and then in a separate step render that image onto the display. This is the architecture built into Apple's Quartz window compositor.[6]

Once the application's presentation is represented in an image there are a variety of things that the windowing system can do. In Macintosh OS X windows appear to shrink and "pour" when iconified and then expand out of the icon when opened. This effect is simply a distortion of the window's image. Rendering to an image also allows for redraw of an exposed portion of a window without invoking the application's redraw() method. The exposed segments are simply composited onto the screen from the window's image. This technique was used in the old Blit[7] terminals before the object-oriented damage/redraw architecture was invented. The image representation can also be scaled down to create smaller overviews of a window. This is the basis for Apple's Expose window management. When desired the windowing system will shrink all of the window images and move them apart so that none of them overlap. This gives the user a full view of all of the windows from which the desired window can be selected. A similar technique is behind the thumbnail views in Scalable Fabric.[8] Representing windows as images is behind the VNC collaboration architecture.[9] Once a window is rendered as an image it becomes a data object that can be used for a variety of purposes and in a variety of ways.

Metisse

These architectural ideas were formalized in the Metisse[10] windowing system. Metisse is implemented as a special X-Windows server. The graphics calls are received in the traditional way and then rendered into off-screen images. These images are then passed to the *compositor* that is responsible for rendering the images on the screen. Changes to the compositor allow for many different window management techniques to be implemented without requiring the cooperation of the application. This is further simplified by the X-Windows approach of separating window management from the windowing system. The compositor is simply a special window manager. Figure 23.6 shows some of the new ways to render windows.

When window images are distorted, modified and moved around in new ways, input handling becomes an issue. In Metisse interactive inputs are received by the compositor and the appropriate window is identified to receive the event. The compositor then applies the inverse transform or distortion to the input point before passing the event on to the application through the X-server. Thus the application feels and behaves appropriately without knowing about what has been done to its display on the screen.

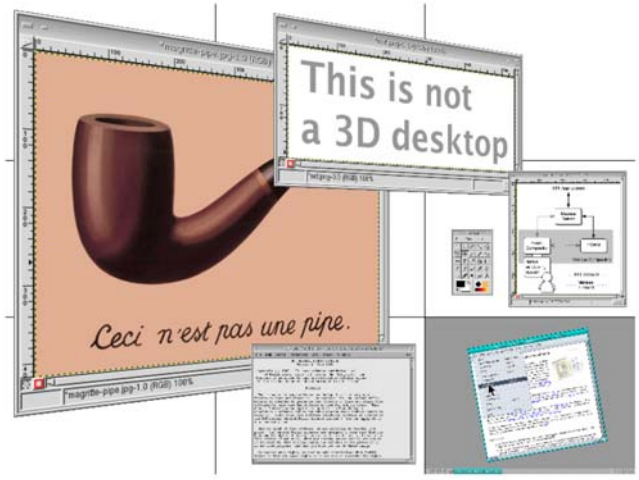

Figure 23.6 – Metisse image-based rendering of windows

Facades

The flexibility of separating image compositing from graphics rendering in Metisse has been used to implement Facades.[11] Facades are a mechanism for

adapting a user interface to the needs of a particular user without requiring the application to cooperate. The user can open a special Façade window and then can copy and paste user interface fragments from existing applications. The fragments included in a façade need not be all from the same application. Using the Metisse architecture these fragments can be rendered into memory images and then rendered on the screen wherever the fragment has been pasted. When a façade receives user inputs the events are mapped back to the original locations in the off-screen window images. When a window is damaged and redrawn, the off-screen image of the window is repainted and the façade is then updated from the off-screen image. WinCuts[12] is a similar tool built on top of Microsoft Windows. To the user the interactive fragments appear to be live parts of the original applications. An application can now provide a large and complex palette of controls and the user can create a façade of only the controls that they use most frequently. The assembled façade can be copied from many different parts of an application's controls.

Though image-based techniques are quite effective in creating facades the user can be assisted in selecting a specific control. Facades does this by accessing the window tree and specifically the accessibility information for the blind stored the tree. The accessibility information is similar to what Attachments got from groupStart/groupEnd. The information indicates the bounds and function of regions of the screen. These can be used to simplify selection, copying and pasting fragments into a façade.

Mnemonic Rendering

A last image-based windowing technique is called Mnemonic Rendering.[13] When a display is very large, such as a 30 foot wall display, the user may not be paying attention when an application on a distant part of the screen makes changes to its presentation. When the user's attention is transferred to that part of the screen the user needs to be reminded what has happened while they were looking elsewhere. This is a feature that we want to work on all applications, not just those that have been specially modified. A similar situation occurs when a window is partially or fully hidden and some change occurs to its presentation. When the window is exposed the user wants to recap what has changed.

As an application makes changes its window presentations are rendered into an off-screen image, as in Metisse. In addition to the rendering and compositing, the windowing system keeps a history of the changes to the window image. When the user's attention is directed at the window or the window is exposed, depending on the usage scenario, the changes are *restituted* on the screen so the

user can see what has happened. Mnemonic Rendering proposes two restitution techniques. The *persistence* technique shows a pixel-by-pixel blend of the historical pixels and the current ones. As the blend moves forward in time, the image shifts from its historical value to the current one. If there has been movement, this will appear as *motion trails* across the screen. The *flashback* technique rapidly displays each historical change in fast time so that the user can quickly see all that has happened. These techniques are possible over any set of windows because of the off-screen rendering and use of the window image as a data object to be manipulated by the windowing system.

Triggering the mnemonic display depends on how the technique is being used. If the display is large then tracking the user's attention is important. This can be done by gaze tracking or by head motion tracking. The mouse or other pointer location can be used as a surrogate for the user's attention. For example, touching a new region of the screen can trigger the mnemonic rendering. For smaller displays where mnemonic rendering is used on windows that were previously concealed, the window exposures can be used to trigger a mnemonic rendering.

Scene graphs

Graphics object and window image techniques work at the outer most levels of the display system and thus minimize their entanglement with actual applications. These techniques make it easy to provide functionality that is independent of any application and impose minimal requirements on the application's implementation. As was seen with the Attachments work, it helps to have additional information about the structure of the application's information. There are a series of architectures that make the structure of the presentation more explicit and thus make more information available to external tools.

Most of these techniques revolve around the idea of a scene graph and object-oriented techniques in implementing the nodes of that graph. Scene graphs are sometimes called structured graphics. The ideas are the same. In the very early days of computer graphics a scene to be drawn was represented as an interconnected set of nodes, each node contained a list of drawing specifications. This was known as the *scene graph*. The old graphics hardware would traverse this structure 30+ times a second to redraw the scene on the display (see figure 23.7). To modify the display the application program would modify the contents of the scene graph. The next time the graphics hardware traversed the scene graph the image would be drawn differently and thus would instantly change.

There was no damage/redraw. The approach was to change the scene graph and the display would instantly change.

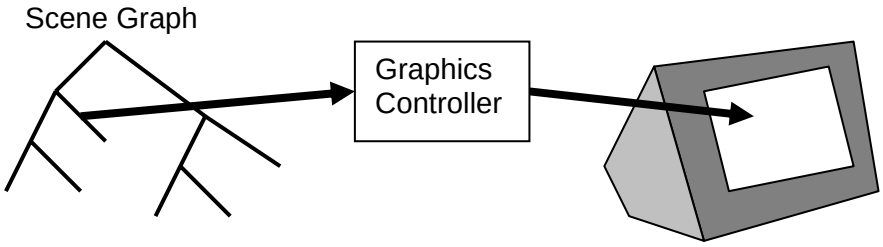

Figure 23.7 – Scene graph refresh display hardware

Scene graphs disappeared from 2D interface architectures with the advent of frame-buffer graphics where the graphics model is pixel-based rather than stroke-based. The display hardware now refreshed from the frame buffer rather than the scene graph. The damage/redraw architecture took care of getting the screen updated when changes occurred. Scene graphs continued to be used in 3D architectures and animation tools. In 3D architectures, the hidden surface problem made the identification of a damage region quite difficult. The problems of transparency and cast shadows also made it necessary for the rendering tools to have a structure that could be traversed as needed to correctly produce the desired picture. In animation, there are many changes going on at the same time. The scene graph architecture allows the animation engine to make the changes that it needed at the times that were needed while leaving the details of screen update to the scene graph. This idea of "change the scene graph and let screen update take care of itself" can greatly simplify application design. The Morphic[14] toolkit used a scene graph architecture to support its animation.

In a 2D pixel-based drawing model, scene graphs work slightly different from the graphics hardware in figure 23.7. A scene graph is a collection of two kinds of objects. First are the drawing objects which correspond to the drawing methods found in any Graphics object. There are also container objects that contain collections of both draw objects and other containers. Some containers are just a list of draw objects. Other containers can impose attributes such as geometric transformations, default colors or transparency. Some containers can have rectangular bounds that are checked at redraw time to see if the bounds are actually visible. If not then the drawing of that subtree can be avoided.

Drawing the scene graph

A simple scene graph drawing architecture uses a separate thread and a timer. Whenever the timer goes off (30 times/second), the tree is traversed and the scene redrawn. This mimics the behavior of the old graphics display hardware. If there are lots of changes going on continuously, as in an animation or video game, then this architecture works well. If changes are limited or infrequent, as in most applications that only change when an input event occurs, then a lot of compute power is wasted on unnecessary redrawing of the same image.

An alternative change propagation architecture can be used. When a change is made to some part of the scene graph, a notification is sent to the parent of the changed object and thus propagated up the tree. If a node has multiple parents (scene graphs are directed acyclic graphs) then all parents are notified of the change. Whenever a container node receives a change notice it marks itself as changed. If it is already marked then the change is not propagated up the tree. This makes certain that the root of each tree knows about changes but prevents extensive notification costs when many changes are made.

Periodically (30 times/second) the redraw process checks the root of each tree to see if changes have been made. If a change has been made then a redraw can be initiated. When a tree is redrawn, the change flags are cleared. If there are multiple parents a count must be maintained so that the change flags are not cleared until the last redraw has occurred. It is possible in some architectures for the redraw method to check what has changed and thus avoid redrawing screen fragments that have not been changed.

Scene graph/view relationship

Regardless of the way in which the scene graph gets redrawn, the view manages its presentation by modifying the scene graph rather than through a `paint()` method. This allows the view to ignore getting the screen updated. A scene graph also helps with essential geometry issues. Since each primitive to be drawn is represented as an object in the graph those objects can also have geometry methods such as `isHit()`, `distanceToPoint()` or any of the other geometry needs discussed in chapter 12. Because these methods are defined on scene graph objects, the application does not need to implement them. This architecture also guarantees that the essential geometry exactly corresponds to what is drawn.

To illustrate how this works consider the tree shown in figure 23.8 and its corresponding display on the right. When the mouse goes down over the line, the application's view simply asks the tree "what was hit?" The tree is traversed in

front to back order (opposite drawing order) and the geometry of each object in the tree is tested by the scene graph and its code. Finally it returns the line that is very close to the mouse point.

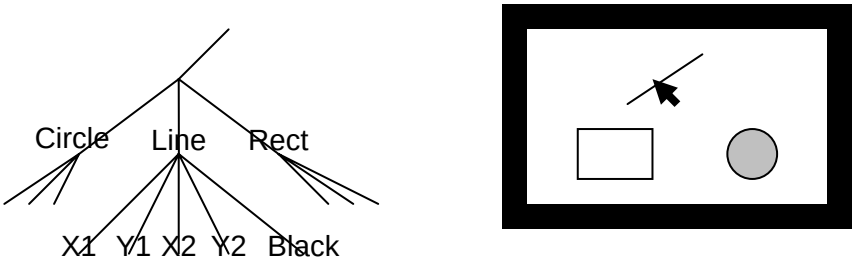

Figure 23.8 – Scene graph for a drawing program

The view responds by adding two new little "handle" objects to the tree (figure 23.9). These are subclasses of rectangle and have selection information. Adding the rectangles causes changes to be propagated and the view is redrawn. When the mouse goes down over one of the handle rectangles the view again asks "what was hit?" and the scene graph returns the rectangle subclass object. This object contains information about the control point. As further mouse-move events are received the view need only modify the position of the rectangle and the corresponding end point of the line. All other redraw and screen updating is handled automatically by the scene graph. When the line becomes unselected, the control rectangles are removed from the scene graph and automatically disappear from the display. Once the line is selected the view can change its color or any other attributed and the display will correspondingly change.

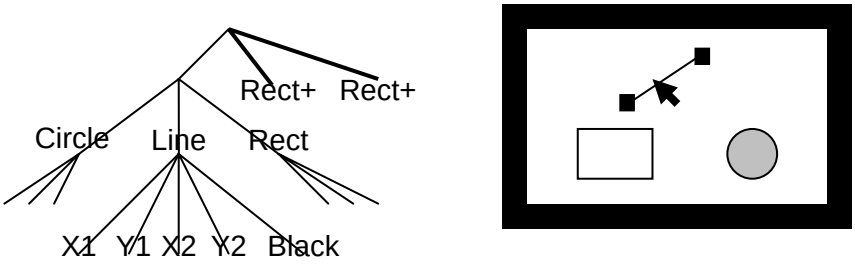

Figure 23.9 – Modified scene graph

One view/scene-graph architecture is for the view objects to maintain pointers to their scene graphic objects. When the view receives a model change

notification it follows this pointer and modifies the scene graph to reflect the new model values. However, in an object-oriented world one can create subclasses of the nodes in the scene graph. Thus a view can be implemented as a subclass of one of the container node classes. This integrates the view with the scene graph. The view now inherits all of the redraw, change propagation and essential geometry functionality. This blurring of the view and the scene graph starts to look very much like the widget trees that we worked with before. The difference is that a view object functions by modifying its children in the scene graph rather than by implementing a `redraw()` method.

Bederson et. al.[15] have defined two basic architectural styles for scene graph systems. They define *polylithic* architectures as ones where the composition of attributes and other information are defined as separate nodes and are composed at run-time while traversing the scene graph. They define *monolithic* toolkits as ones where each object is a subclass of some root class and each defines its own attributes. The widget sets in chapter 4 are of the monolithic variety. Many 3D graphics systems such as OpenGL and Java3D use the polylithic approach. The idea behind the polylithic approach is that there are lots of primitive facilities that can be composed into a scene graph and the composition of these defines the behavior.

Bederson and colleagues compared the Jazz[16] (polylithic) and Piccolo[15] (monolithic) tool kits on a variety of applications. These toolkits have almost identical functionality and differ primarily in their architecture. They found that their execution times were almost identical but that the monolithic Piccolo toolkit was much easier to teach and much easier to debug.

The Fresco[17] toolkit was a very early monolithic, scene graph architecture. Fresco integrated the concepts of variably intrinsic sized layout management with the flexibility of structured graphics. The delegation object-oriented language Self[18] demonstrated how highly flexible dynamic interactions could be built when a scene graph architecture is combined directly with a highly object-oriented language. More recently the UBit[19] toolkit is resurrecting the polylithic architecture with a more flexible interconnection of components.

The primary advantage of the scene graph architecture is that it simplifies redraw, makes essential geometry much simpler and also simplifies many basic interactions. For example drawing a rubberband selection rectangle is as simple as adding the rectangle to the graph and then updating one of its corner points on every mouse move. The redraw issues are all taken care of.

In addition the scene graph architecture provides the toolkit with a structured representation of the application. This is the kind of information that Attachments were trying to extract by adding groupStart/groupEnd to the Graphics object. In a scene graph this structure is readily exposed and does not need to be reconstructed from the Graphics object calls.

This scene graph information was exploited by Debugging Lenses[20]. A debugging lens behaves very much like a toolglass except that the information presented inside of the lens is targeted towards debugging user interfaces. A problem in user interface implementation is that when the display looks wrong it can be very difficult to figure out why. Debugging lenses can show the class of the objects being displayed as well as shrinking them away from each other to show the container relationships. Debugging lenses can display a variety of properties about the view's structure because that structure is directly available to the toolkit.

This object-oriented structure of the scene graph is exploited even further in the C3W system[21]. This system shares the goals of Facades[11] and WinCuts[12] in that it allows portions of the user interface to be extracted and copied elsewhere. The fact that a scene graph is a graph rather than a tree means that the new location can simply reference a portion of the scene graph from the old location. This generates appropriate updates whenever the scene graph changes. This is much less memory intensive than the off-screen image techniques and allows transformations to be performed more efficiently than by image manipulation. The big contribution of C3W, however, is that each node in the graph can have their attributes interconnected much like a spreadsheet. This allows end users to make arbitrary connections between application fragments that they see in the screen. Whenever an attribute changes, the scene graph notification mechanism informs the toolkit and if the changed attribute participates in some formula then all dependent scene graph fragments can be updated.

Summary

Applications must render all of the information that a user will use through their presentation layer. By capturing and modifying this presentation layer a whole new set of capabilities can be defined that work across all applications. One simple mechanism is to provide an alternative Graphics object to the redraw() method. This technique allows many toolglasses and attachments to be created. An alternative architecture is to capture all drawing in an off-screen image for each window and then allow many different ways for rendering those images. This provides for new windowing techniques as well as copy and paste of custom

user interfaces. Lastly we can represent our presentation as a scene graph rather than in a redraw() method. This simplifies the redraw task, greatly simplifies the computation of essential geometry and offers new information that can be exploited in new lens and customization techniques.

---

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

[2] Edwards, W. K., Hudson, S. E., Marinacci, J., Rodenstein, R., Rodriguez, T., and Smith, I. "Systematic Output Modification in a 2D User Interface Toolkit", *User Interface Software and Technology (UIST '97)*, ACM (1997), pp 151-158.

[3] Olsen, D. R., Hudson, S. E., Verratti, T., Heiner, J. M., and Phelps, M., "Implementing Interface Attachments Based on Surface Representations", *Human Factors in Computing Systems (CHI '99)*, ACM (1999), pp 191-198.

[4] Gutwin, C., and Greenberg, S., "Workspace Awareness Support with Radar Views," *CHI '96 Conference Companion*, ACM (1996), pp 210-211.

[5] Olsen, D. R., Boyarski, D., Verratti, T., Phelps, M., Moffett, J. L., and Lo, E. L., "Generalized Pointing: Enabling Multiagent Interaction", *Human Factors in Computing Systems (CHI '98)*, ACM (1998), pp 526-533.

[6] Mac OS X v10.2 "Technologies: Quartz Extreme and Quartz 2D." *Apple Technology Brief*, October (2002).

[7] Pike R., "The Blit: A Multiplexed Graphics Terminal", *AT&T Bell Labs Technical Journal*, 63(8), part 2.,(1983), pp 1607-1631.

[8] Robertson, G., Horvitz, E., Czerwinski, M., Baudisch, P., Hutchings, D., Meyers, B., Robbins, D., and Smith, G., "Scalable Fabric: Flexible Task Management", *Advanced Visual Interfaces (AVI '04)*, ACM, (2004), pp 85-89.

[9] Richardson, T., Stafford-Fraser, Q., Wood, K. R., Hopper, A., "Virtual Network Computing", *IEEE Internet Computing*, Vol. 2 , No. 1, 1998.

[10] Chapuis, O., and Roussel, N., "Metisse is not a 3D Desktop!" *User Interface Software and Technology (UIST '05)*, ACM (2005), pp. 13-22.

[11] Stuerzlinger, W., Chapuis, O., Phillips, D., and Roussel, N., "User Interface Facades: Towards Fully Adaptable User Interfaces," *User Interface Software and Technology (UIST '06),* ACM (2006), pp. 309-318.

[12] Tan, D. S., Meyers, B., and Czerwinski, M., "WinCuts: Manipulating Arbitrary Window Regions for More Effective Use of Screen Space," *Late Breaking Results (CHI '04),* ACM (2004), pp. 1525-1528.

[13] Bezerianos, A., Dragicevic, P., Balakrishnan, R., "Mnemonic Rendering: An Image-Based Approach for Exposing Hidden Changes in Dynamic Displays" *User Interface Software and Technology (UIST '06),* ACM (2006), pp. 159-168.

[14] Maloney, J. H., and Smith, R. B. "Directness and Liveness in the Morphic User Interface Construction Environment" *User Interface Software and Technology (UIST '95),* ACM (1995), pp. 21-28.

[15] Bederson, B. B., Grosjean, J., and Meyer, J., "Toolkit Design for Interactive Structured Graphics," *IEEE Transactions on Software Engineering 30(8),* IEEE (Aug 2004), pp 1-12.

[16] Bederson, B. B., Meyer, J., and Good, L., "Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java", *User Interface Software and Technology (UIST '00),* ACM (2000), pp. 171-180.

[17] Tang, S. H., and Linton, M. A., "Blending Structured Graphics and Layout", *User Interface Software and Technology (UIST '94),* ACM (1994), pp. 167-174.

[18] Smith, R. B., Maloney, J., and Ungar, D., "The Self-4.0 User Interface: Manifesting a System-wide Vision of Concreteness, Uniformity and Flexibility", *Object-Oriented Programming Systems, Languages and Applications (OOPSLA '95),* ACM (1995), pp. 47-60.

[19] Lecolinet, E. "A Molecular Architecture for Creating Advanced GUIs", *User Interface Software and Technology (UIST '03),* ACM (2003), pp. 135-144.

[20] Hudson, S. E., Rodensein, R., and Smith, I. "Debugging Lenses: A New Class of Transparent Tools for User Interface Debugging", *User Interface Software and Technology (UIST '97),* ACM (1997), pp. 179-187.

[21] Fujima, J., Lunzer, A., Hornbaek, K., and Tanaka, Y., "Clip, Connect, Clone: Combining Application Elements to Build Custom Interfaces for Information Access", *User Interface Software and Technology (UIST '04),* ACM (2004), pp. 175-184.