Redisplay in Fresco

Mark Linton, Steven Tang, and Steven Churchill

Abstract

Redisplay after a change to an object in a traditional structured graphics environment only affects the modified object and any objects that intersect it on the screen. For user interface components such as Xt widgets, redisplay may involve recomputing the layout of other user interface objects. Fresco unifies structured graphics and user interface components under a common base abstraction, which provides a simple, powerful model but complicates the redisplay problem. As part of the Fresco implementation, we have implemented a general redisplay algorithm that recomputes layout when necessary but minimizes the cost of performing transformations when updating a structured graphics hierarchy. Our implementation can also run redisplay as a separate thread, allowing independent application threads to trigger screen update.

Introduction

Fresco is a user interface system that has three distinguishing features with respect to Xt and many other systems. First, the Fresco specification uses a standard object model from the Object Management Group (OMG). Second, all graphics in Fresco may be specified in resolution-independent coordinates, meaning objects have the same size and appearance regardless of the dots-per-inch of a particular screen or printer. Third, and perhaps most important, Fresco combines the traditional areas of structured graphics and application embedding into a single unified model for user interface objects. This integration allows embedded applications to be appropriately transformed when they appear in graphical scenes.

The first two features, the object model and resolution-independence, are discussed in [2] and [3]. This paper describes one specific aspect of graphical embedding—how objects are redisplayed on the screen when a change occurs. The redisplay problem in Fresco is different from other systems

---

Mark Linton is a Principal Scientist at Silicon Graphics.
Steven Tang is a Ph.D. student at Stanford University.
Steven Churchill is a Member of Technical Staff at Fujitsu, Ltd.

because Fresco allows the mixture of structured graphics objects, which may be referenced by more than one aggregate object, and user interface objects, which may use a layout algorithm to assign the positions of their children. Structured graphics systems (e.g., PHIGS+[4], Inventor[5]) support the redisplay of a directed acyclic graph (dag) of objects, and user interface systems have supported the redisplay and resizing of individual widgets. However, Fresco is the first system to provide a single redisplay strategy for a dag that contains both kinds of objects.

Viewers and glyphs

The primary building block for user interfaces in Fresco is called a *viewer*, which handles both presentation and editing. Viewers are similar to widgets in Xt, though they also provide embedding-specific functionality such as saving their data or specifying the contents of a shared menubar when they are given input focus.

The viewer type is derived from a more primitive, presentation-only type called *glyph*. While viewers are organized in a strict hierarchy, glyphs may be shared. That is, two or more composite glyphs may contain references to the same glyph (one composite could also reference the same glyph twice). Sharing offers both semantic and performance advantages. From a semantic point of view, a single change can affect all the uses of a glyph. From a performance perspective, sharing can significantly reduce the memory cost associated with a frequently-used object.

Sharing implies that a glyph reference does not distinguish the glyph with respect to its parent, as it could appear more than once in the composite. A *glyph offset* uniquely denotes a child glyph within a composite—in other words, an offset refers to an edge in the glyph dag. An offset can only be created by its composite because the implementation of an offset must know the composite's representation. Operations such as insert, remove, and replace, are defined on the offset rather than the composite.

In the center of Figure 1 is an example drawing containing triangles and rectangles. On the left is a glyph dag that could represent the drawing using sharing and transformations. On the right is a dag that could represent the drawing using layout. Both dags contains a single rectangle and triangle instance, denoted by *R* and *T*, respectively. On the left, the group *G* consists of the rectangle and four instances of a triangle, each with a different transformation. On the right, the layout object L will compute the positions of the rectangle and triangles on-the-fly. The drawing contains two instances of the composite (either *G* or *L)*, each with a transform.

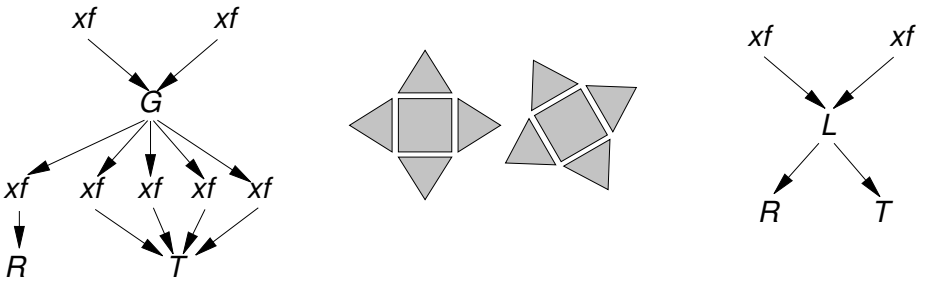

Fig. 1: Example drawing and possible associated dags

Traversals

A *glyph traversal* encapsulates the shared state associated with visiting every glyph in the dag and calling a particular operation on each of the leaves. For example, a draw traversal starts at the root of the dag and recursively visits every glyph, calling the Glyph::draw operation on each leaf.

A traversal maintains the current rendering state (e.g., accumulated transformation), as well as a list of the offsets that define the path from the root to the current glyph. Unlike Xt and most other systems, Fresco does not define a traversal specifically for layout. During a draw or pick traversal, a composite may assign each child an *allocation*, which is a 3-dimensional region in the parent's object coordinates. A composite determines the allocations of its children based on its own allocation and the desired sizes of its children. Allocations must be passed during a draw traversal because a glyph that is shared could have different allocations depending on where it is used.

An allocation represents formatting information, which does not strictly speaking define where a glyph appears. For example, parts of the letters of very slanted text could be drawn outside the natural formatting region for the text. The extension operation on glyphs can be used to compute the bounds of where the glyph actually draws.

Modifying a glyph

Glyphs (and therefore viewers) are not necessarily opaque and rectangular—they may have an arbitrary shape, including one with holes. This feature means that a modification to a glyph cannot simply update the screen directly because of the possibility that other glyphs could appear above or below the modified glyph. For example, a label on top of an image may need to be redrawn when the image is redrawn.

Instead of updating the screen directly, a glyph requests that the area or areas where it appears should be redisplayed. The system will perform a draw traversal starting from the root of the glyph dag, with clipping set to the region that should be redrawn. A glyph may check to see if its output might intersect with the clip region; if not then the glyph need not draw itself or its children.

In addition to allowing objects to share the same area of the screen, this deferred update mechanism allows output to be double-buffered to give the user a smoother, less flashy appearance. Double-buffering in combination with partial update (where the redrawn region is not the entire buffer) requires either copying pixels from the front to back buffer or rendering the region twice (once for each buffer). In the latter approach, the second rendering can be deferred until the next screen update and possibly avoided if the same region is redrawn.

InterViews[1] provides a deferred screen update mechanism that is flexible, but very hard for programmers to use correctly. Fresco solves this problem by providing both the low-level mechanisms for flexibility and higher-level interfaces for ease of programming. The primitive mechanism relies on retrieving the allocations for a glyph and damaging the corresponding screen area. The higher-level interface involves simply asking a glyph to redraw or resize itself. A special form of resize is when an object is transformed outside of any layout constraints, such as in a drawing. In this case, using the primitive mechanisms directly is more efficient.

Getting the allocations

The primary mechanism to support redisplay is the allocations operation, which computes the cumulative transformation and allocation information for each place where the glyph appears. This operation also returns a damage object for each allocation that is used to specify the region of the screen that needs to be redrawn.

Both the glyph and glyph offset interfaces define an allocations operation. In the Fresco implementation, the base glyph class maintains a list of parent offsets. The allocations operation simply calls the allocations operation on each of its parents. A glyph subclass that performs layout, such as a box that tiles its children, must additionally assign an allocation for the requesting child (which can be determined from the offset).

An alternative to the current implementation would be for a glyph to retain the allocations of its children rather than computing them for every update. Such an implementation is possible without making any changes to the current interface. We chose to compute the allocations because it takes less memory and does not require any overhead during a traversal to test if the stored allocations have changed.

Redraw

The glyph interface provides a need_redraw operation that should be called when a glyph is modified in a way that affects its appearance but not its size. For example, if we changed the color of the triangle in Figure 1, all the instances of the triangle in the drawing should be redrawn. The need_redraw operation is not appropriate if the glyph might have changed size, such as if we changed the scale of the triangle.

The default implementation of need_redraw calls the allocations operation to get the places where the glyph appears. For each allocation, need_redraw extends the damage region to include the area where the glyph will draw. By default, this area is the entire allocation. If the glyph defines a transformation matrix, then the damage region will be the transformed area.

The glyph interface also defines a need_redraw_region operation for updating part of a glyph. This operation is useful for images or other potentially large objects (in terms of screen space) that wish to update a portion of their appearance.

Resize

The glyph interface provides a need_resize operation that should be called when a glyph modification might cause the desired size of the glyph to change. Examples include adding or removing a glyph from an aggregate, changing the characters in a label, or modifying a transformation matrix.

The strategy that minimizes the area that is redrawn is to damage the old region, compute the new size, and then damage the new region. However, for situations involving layout, this strategy typically causes any composite that changes the layout of its children to be completely redrawn. The reason is the composite will likely change the layout of other children, which means they need to be redrawn as well. For example, the composite *L* in Figure 1 would probably react to a change in the size of the triangle by changing the layout of the rectangle, meaning the rectangle would also need to be redrawn.

We use a simpler strategy to implement need_resize in situations involving layout and handle the other case specially, which we describe in the next section. The default implementation of need_resize calls the notify operation of each parent offset, which in turn calls need_resize on the parent glyph. At some point in the dag a "fixed-sized" glyph may stop the propagation of need_resize calls and call need_redraw to draw all the glyphs within the fixed area. Scrollable viewers, such as for text or graphics, normally stop propagation because the desired size of the viewer is not a function of its components.

Transformations

The default need_resize algorithm is undesirable when no layout objects exist along the path between the modified glyph and the fixed size glyph. The simple algorithm will redraw all the glyphs within the fixed area when the only glyphs that need to be redrawn are those that intersect the region associated with the modified glyph.

For example, if the left dag in Figure 1 were inside a drawing viewer that allows the user to place, move, and transform objects interactively, changing one of the transforms for the triangle would cause the entire dag to be redrawn. Clearly, it is undesirable to redraw all the objects in the drawing when the user manipulates just one.

Damaging old and new areas

To avoid unnecessary drawing, we can rely on the primitive allocation and damage mechanisms directly. Before transforming an object, the viewer can retrieve the object's allocations and damage them. The viewer then can perform the transformation and compute the new damage from the previously-computed allocations and the new transformation matrix. The viewer is therefore able to damage only the old and new areas with a single pass to accumulate the allocation information. The code to change the triangle's position would look something like this:

```
t->allocations(al);   // get list of allocations
damage_all(t, al);    // damage t's old positions
move(t, x, y);        // change t's transform
damage_all(t, al);    // damage t's new positions
```

This algorithm is efficient, but does not correctly handle the case where the viewer contains both layout objects and directly-manipulable objects. This situation could arise with an interface builder that has a user interface similar to a drawing editor and allows the mixing of user interfaces objects and primitive graphical objects.

Recomputing layout when necessary

We can make the algorithm both efficient and correct with one addition. The viewer must insert a special glyph around each top-level layout object. After damaging both the old and new regions, the viewer calls need_resize on the modified glyph. As before, this operation propagates up the dag. The viewer must implement need_resize itself to do nothing so that if no layout object exists along the path then no extra processing occurs.

If a layout object does exist along the path from the modified glyph, then the specially-inserted glyph must compute the area to damage and continue the propagation by calling need_resize on its parents. The glyph computes the damage area by merging its old and new bounding boxes. The old

bounding box must be stored by the glyph. The new bounding box is computed by calling the extension operation, which may layout the glyph's descendants.

Example

For a viewer showing the drawing in Figure 1, the code to update the triangle's position would be as above with the addition of the call "t->need_resize()" to propagate the change. In the case of the dag on the left side of Figure 1, this call would propagate up to the viewer. The drawing viewer would implement need_resize as a nop, so the propagation would stop without incurring any further damage.

In the case of the dag on the right side of Figure 1, the viewer would have put a special glyph around the composite *L*. This glyph would implement need_resize by damaging its old bounding box (which it stored), computing its new bounding box (which involves computing the new layout), and then damaging the new bounding box.

Redisplay thread

Modifying a glyph causes damage that should lead to a draw traversal. An additional requirement for Fresco is supporting multithreaded applications, which complicates the decision of when to perform the draw traversal. In particular, we would like an application thread to be able to cause damage that leads to a draw traversal when no input events are pending, and we would like an expose event to cause a draw traversal while an input event is being processed (especially when the response to the event takes a long time).

Single-threaded systems typically perform screen update when no input events are pending, but this approach is not acceptable if Fresco is running in a multithreaded environment. Our approach is for the Fresco display object to create a separate thread for redisplay.

The redisplay thread communicates with the main (event-reading) thread through a condition variable that indicates whether there is any damage that needs to be repaired and a boolean variable indicating whether the event thread is currently blocked waiting for input. The threads also use a lock for the display to synchronize access to the display object's data.

When a display is notified of damage, the damage will be added to a repair list. If the event thread is waiting for an input event, then the redisplay thread will be immediately awakened. Otherwise, the event thread will awaken the redisplay thread when the event thread finishes processing the current input queue.

When the redisplay thread awakens, it repairs all the damage on the repair list. This processing should not be considered concurrent with other processing, as a draw traversal will lock the entire viewer tree to avoid possible inconsistencies.

The redisplay thread in the current implementation meets our requirements, but needs further experience with multithreaded applications before we can evaluate how well it works. In particular, the favoring of reading input events means that if a large number of input events are pending, one could prevent a redisplay from occurring for an arbitrary amount of time. Having a maximum time that the redisplay thread blocks when there is damage may be desirable.

Conclusions

The Fresco redisplay algorithm supports shared structured graphics, dynamically-changing layout, multithreading, and double-buffering. The programming interface defines both low-level primitives and higher-level operations that make redisplay easy to use. The primitives provide enough flexibility to tune the implementation for common cases.

Redisplay is the key factor that allows Fresco to unify graphical and user interface objects. Because we have one algorithm for updating objects, we are able to mix graphics, text, and widgets seamlessly on the screen.

References

[1] 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.

[2] M. Linton. Implementing Resolution Independence on top of the X Window System. *Proceedings of the Sixth X Technical Conference*, Boston, Massachusetts, January 1992, pp. 109-116.

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

[4] PHIGS+ Committee, Andries van Dam, Chair. PHIGS+ Functional Description, Revision 3.0. *Computer Graphics*, Vol. 22, No. 3, July 1988, pp. 125-218.

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