Blending Structured Graphics and Layout

Steven H. Tang
Fujitsu
tang@hadar.fujitsu.com

Mark A. Linton
Silicon Graphics
linton@sgi.com

ABSTRACT

Conventional windowing environments provide separate classes of objects for user interface components, or "widgets," and graphical objects. Widgets negotiate layout and can be resized as rectangles, while graphics may be shared (not just strictly hierarchical) transformed, transparent, and overlaid. This presents a major obstacle to applications like user interface builders and compound document editors where the manipulated objects need to behave both like graphics and widgets.

Fresco[1] blends graphics and widgets into a single class of objects.We have an implementation of Fresco and an editor called Fdraw that allows graphical objects to be composed like widgets and widgets to be transformed and shared like graphics. Performance measurements of Fdraw show that sharing reduces memory usage without slowing down redisplay.

KEYWORDS: User interface toolkit, object-oriented graphics, structured graphics.

1 INTRODUCTION

As user interfaces evolve toward more realistic representations of their real-world counterparts, the distinction between a user interface component or "widget" and a graphical object or collection of graphical objects becomes more of a barrier than a useful separation. The keypad buttons of a telephone, the jog-shuttle dial of a VCR control, and graphical icons in geological or archaeological maps can all be thought of as traditional widgets. They can be composed in layouts and they interact with the user. However, unlike traditional widgets, they also may be transformed and their appearance need not be rectangular and opaque. Their graphical nature becomes more compelling when they are projected into a 3D world where users interact with them by navigation.

Similarly, one could use layout in arranging a collection of graphical objects. For example, an interactive room design object could automatically place the correct number of chairs around a table. If the user chose a smaller chair size or a larger table, more chairs would appear.

When manipulating graphics, changing size means applying a scaling transformation. When manipulating a layout, the same operation may mean rearranging the layout. If we blend graphics and layout, then we can apply transformation and layout to all objects in a unified fashion. In the room design example, scaling the entire room would change the physical size of the table and chairs, but would not change the arrangement. Resizing just the table, on the other hand, could effect the number and placement of the chairs.

Fresco defines an architecture where all objects that draw or affect layout are derived from a common class, called Glyph, based on the InterViews[3] class of the same name. A glyph is a lightweight, sharable object. Unlike InterViews, the Fresco default implementation of the Glyph supports redisplay management by keeping a list of all its parents. A glyph's geometric extensions therefore can be obtained dynamically by computing the accumulated transform and geometric allocation along each glyph traversal that leads to it.

A special subclass of Glyph, called Viewer, handles input events. Viewers are based on the View class in the Andrew ToolKit (ATK)[4]. Unlike ATK views, a viewer may appear anywhere a glyph may appear. In particular, viewers may be transformed like any glyph.

To experiment with Fresco's integration of graphics and layout, we have implemented an application called Fdraw. Like a drawing editor, Fdraw can create, transform, and group graphical objects. It allows sharing of objects such that manipulating an object immediately updates all the places where the object appears on the screen. Because viewers are glyphs, Fdraw can manipulate objects such as buttons in the same manner as graphical objects. Finally, Fdraw allows grouping of objects to control the layout of the objects in a group. One can get the effect of a traditional drawing editor by having a group with no layout. Fdraw also supports groups that perform horizontal or vertical tiling.

2 RELATED WORK

Commercial window systems such as Microsoft Windows, MacOS, X, and NextStep all have programming interfaces for developing user interface components such as buttons, menus, sliders, and dialog boxes. None of these systems, however, includes support for structured graphics with transformations. Many research systems have a similar focus on widgets and layout.

Structured graphics toolkits such as Inventor[5] support transformations and sharing, but not layout. Because the issues of sharing and redisplay in the presence of transformations and partially-transparent objects dominate the problems of layout, the Fresco model is much closer to Inventor than conventional UI toolkits.

SUIT[6] is a toolkit that focuses on the graphical and interactive aspects of UI objects, but does not directly support layout. ET++[7] and GARNET[8] are both similar to Fresco in that the base class of objects can be used for both graphics and user interface components. However, neither system supports physical sharing of objects. The semantics of sharing can be provided using prototypes in GARNET or multiple views in ET++, but either approach will use much more memory than physical sharing if the object is large. Moreover, neither system addresses graphics and layout blending issues, like transformation with allocated geometry.

The Fresco architecture has evolved from previous work on InterViews and Unidraw[9]. The InterViews builder, Ibuild[10], used structured graphics to simulate user interface objects because InterViews widgets originally could not be transformed. Fresco unifies the InterViews glyph and Unidraw structured graphics classes, and Fdraw demonstrates how a builder can use Fresco objects directly to construct a user interface.

3 FDRAW

Fdraw supports creating, transforming, sharing, and grouping graphical objects. Unlike a conventional drawing editor, it also supports some builder capabilities like the creation of widgets and layouts, and the testing of their resizing and interactive behavior. Fdraw has two operating modes: edit and run. In edit mode, all objects behave like graphics. In run mode, the tool palette disappears and the interactive behavior of widgets can be tested directly.

One intriguing feature of Fdraw is that it is able to instantiate itself in edit mode, then in run mode the newly created Fdraw can instantiate more Fdraws within it. This demonstrates that Fdraw potentially can be self-modifying. In the simpliest case, this feature can be used to adjust and test the layout of Fdraw itself at run time. In cases where object linking and embedding is available, Fdraw can use this feature to generate variants of itself that embed other applications, then incorporate the new instance for its own definition.

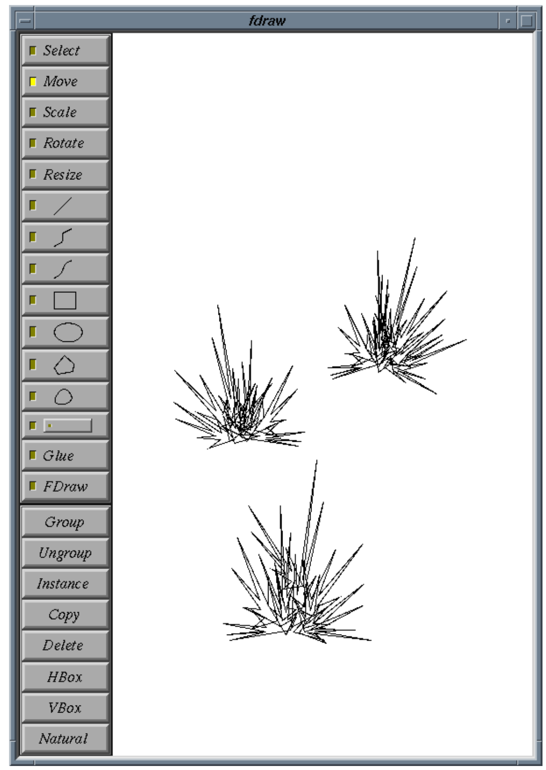


Figure 1 : Creating drawings with instances

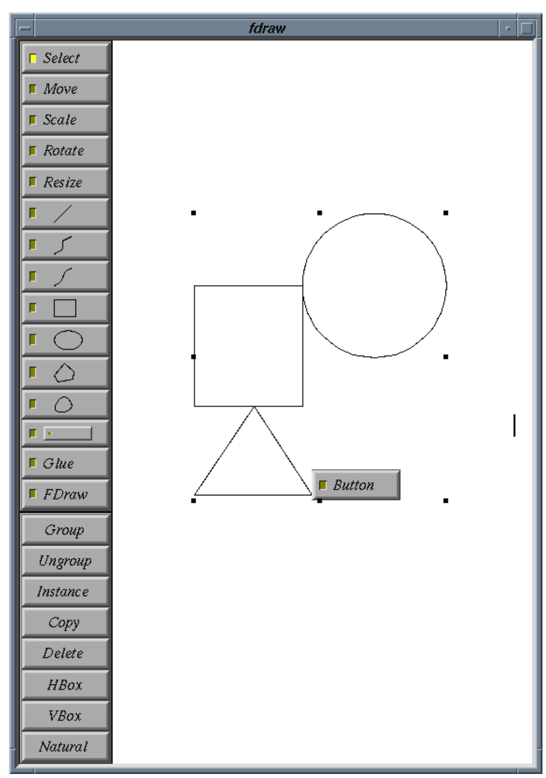


Figure 2 : Embedding graphics in layouts

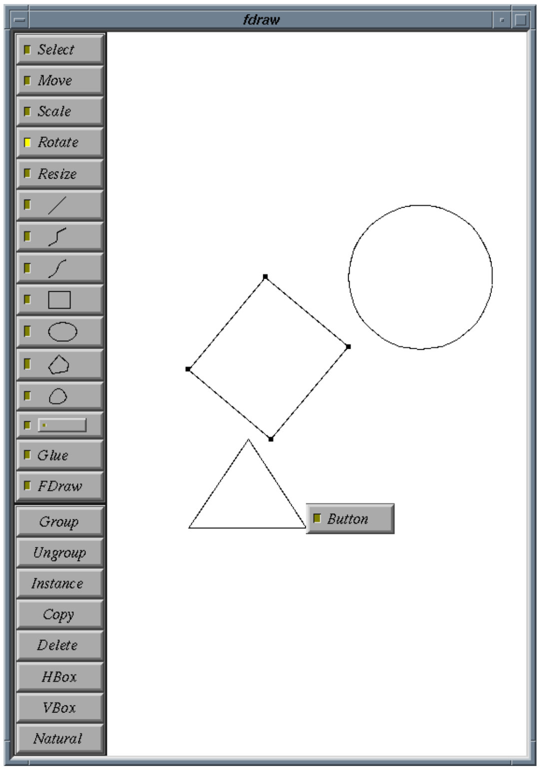


Figure 3 : Manipulating graphics in layouts

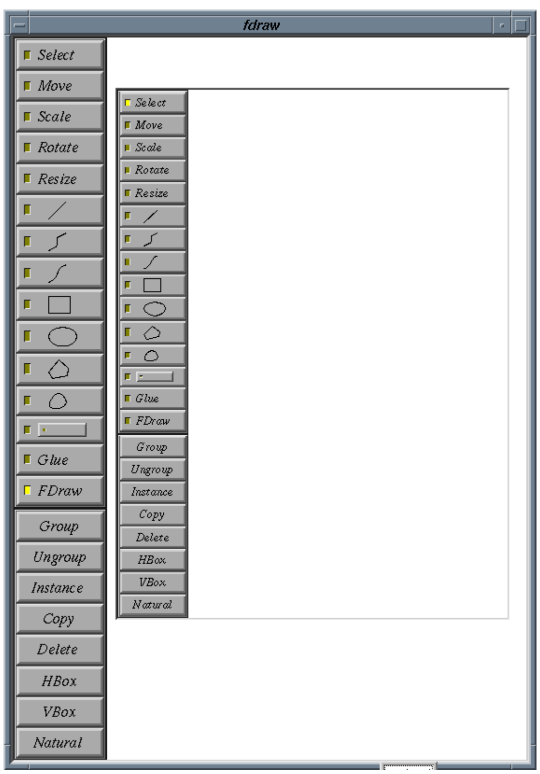


Figure 4 : Instantiating Fdraw in Fdraw

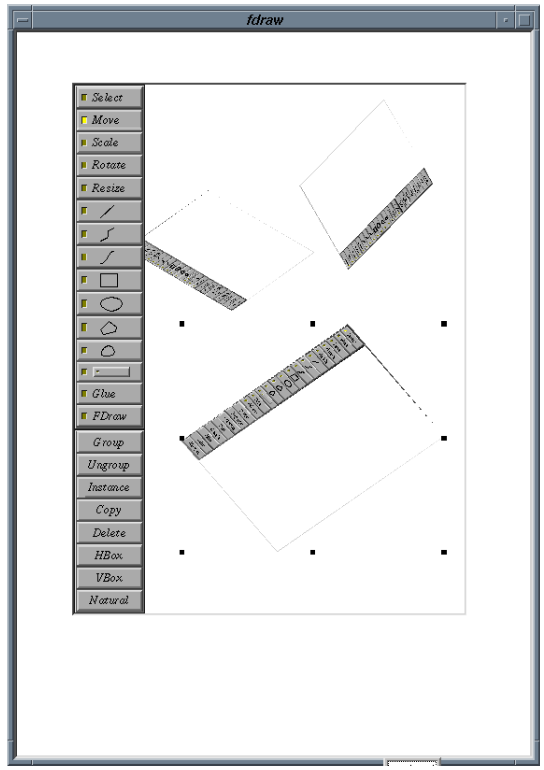


Figure 5 : Running Fdraw in Fdraw

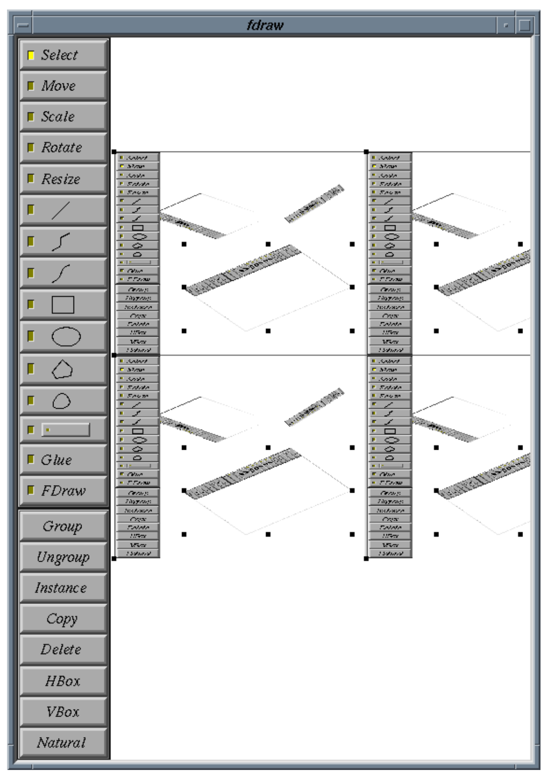


Figure 6 : Composing Fdraw in Fdraw

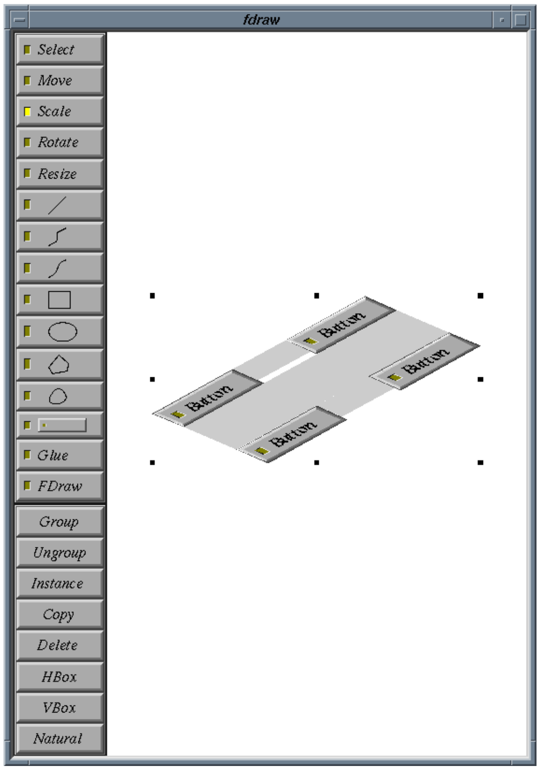


Figure 7 : Transformed composition

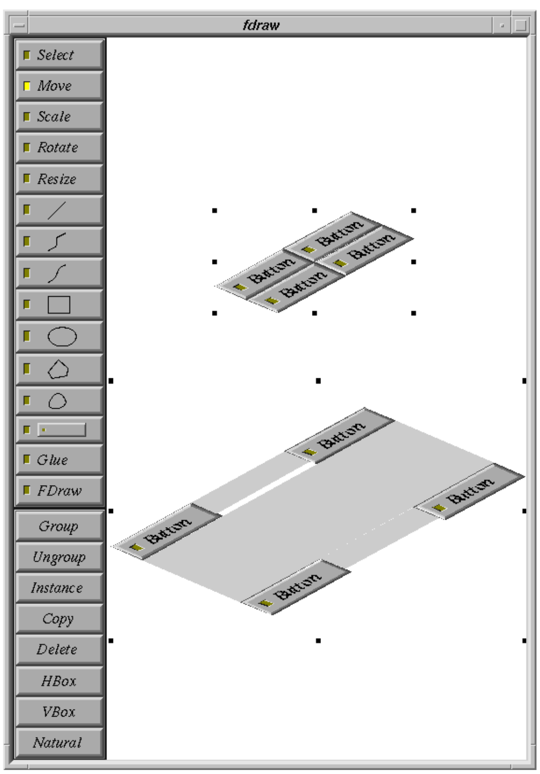


Figure 8 : Resizing transformed compositions

Figure 1 shows the appearance of an Fdraw in edit mode. The artistic drawings are made of instances of one multi-line object only. The composed drawings can be further instanced to create others. Instances are substantially cheaper in size than even a leaf graphic. They consist of a transformation matrix and a child pointer only. Using them implies substantial savings in memory with little loss in performance.

In Figure 1, a rectangle and circle are composed in a horizontal box, a triangle and a button are composed in another horizontal box, and the two boxes are composed in a vertical box. A box tiles its children along its major axis and aligns its children along its minor axis. Because of the way they were created, the rectangle aligns at its top left edge while the circle aligns at the center. This explains the vertical offset between the two objects. Similar offset is also observed in the composition between the triangle and the button.

Figure 1 shows the effect of manipulating a graphic inside a composition. As the rectangle rotates, the position of its top left edge shifts. As a result, the upper box translates the rectangle down to maintain the vertical alignment. This causes the vertical span of the upper box to grow and consequently, pushes the lower box down.

Since Fdraw itself can be treated as a widget, we can actually instantiate Fdraw within Fdraw itself as shown in Figure 1. The newly created Fdraw can participate in any manipulation like a normal graphic. Figure 1 shows an Fdraw in run mode. The embedded Fdraw instance is now executing like the original one. Furthermore, it can create more Fdraws within itself. Figure 1 shows four instances of Fdraw composed in horizontal and vertical boxes. If we select to run these Fdraw instances, all four Fdraws will manipulate simultaneously. Here the importance of resizing in the presence of transform is highlighted: A user may wish to be more aware of surrounding objects by zooming out, thus causing the Fdraw instances to be displayed at a different scale factor. A user may also wish to adjust the aspect ratio of an instantiated Fdraw to view more of its manipulated objects.

Figure 1 shows a transformed composition consisting of glue and buttons. Figure 1 shows two instances of the original composition being resized. Both the glue and the buttons maintain their resizablity after being transformed.

4 FDRAW IMPLEMENTATION

Fdraw relies on the Fresco library for the basic implementation of glyphs that supports object traversal and update. Fdraw also uses the Fresco Figure class to represent graphical objects, as well as the Fresco layout objects such as boxes and glue. In addition, Fdraw defines a Manipulator class to coordinate the direct manipulation of glyphs.

4.1 Glyph

The fundamental object class in Fresco is called Glyph. It defines operations for geometry management, drawing, picking and directed-acyclic structure (DAG) manipulation. Subclasses define how these operations are performed. For example, a MonoGlyph is a glyph that delegates most of its operations to its body glyph. A PolyFigure is a composite glyph that defines no layout policy, the locations of its children are completely determined by their transforms. To support blending, layouts position their children by modifying the cumulative transform during a traversal much like composite figures do. Because of this strategy, a figure does not have to distinguish whether its parent is a layout or another figure and the traverse behavior is uniform.

4.2 Figure

A figure represents a graphical object that defines a transform. During a traversal, the transform is accumulated as part of the traversal state for drawing and picking. The transform is also used for adjusting a glyph's geometry requisition to reflect its true appearance. A figure's effective requisition is always rigid. Hence, it always assumes that it gets its desired geometry. As an optimization, figures ignore their allocations during traversals. Composite figures simply pass a nil allocation to their children. This strategy has two benefits:

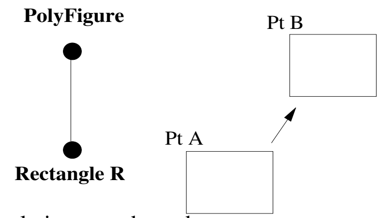
1. During a traversal, composite figures do not have to request their children's desired geometry and compute their corresponding allocations.

2. Their children may use this as a hint for their own geometry management. A button embedded inside a polyfigure for example, may self-allocate itself.

4.3 Manipulator

The use of a manipulator has been discussed in various object-oriented systems. A manipulator is responsible for providing visual feedback during a manipulation sequence. Typical visual feedback is achieved by redrawing a rubberband using the xor technique. This approach is indirect and the feedback is potentially unfaithful. For example, when one rotates a rotated object, the visual feedback corresponds to the rotation of the graphic's bounding box, which could be quite unrelated to the graphic's actual orientation. Moreover, the effect of the manipulation is applied to the selected graphics only at the end. A user potentially has to go through multiple trials before the desired effect is achieved.

Capitalizing on the blending of graphics and widgets, an Fdraw manipulator is a glyph that represents an instance of a manipulable glyph. During manipulation, a manipulator generally modifies its transform and manages partial update in response to incoming events. Since a manipulator belongs to the glyph dag itself, the visual feedback is achieved by directly participating in the redraw traversal of the dag. This approach not only provides more tightly-coupled interaction between the user and the user interface, it also gets rid of the need of a third party object that simulates the reaction of the target glyph during manipulation.



Manipulation pseudo-code:

```
R->need_redraw();
Transform* t = R->transformation();
t->translate(ptA to ptB);
R->need_redraw();
```

Figure 9 : Simple graphics editing

4.4 Editing Examples

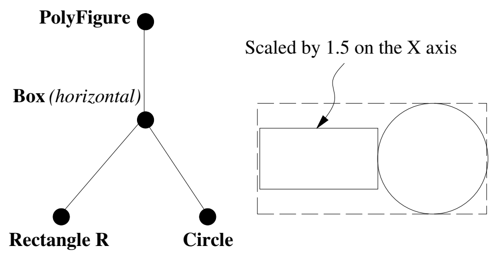
Figure 9 shows an example where a rectangle is translated from pt A to pt B. A partial instance hierarchy and a segment of the pseudo-code are provided to describe the usage of the Fresco protocol.

For partial update, the rectangle's geometric extensions at pt A and pt B need to be damaged. The pseudo-code achieves this by calling need_redraw twice: one before and one after the transformation. However, these two need_redraw calls cause two up traversals to the root glyph to compute the same cumulative transform. We can optimize this code by calling allocations once on R to obtain the cumulative transform from the root to its parent, then compute the two extensions using the old and new transforms of R. This way we only cause one up traversal.

When rectangle R is embedded in a horizontal box, and the box is itself embedded in another graphic, the only change to the above pseudo-code is the addition of a need_resize call on R as shown in Figure 9. As before, in order to update the screen consistently, rectangle R is damaged before and after the transformation. The need_resize call causes an up traversal to notify any layout object that its natural geometry has changed. In this case, the horizontal box will damage its old and new extensions. This means if a layout object is known to exists as the manipulated object's parent, only one need_resize call is needed to guarantee consistent screen update. Existence of the layout parent can also be computed by checking the geometric allocation passed to R because layouts pass valid(non-nil) allocations to their children.

5 Transforming layout

A glyph's requisition contains information about the desired geometry and flexibility along the X, Y and Z axes. When a glyph is transformed, the effective requisition is computed by transforming the original minimum, natural and maximum bounding boxes individually to form new ones. Combining these bounding boxes form the transformed requisition. Figure 9 shows the computation of a requisition that is rotated 45 degrees.



Manipulation pseudo-code:

```
R->need_redraw();
Transform* t = R->transformation();
t->scale(1.5 on the X axis);
R->need_redraw();
R->need_resize();
```

Optimized version:

```
Transform* t = R->transformation();
t->scale(1.5 on the X axis);
R->need_resize();
```

Figure 10 : Graphics editing with layouts

Perhaps more interesting is how a transforming monoglyph computes its child allocation. Figure 9 shows a button embedded inside such a monoglyph called a Transformer. The dotted boxes describe the transformed requisition. The solid box describes the actual allocation given to the transformer. If the entire allocation is passed to the button, the output will be incorrect since the transformer's transform is accumulated for drawing the button. For transformations that do not involve rotation, the allocation can simply be inverse transformed by the transformer. For those who do, inverse transforming the allocation can potentially introduce an allocation that is larger than the original. Referring to Figure 9, if the transformer

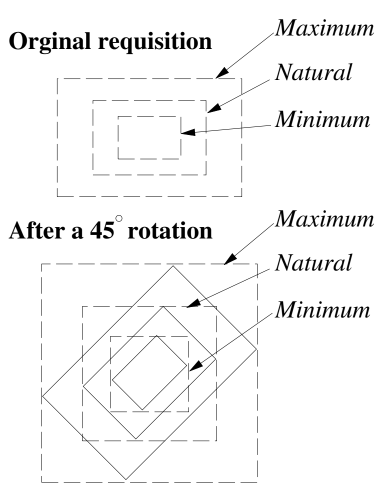


Figure 11 : Requisition transformation

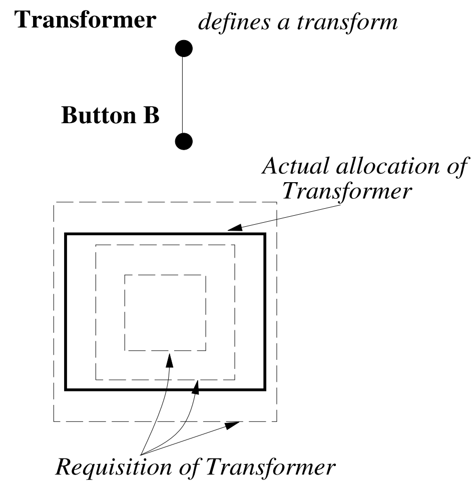


Figure 12 : Allocation transformation

rotates the button by 45 degrees, rotating its allocation by -45 degrees will produce a child allocation much larger than the natural allocation of the button.

The strategy adopted by Fresco for allocation computation is shown in Figure 9. We first break down the transformed requisition into their corresponding vectors on each axis. A total of 9 flexible vectors for the case of 3D, or 4 flexible vectors for the case of 2D are obtained. We then tile these vectors on each axis to compute the effective requisition. The allocation is broken down into its component axis vectors also. These vectors are then subdivided for the requisition vectors using a strategy similar to the one used by boxes. For example, on the X axis, there is an x requisition vector that corresponds to the transformed width and height of the button's requisition. Then we use the box tiling algorithm to compute the allocation for each of these two requisitions. After repeating the same procedure for the Y axis, we now obtain a pair of allocation vectors for each axis. These two pairs of vectors then have to be adjusted to maintain their original aspect ratio. Joining these vectors back form the effective child allocation. When the transformation involves no rotation, it can be shown that this strategy produces the same result as inverse transforming the allocation.

6 Performance

Fresco and Fdraw have a set of features that make it difficult to compare them with other systems. Nonetheless, we wanted to measure the performance effects of some of the design choices we made, particularly in the use of sharing. For our comparison, we used a special version of Idraw, the drawing editor built with Unidraw, by removing the functionality not yet implemented in Fdraw.

Table 1 shows the results of our measurements. We determined the initial memory sizes (code and data) of the two programs running on a Sparcstation 10 with 32Mb of memory and Solaris

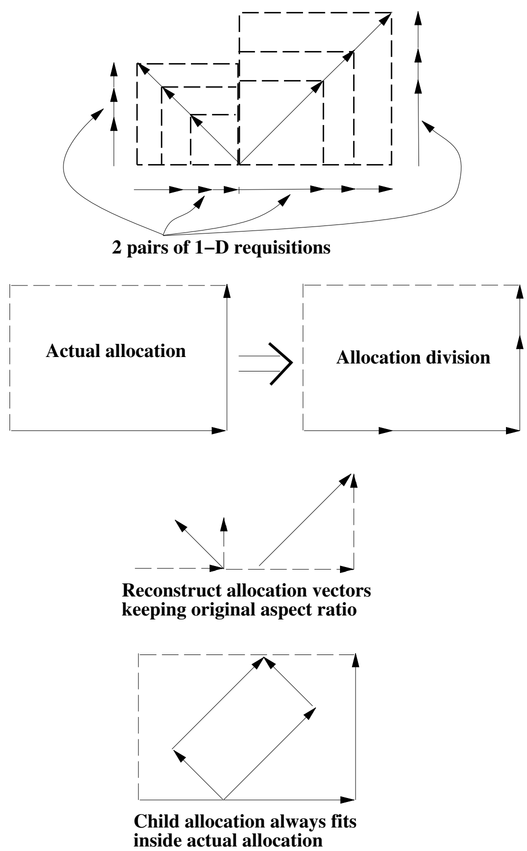


Figure 13 : Allocation computation

2.3. We then measured the memory usage and update time after creating 1024rectangles of identical size in the following structures:

Rectangle list – a flat list of 1024 different rectangles

Instances – a list of 1024 instances pointing to one distinct rectangle

Rect tree – a balanced binary tree containing 1024 distinct rectangles

Instance tree – a balanced binary tree containing instances pointing to one distinct rectangle

Fdraw's initial image size is 40% smaller than Idraw's, primarily because Idraw uses libraries with more built-in widgets. To create 1024 rectangles in a flat hierarchy, Fdraw uses 196K of memory, compared to Idraw's 190K. Fdraw's usage could be significantly smaller if manipulators were created on-demand rather than created one-per-object during startup. Graphical objects in Fdraw are also slightly more expensive because they use 3-dimensional data structures.

When the rectangles are shared in Fdraw, the memory usage drops to 110K, a 43.9% reduction. When the rectangles form a binary tree, the memory usage is 336K in Fdraw and 500K in Idraw. Fdraw is smaller because its composite objects are

	Fdraw	Idraw
Initial size	819K	1365K
Startup	0.911 sec	0.966 sec
Rectangle list	196K	190K
Full update	0.473 sec	0.375 sec
Instances	110K	190K
Full update	0.467 sec	NA
Rect tree	336K	500K
Full update	0.573 sec	0.543 sec
Instance tree	14K	500K
Full update	0.560 sec	NA

Table 1: Fdraw vs. Idraw

smaller. When this binary tree is shared in Fdraw, the instance hierarchy formed a stack of 10 diamonds, each consisting of 3 manipulators. The storage needed in this case is only 14K.

In terms of run time performance, sharing does not have a significant impact since it does not reduce the number of traversed nodes in a glyph dag. From the results generated by Fdraw however, sharing does seem to improve performance slightly. This is probably due to an increase in the cache hit ratio. Idraw has longer load time than Fdraw. This is probably because of its larger image size. On the other hand, since idraw does not have manipulators, there are few nodes to visit during a draw traversal. Consequently, its update time is generally better than Fdraw's. Creating manipulators on-demand should eliminate this difference.

7 Conclusions

The Fresco architecture allows arbitrary blending of graphics and layout, meaning graphics can be composed in layouts and widgets can be transformed like graphics. Fresco also supports sharing, which can significantly reduce memory usage while having little or no effect on update performance.

Fdraw demonstrates the power and efficiency of blending graphics and layout by combining the features of a drawing editor and interface builder into a single application. In Fdraw, a user can both scale and resize objects through transformation and layout, respectively. In particular, Fdraw itself can be instantiated and manipulated in Fdraw, enabling the possibility of a self-modifying editor.

Integrating graphics and layout results in a simpler overall model, as programmers and users alike do not need to learn about two different kinds of objects. In addition to a builder, this approach could be applied to what is traditionally called a "window manager." Using Fresco, one could implement a manager that allowed the user to place and manipulate realistic-looking application objects in a 3D scene.

8 References

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

[2] M. Linton, S. Tang, and S. Churchill. Redisplay in Fresco. *Proceedings of the Eighth X Technical Conference*, Boston, Massachusetts, January 1994.

[3] P. Calder and M. Linton. Glyphs: Flyweight Objects for User Interfaces. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Snowbird, Utah, November 1990.

[4] A. Palay et al. The Andrew Toolkit: An overview. *Proceedings of the 1988 Winter USENIX Technical Conference*, Dallas, Texas, February 1988, pp. 9-21.

[5] P. Strauss and R. Carey. An Object-Oriented 3D Graphics Toolkit. *Computer Graphics*, Vol. 26, No. 2, July 1992, PP. 341-349.

[6] R. Pausch et al. SUIT: The Pascal of User Interface Toolkits. *Proceedings of the ACM Symposium on User Interface Software and Technology*, Hilton Head, SC, November 1991, pp. 117-125.

[7] A. Weinand, et al. “ET++-An Object-Oriented Application Framework in C++”, *OOPSLA ‘88*, San Diego, CA, September 1988, pp. 46-57.

[8] B. Myers et al. Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces. *Computer*, November 1990, vol. 23, no. 11, pp. 71-85.

[9] J. Vlissides and M. Linton. Unidraw: A Framework for Building Domain-Specific Graphical Editors. *Transactions of Information Systems*. Vol. 8, No. 3, July 1990, pp. 237-268.

[10] J. Vlissides and S. Tang. Ibuild: A Unidraw-Based User Interface Builder. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Hilton Head, South Carolina, November 1991.