Fresco™ Specification

X Consortium Working Group Draft
Version 0.7
April 11, 1994

Fresco is a registered trademark of X Consortium, Inc.

Copyright © 1994 Silicon Graphics, Inc.

Copyright © 1994 Fujitsu, Ltd.

Permission to use, copy, modify, and distribute this documentation for any purpose and without fee is hereby granted, provided that (i) the above copyright notice and this permission notice appear in all copies, and (2) the names of Silicon Graphics and Fujitsu may not be used in any advertising or publicity relating to this documentation without prior written permission of Silicon Graphics and Fujitsu.

Silicon Graphics, Fujitsu, and the X Consortium make no representations about the suitability for any purpose of the information in this document. This documentation is provided as is without express or implied warranty.

This document describes a work-in-progress, not an X Consortium standard or draft standard. Both the document and the system described are subject to change.

Table of Contents

Guide to Fresco Specification 5

Overview 7

1. Introduction 7
2. Specifying interfaces 7
   - 2.1 IDL overview 8
   - 2.2 IDL and Fresco implementations 10
   - 2.3 Concurrency 10
3. Basic types 11
   - 3.1 Strings 11
   - 3.2 Memory management 11
   - 3.3 Callbacks 11
   - 3.4 Coordinates 12
   - 3.5 Notification 12
   - 3.6 Styles 12
4. Resolution independence 12
5. Graphical Embedding 13
   - 5.1 Spatial composition 13
   - 5.2 Logical composition 15
   - 5.3 Scripting 17
6. Summary 17

Standard Definitions 19

Action 19
Alignment 19
Axis 19
Brush 19
CharCode 20
CharString 20
Color 21
Coord 22
Cursor 22
Damage 22
Display 23
DrawingKit 25
Font 27
Fresco 28
Fresco_init 29
Fresco_open, Fresco_open_display 29

FrescoObject 30
Glyph 30
GlyphOffset 34
GlyphTraversal 35
Option 37
Painter 37
Pattern 40
PixelCoord 40
Raster 40
Region 42
Screen 43
Style 44
StyleValue 46
StyleVisitor 47
Tag 48
Transform 48
Vertex 49
Window 49

Guide to Fresco Specification

The Fresco specification contains the following two sections:

- an overview of Fresco concepts
- an alphabetized listing of all the Fresco types and functions

The specification requires C++ version 3.0 or later. In particular, we use struct declarations inside of classes that we refer to as *class-name::struct-name*. The same nesting is used for enumerations and typedefs.

The specification assumes but does not require that a target platform support multiple (preemptive) threads within a single address space. In particular, Fresco does not have a single main application loop analogous to XtAppMainLoop. Fresco provides an event reader thread for the default display; we assume one would use additional threads to support timers or other (non-X) input devices. In you want to use timers or alternate input on a platform without threads, then you should put the top-level Fresco object in an XtfViewer widget and use XtAppMainLoop.

Overview

1. Introduction

Fresco supports *graphical embedding*, which brings together the traditional areas of structured graphics and application embedding. Graphical embedding is a simple, uniform architectural framework where one can compose both graphical objects (such as the circles or rectangles found in a drawing editor) and application-level editors (such as the word processors or spreadsheets connected through Microsoft's OLE). As a result, embedded applications may appear within graphical scenes and will be appropriately transformed.

Fresco objects follow the CORBA standard object model developed by the Object Management Group (OMG), and, as a consequence, Fresco objects can be distributed across machines in a network or across protection domains on a single machine.

Fresco is a joint development project under the auspices of the X Consortium. In the first release, version 0.7, the emphasis has been to define a standard application programming interface for structured graphics. The Fresco 0.7 sample implementation also contains additional features in preparation for standardizing the graphical embedding API in the future.

This manual describes the objects and operations in Fresco 0.7. In this overview, we present the basic concepts of Fresco and the notation we use to specify the API.

Three major features distinguish Fresco from the APIs previously available in X and most other systems:

- Standard object model from the Object Management Group (OMG)
- Resolution independence
- Graphical embedding

We give an overview of the OMG object model in Section 2. The basic types pervasive throughout Fresco are presented in Section 3. Resolution independence means that graphical objects have the same size and appearance regardless of the dots-per-inch of a particular screen or printer. We describe the objects that support resolution independence in Section 4. Graphical embedding is described in Section 5.

2. Specifying interfaces

Programming languages typically provide a mechanism for specifying interfaces, but not in a manner that is appropriate for remote access to procedures or data. Distributed systems often provide a language for specifying how procedures can be called remotely, but without support for object references or polymorphism. The Object Management Group (OMG) has developed the Interface Definition Language (IDL) as a standard way to specify objects that may be accessed remotely. IDL is part of OMG's Common Object Request Broker Architecture (CORBA) that defines the mechanisms for transparent access to objects in a network.

Using IDL has several advantages over other alternatives:

- IDL is more abstract than object-oriented programming languages, yet concrete enough to be translated automatically into source code.
- IDL can be mapped to several languages. OMG has defined a standard binding to C, and efforts are in progress to standardize mappings for C++ and Smalltalk. As a result, an IDL specification is not tied to a particular programming language.
- IDL can support distribution; that is, the sender and receiver of an operation may be on different machines in a network.

2.1 IDL overview

An IDL object type is called an *interface*. The definition of an IDL interface simply states what operations can be performed on objects of that type; no implementation details are given. A subtype may be defined that inherits the interface of one or more other types. The IDL syntax for defining an interface is similar to a C++ class declaration. For example, consider the following possible definition of a rectangle object in IDL:

```
typedef float Coord;

interface Rectangle {
    Coord left();
    Coord top();
    Coord width();
    Coord height();
    void move(in Coord left, in Coord top);
    void resize(in Coord width, in Coord height);
};
```

We do not propose this interface as the only or best interface for a rectangle; we simply use this example for convenience. The interface defines six operations. The first two return the upper-left corner of the rectangle's position. The width and height operations return the dimensions of the rectangle. The move and resize operations change a rectangle's position or size.

In C++, one might be tempted to give a similar definition as follows:

```
class Rectangle {
public:
    Coord left() { return left_;}
    Coord top() {return top_;}
    Coord width() {return width_;}
    Coord height() {return height_;}
    void move(Coord left, Coord top);
    void resize(Coord width, Coord height);
private:
    Coord left_, top_, width_, height_;
};
```

Here we have written a partial implementation of the rectangle class as well as the interface, indicating both a representation and the implementation of several operations. Such a mixture of interface and implementation can cause confusion for programmers, as well as eliminate the possibility of certain alternate implementations. In particular, one could not access a rectangle remotely using this class.

A more flexible C++ definition of this rectangle class is as follows:

```
class Rectangle {
protected:
    Rectangle();
    virtual ~Rectangle();
public:
    virtual Coord left() = 0;
    virtual Coord top() = 0;
    virtual Coord width() = 0;
    virtual Coord height() = 0;
    virtual void move(Coord left, Coord top) = 0;
    virtual void resize(Coord width, Coord height) = 0;
private:
    void operator =(const Rectangle&);
};
```

In this version of the Rectangle class all the operations are "pure virtual," meaning no implementation is defined for them. One must therefore use a subclass of Rectangle to provide an implementation. One

subclass could provide an implementation for local rectangles, and another could provide an implementation for the remote case.

This approach of using a pure virtual class is compatible with an IDL definition. In particular, a translator could generate the C++ code automatically from IDL code.

2.1.1 Object references

IDL uses the concept of an object reference rather than a pointer, as a reference can denote an object in a foreign address space. IDL only supports passing or returning an object by reference, so any uses of an interface type in IDL are implicitly object references. For example, if an interface contained the operation

```
void clip(in Rectangle cliprect);
```

then a call to the clip operation would pass a reference to a rectangle object. The parameter passing mode may be “in,” “out,”, or “inout,” indicating that the value should be copied to the receiver, copied from the receiver, or copied both before and after, respectively.

In a C++ implementation, an object reference is logically a pointer to the corresponding pure virtual class. Several C++ types are defined for object references, to reflect the possible uses. In the example above, the operation would be declared as

```
void clip(Rectangle_in cliprect);
```

2.1.2 Invoking an operation

A call to an operation defined in IDL is just like a C++ virtual function call, with an object reference denoting the target on the left-hand side of the arrow (“->”). If the object is in the same address space as the caller then the call is simply an indirect function call. If the object is in a remote address space then the call is to a special implementation called a *proxy*, which uses system support for distributed objects to marshal parameters, locate the object, send a message across the network, and unmarshal the reply. On the receiving side, a special function called a *skeleton* unmarshals the parameters, calls the operation on the implementation object, and marshals the reply. Specific object types also may provide proxy and skeleton implementations that are tuned for a particular protocol.

For library objects, the cost of using IDL is equivalent in C++ to using virtual functions for all calls. Experience with C++ toolkits such as InterViews has been that these conventions are good to follow anyway. Virtual functions allow extensibility, and keeping data out of the public interface avoids recompilation of code that uses a class when the class representation changes.

2.1.3 Creating Objects

The possibility of remote objects raises the question of where objects should be located when first created. Furthermore, creating an object reference that can be passed to a remote operation may require more mechanism than calling a C++ constructor.

The basic approach to creating an object in a distributed environment is to perform an operation on another object that returns the new object. This “creator” object is sometimes called an “object factory,” though in Fresco we prefer to use the term “kit” for those objects whose primary function is to create other objects.

Kits offer advantages over C++ constructors even when creating objects in a single address space. A kit introduces a layer of abstraction that hides the details of object creation, particularly the existence of classes that add no operations but exist solely for the purposes of implementation. Additionally, modeling factories as kits provides a means for encapsulating contextual information that must be passed when creating the object, but is more convenient to set early in an application rather than every time a

new object is created. Relying on information in a kit rather than global data is helpful in the presence of both distribution and concurrency.

2.1.4 IDL usage conventions

IDL defines a number of features that are not useful for Fresco, including the "any" type, contexts, and unions. We also assume that a Fresco implementation will provide some mechanism to guarantee that class names do not clash with names in an application or other libraries. Since we do not use IDL modules, we assume that either the IDL language mapping or C++ implementation will handle this problem.

By convention, we use identifiers that begin with an upper case letter for types; we use lower case for operations and parameters. In type names consisting of multiple words, we capitalize the beginning of each word, as in FirstSecondThird. For operations or parameters we use underscores to separate words, as in first_second_third. An operation **f** for an interface *A* is denoted by A::f. Because IDL does not allows two distinct identifiers to differ only in case, in certain situations we use interface_op to avoid a class when an operation would naturally have the same name as a type.

2.2 IDL and Fresco implementations

Though we use IDL to specify the Fresco API, a Fresco-conformant implementation need not support either IDL or distribution. In particular, one can build a Fresco implementation without the existence of a complete CORBA environment or even IDL. The minimal requirements on a Fresco implementation are to provide a set of C++ header files that are compatible with the IDL interfaces in this specification and a library that implements those interfaces. If an implementation provides support for C, it must provide header files that are also compatible with the IDL interfaces. By "compatible," we mean that an application written in the context of one Fresco implementation will be portable to other Fresco implementations (at least with respect to the Fresco API—the application might include non-portable code as well).

2.3 Concurrency

Distribution increases the likelihood of concurrency because of the desire to take advantage of the available processing power. However, even in the absence of distribution, there are certain applications that are simpler when implemented using multiple threads. Within a visual interface to a file system, for example, we would like to avoid having long operations block user activity.

The minimum requirement for Fresco is to function properly in the presence of concurrent tasks sharing memory within an application, where more than one task is performing user interface operations at the same time. By "function properly," we mean the application does not deadlock and does not fault because of a race condition.

We believe many Fresco implementations will take advantage of concurrency by performing certain functionality in parallel with other operations. For example, in the sample implementation the screen redraw runs in a separate thread by default.

The consequence of using concurrency is that programmers who implement new objects will need to use locking to ensure safe access to data. Creating objects through a builder or program will not require any additional locking, as the objects should already be safe with respect to concurrent access.

We assume that concurrency is orthogonal to interface definitions; that is, Fresco objects do not by themselves define a thread of control. An object implementation may choose to allow only one thread access to the object's state at a time, but the form and granularity of concurrent access is entirely dependent on the type and implementation of the object.

We do not define threads, locks, or other concurrency abstractions as part of the Fresco specification. Instead, we assume that a Fresco implementation will use an implementation of threads provided by the underlying operating system. Where issues of concurrency are relevant in the specification of a Fresco operation, we state the minimal requirements for the implementation.

3. Basic types

The basic Fresco types provide common functionality such as strings, memory management, callbacks, coordinates and regions, update notification, and attribute management. The base interface for all objects is called FrescoObject. This interface defines operations for memory management and observation, but does not require any particular implementation and therefore does not incur a specific overhead.

Fresco defines a special object type, itself called Fresco, that provides access to kits and other top-level objects. Because this type may need to work with language-specific objects, we define the operations for a specific language. The Fresco class provides the necessary functionality from C++.

3.1 Strings

Fresco does not use the IDL string data type, as it is defined to be a sequence of 8-bit characters. To provide better support for multilingual environments, we define a CharString interface that is logically a sequence of CharCode values. A CharCode is a 32-bit integer that denotes a character in a standard character encoding such as Unicode.

CharString is not intended for large strings of text (a text object should handle these). The only access is therefore to extract all the characters and convert them directly to a sequence. By using indirect access, the CharString may be implemented using mixed length character encodings.

3.2 Memory management

Fresco does not require garbage collection of objects, though some implementations may rely on such an environment. The FrescoObject interface, from which all other interfaces are descended, defines operations for managing a count of references to a shared object. To allow manipulation of nil object references, the Fresco class defines static member functions for first testing if a reference is non-nil and then incrementing or decrementing the count.

We use the following conventions for placing calls to increment or decrement the count:

- A function that returns an object reference always delivers a counted reference for the caller. When finished with the reference, the caller is responsible for decrementing the count.
- An object that holds an object reference beyond the lifetime of the call must increment the count when storing the reference and decrement the count when no longer holding the reference.

3.3 Callbacks

A callback is an object one can pass to another object that the second object can use to perform an operation of the first object's choosing at some later time. Callbacks provide a level of indirection that allows the caller to be dynamically associated with the callee. In an imperative language, a callback may simply be a pointer to a function. However, in an object-oriented system, a callback must refer both to an object and the operation to perform on the object.

Fresco defines an Action type for simple callbacks with no parameters or return value. Typically, an action implementation will consist of an object and a pointer to a member function defined on the object's type. Because users may wish to undo the effects of an input event, the action interface defines operations both to execute and "unexecute" a callback.

3.4 Coordinates

User interface objects specify positions and sizes in terms of coordinates. The Fresco Coord type represents a position or distance in units of printer's points (1/72 of an inch), the PixelCoord type in units of pixels on the screen. Application objects should generally use only Coord to avoid undesirable behavior when viewing the object at a different resolution (such as on a different screen or a printer).

Fresco is neutral as to whether coordinates are 2D or 3D. All interfaces support 3D coordinates, but an implementation may choose to ignore Z-related data.

The Vertex type defines a structure with X, Y, and Z coordinate values. The Transform type defines a 4x4 matrix for scaling, rotating, or translating coordinates. The Region type defines a portion of the possible coordinate space.

The Axis type specifies a dimension of cartesian coordinates. Many user interface operations, especially involving layout, can be parameterized by dimension.

3.5 Notification

User interface objects need to be notified when a data object changes. In many cases, the notification interface is domain-specific. For example, a text object may wish to notify its viewers that a specific range of the text has been modified and as well as the particular kind of modification (insertion or deletion). A general-purpose interface is therefore useful, but we assume that many objects will also need to define additional interfaces.

The base FrescoObject interface defines five operations to support notification: attach, detach, notify_observers, disconnect, and update. The first three (attach, detach, and notify_observers) are operations on the object under observation. The other two operations, disconnect and update, are called on the observer.

3.6 Styles

We define a base type, Style, that provides a common dynamic name-based interface for a hierarchy of styles. Subtypes of Style may define a static interface with operations for setting and getting specific values, such as fonts or colors. We can narrow to determine whether a style supports a particular set of values. That is, given a style, we can test whether it is actually an object that supports the static interface of a subtype.

Inheritance means that retrieving a value may require searching up the Style hierarchy, looking for a context that both defines the interface supporting the value and defines the value. Alternatively, values can be cached and a notification sent when the original value is modified.

4. Resolution independence

Fresco coordinates are resolution independent because they specify an area that can be mapped to the appropriate set of pixels on the screen using the screen's dots-per-inch ratio. Because the basic Xlib functions are pixel-based, Fresco provides a set of interfaces for drawing and window management that are resolution-independent.

The Display type is the unit of window system control, typically consisting of a single screen, keyboard, and a mouse or other pointing device. The Screen type defines the size, resolution, and color properties of a physical screen associated with a display. The Window type defines a rectangular area within a screen and provides the interface to an external window manager.

The DrawingKit type is a kit for finding and creating objects that represent drawing state, including brush, color, fill pattern, font, and raster images. Because name resolution for objects that represent graphics state is typically provided per display, a drawing kit is actually associated with a specific display. However, the objects that the drawing kit returns still may be used on other displays.

The Painter type defines a simple set of stencil/paint drawing operations. Painter is *not* intended to be either a novel or a complete set of rendering operations. Painter provides a small subset of the functionality offered by PostScript. An application object needing additional functionality should use a richer rendering interface, such as Display PostScript, OpenGL, or PHIGS+. These interfaces are becoming broadly available, and the focus of Fresco is on the object organization rather than rendering.

5. Graphical Embedding

We refer to the composition of objects within a window as "embedding" to indicate that one object may contain one or more other objects. Embedding encompasses both presentation, which involves the *spatial* composition of user interface objects, and editing, which involves the *logical* composition of objects. The problems of presentation include layout, rendering, and hit detection; the problems of editing include input focus management, event routing and translation, and transcription to persistent storage.

In some older systems, such as Microsoft's OLE-1, embedding is a relatively loose connection between two objects. The outer object, which we will call the *container*, presents the inner object, which we will call the *embedded object*, using an image produced by the embedded object. To modify the embedded object, the user must open a separate window on the embedded object and updates will be sent to the container. More recent systems, including OLE-2, support "edit-in-place" embedding, meaning the embedded object can be modified directly while inside the container. Edit-in-place embedding requires a tighter integration between the container and the embedded object to coordinate presentation and input handling.

When we use the term embedding, we mean edit-in-place. We use the term graphical embedding to mean edit-in-place embedding with the additional functionality that a container may perform arbitrary graphical transformations, such as scaling and rotation, on embedded objects. For example, consider a drawing that may contain geometric objects as well as text labels. The drawing editor supports the creation and altering of geometric objects, as well as moving, scaling, or rotating any object. An embedded text editor supports entering and altering characters in labels. In a system like OLE-1, the user must open a separate window to edit the text. In a system like OLE-2, the user may edit the text inside the drawing, but a general text editor that is independent of the drawing editor would not be able to edit rotated text in place. With graphical embedding in Fresco, we will be able to edit the text in place within the context of a graphical transformation.

5.1 Spatial composition

Containers and embedded objects must coordinate how they affect the display, both in terms of negotiating positions and rendering the correct image. We organize this functionality into three parts: the structure for rendering, layout negotiation, and the screen update mechanism that embedded objects must use to guarantee the visible image is correct.

5.1.1 Structure

The presentation structure is a directed acyclic graph (dag) of primitive objects called *glyphs*. Glyphs may be shared; that is, two composite glyphs may contain references to the same glyph (one composite also could reference the same glyph twice). Sharing offers both semantic and performance advantages. From a semantic point of view, a single change can effect all the uses of a glyph. From a performance perspective, sharing can significantly reduce the memory cost associated with a frequently-used object.

Many different operations can be performed on all the glyphs in a dag. A *glyph traversal* object encapsulates the operation to perform at each glyph, allowing container glyphs to define the order in which to traverse their components independently from the operation being performed. A traversal also encapsulates the sequence of glyphs from the root to a particular glyph, thereby allowing one to compute bounding box information.

Sharing also implies that a glyph reference does not distinguish it with respect to its container, as it could appear more than once in the container. A *glyph offset* is an object that uniquely denotes a child within a container.

5.1.2 Layout

Fresco provides a one-pass layout negotiation between the container and the object: the container asks the object what it wants, then tells it what it gets. A container asks a object to fill out a *requisition* that describes the component's *requirement* in each dimension, where a requirement specifies natural, minimum, and maximum sizes and an origin. The object may also specify that the aspect ratios of its natural sizes should be preserved whenever possible.

During traversals that may involve layout, such as rendering, printing, and picking, a container may define an *allocation* for each child. An allocation is a 3-dimensional region in the container's object coordinates solely for the purposes of layout; it does not imply clipped output. Very slanted text or a shadow, for example, could extend outside a text object's allocation.

5.1.3 Screen update

Fresco objects do not update the screen directly, but ask for a redraw and then let the system call draw on the object in the future. Logically, updating the screen involves redrawing all the objects. However, if redraw requests can restrict the region that needs to be redrawn then output will be clipped to that region. 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.

Deferred update has three desirable effects. First, it allows objects to share the same area of the screen because whether to redraw is determined by the clip region rather than which object has changed. For example, a label can appear on top of an image and the label may need to be redrawn if the image asks to be redrawn. A direct update approach, where the image draws directly to the screen when it changes, cannot handle sharing screen area without explicit knowledge between objects (the image would need to know that the label is above it).

The second effect is that output may be double-buffered to give the user a smoother, less flashy appearance. Although traditionally associated with animation, double-buffering also can improve the appearance of common interactive techniques such as scrolling, text selection, and drag-and-drop. Double-buffering in combination with partial update (where the redraw 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 again.

The third effect of deferred update is the ability of objects to determine what granularity of update is most efficient. Objects that are cheap to draw usually do not check clipping; objects that are expensive to draw usually do.

5.1.4 Dynamic extensibility

Because glyphs are cheap objects, one can afford to define a glyph object that delegates some or all of its glyph behavior to another glyph. Such glyphs are normally *structurally invisible*, meaning that operations that affect composite glyphs are passed directly to the subordinate glyph. For example, one

could define a shadow glyph that draws a drop shadow under another glyph. One can put a shadow around a composite without affecting operations on the composite.

We can use delegation for many purposes: to perform input handling or filtering, to decorate with shadows, borders, or bevels, to perform layout alignment such as centering, and to adjust a glyph's requisition (making a rigid glyph become stretchable, for example). Another use is to provide a special glyph for debugging that displays parameter values before passing operations on to the subordinate.

Delegation provides a general way to alter the behavior of a glyph without the complexity of subclassing. The behavior adjustments can also be dynamic--a new glyph that delegates can be added or removed around another glyph at runtime.

5.2 Logical composition

Embedding has two sets of requirements: one for an object's role as a container and one for the role of the embedded object. All user interface components must satisfy the latter role--even an object at the outermost level of a document will be embedded inside a special object called the *main viewer*. The role of a container is optional; some components may refuse to allow embedding of other components because they cannot define a useful meaning to this form of embedding. For example, an image editor may operate solely on arrays of pixels; the editor may refuse to accept requests to embed arbitrary components.

5.2.1 Separating the user interface from the data

An important part of embedding is the separation of the *data* for a component from a *viewer* of that data. This separation is necessary to simplify both the construction of the user interface and the transcription of the data to and from persistent storage. Separating data and user interface also has the benefit of allowing multiple viewers to be active at the same time. For example, a user could bring up two viewers on the same text data, possibly looking a different positions within the text in each viewer.

5.2.2 Inset: Data that refers to an embedded object

When a user asks to save a component, the relevant data must be saved including embedded objects. In addition to the data associated with the embedded object, we also must store the type of viewer and any persistent viewer-specific information, such as current user preferences. An *inset* is a data object that encapsulates this viewer information associated with an embedded object.

The purpose of an inset is to be a general-purpose transcription of viewer information. In particular, one should be able to create a slightly different viewer for the same inset. For example, an inset for text data might refer to a viewer that formats the text into pages. A user should be able to open this inset with only a viewer that scrolls text available. Thus, an inset provides a layer of abstraction between the transcribed data and the specific viewer. The figure below shows an example viewer-data structure for a graphics

object embedded inside a text object. The rectangles represent data objects; the ellipses represent viewers.

5.2.3 Transcription

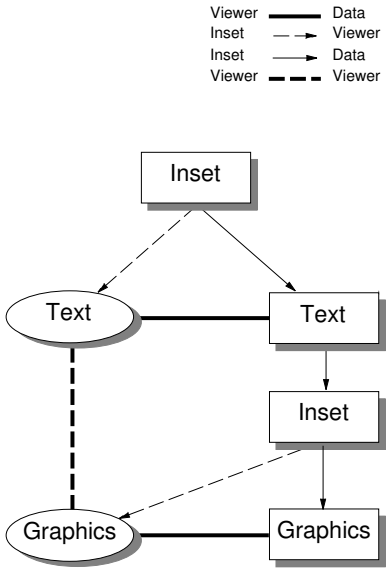
An object transcription is a persistent representation of the object in an externally-controlled format. The external definition of the format is what distinguishes transcription from persistence or object-oriented databases. In a persistent store or database, the system designer can choose one format for objects. However, for Fresco we need the ability to read and write several formats that may be defined by other vendors or standards groups.

Transcription therefore must be extensible in two directions: we must be able to add new objects to existing formats and we must be able to add new formats for existing objects. This requirement means we cannot use the common approach of having a read and write operation on each object. Instead, we define a Transcriber object for each format. This object defines operations to read and write objects. We must be able to distinguish the object types by some form of run-time type information.

In addition to transcribers for a specific format, we need to ensure that a transcriber can write an object back in its original format as well as the object's "natural" format. Important compound document formats include MIME and RTF; there are also a number of domain-specific formats, such as TIFF for images and AIFF for audio.

5.2.4 Input events

The main viewer at the root of an embedding hierarchy is responsible for delivering events to the appropriate viewer. Events typically come from one of two sources: focusable devices, such as keyboard or voice, and positional devices, such as the mouse.



For a focusable device, the main viewer will use a translation table to map the raw event into an action. An implementation of an action may interpret a script to execute the command.

For a positional device, the main viewer calls the handle operation on the outermost embedded object. This operation passes the operation down until the appropriate viewer is found to handle the event.

A viewer may *filter* positional devices, guaranteeing the viewer will receive all events from the device until the filter is removed. Filtering is useful for a viewer that needs to know when the device position leaves the viewer, such as a button that highlights when a pointer is inside it.

5.3 Scripting

Fresco supports scripting in two ways. First, the Action type effectively defines an executable object that can be implemented by a script or grouped to replay a script of user commands. Second, all Fresco operations are accessible through the dynamic invocation mechanism of CORBA. This mechanism allows an application to create a request object based on run-time data (such as a string containing the operation name) and invoke the request to perform the specified operation.

6. Summary

Fresco uses the CORBA standard object model, provides resolution-independent windowing primitives, and supports a uniform architecture for graphical embedding. Using CORBA also allows the possibility of distributed embedding, where an embedded application runs on a different machine in the network.

Standard Definitions

Action

interface Action : FrescoObject

An action executes some functionality with no parameters and no return value. An action may be undo-able, though it is up to the action itself to save the necessary state for undo.

execute

void execute();

Perform the action.

reversible

boolean reversible();

Return whether the action can be undone.

unexecute

void unexecute();

Undo the previously done action. If the action has not been done or was most recently undone, then this operation has no effect. This operation is also likely to fail if intermediate operations have been executed (and not unexecuted) since the execute operation was most recently called.

Alignment

typedef float Alignment

An alignment is a floating point value that represents a relative position. Alignments are typically between zero and one inclusive, with zero indicating the position is aligned with the lower point and one indicating alignment with the higher point. For example, an alignment of zero along the X-axis indicates that the position is left-aligned, a value of one means the position is right-aligned.

Axis

enum Axis { X_axis, Y_axis, Z_axis }

Axis is a type for specifying a dimension of cartesian coordinates. Many user interface operations, especially involving layout, can be parameterized by dimension.

Brush

interface Brush : FrescoObject

A brush defines the line thickness and line style for drawing operations. The effect of these

operations is as if a line segment equal in length to the brush's width were dragged along an infinitely thin path between the specified coordinates. At each point along the path the brush is angled perpendicular to the path. As a special case, a brush width of zero specifies a minimal-width line. Many devices can render minimal-width lines more quickly than wide lines, but the resulting display may vary slightly across devices.

A solid brush style paints all pixels along the path with a single color. A dashed brush defines alternating foreground and background segments, measured along the length of the path. Foreground segments are painted, while background segments are not.

equal, hash

boolean equal(in Brush b);
unsigned long hash();

The equal operation tests whether two brushes have the same line width and style. The hash operation will return the same value for two equal brushes, though there is also some chance it will return the same value for two unequal brushes.

width

Coord width();

Return the width of a brush.

CharCode

typedef long CharCode

CharCode is a value that denotes a character from a particular character set, such as Unicode.

CharString

interface CharString : FrescoObject

CharString represents a sequence of characters, where a character is a logical value rather than a specific physical size.

CharData

typedef sequence<char> CharData;

The CharData type is useful for extracting the data from 8-bit strings.

copy

void copy(in CharString s);

Copy the contents of the given string into this string.

count

long count();

Return the number of characters in the string.

Data

typedef sequence<CharCode> Data;

The data in a character string can be interpreted as a sequence of character codes.

equal, case_insensitive_equal

boolean equal(in CharString s);
boolean case_insensitive_equal(in CharString s);

Compare two strings and return true if they contain the same characters. The case_insensitive_equal operation returns true if the strings only differ in case.

get_data, get_char_data

void get_data(inout Data d);
void get_char_data(inout CharData d);

Retrieve the characters in the string as a sequence of either 8-bit values or character codes. The parameters are inout so that a caller can provide a buffer.

hash

HashValue hash();

Return a hash value for the string.

HashValue

typedef unsigned long HashValue;

A hash value is computed from the characters in the string. If the hash values of two strings are different, then the strings differ in content. If the hash values are the same, the strings may or may not differ.

put_data, put_char_data

void put_data(in Data d);
void put_char_data(in CharData d);

Store the given 8-bit or character code data into the string.

Color

interface Color : FrescoObject

A color object represents a logical color for output.

equal, hash

boolean equal(in Color c);
unsigned long hash();

The equal operation tests whether two colors are logically the same (the exact comparison depends on the underlying color model). The hash operation will return the same value for two equal colors, though there is also some chance it will return the same value for two unequal colors.

Intensity

typedef float Intensity;

A color intensity is a fraction between zero and one inclusive, where a value of zero indicates no intensity (and therefore will not be visible) and a value of one indicates full intensity.

rgb

void rgb(out Intensity r, out Intensity g, out Intensity b);

Return the RGB values for this color. RGB values are device-dependent and should be avoided when possible. The returned values are relative to the display that created the drawing kit (see Display::drawing_kit).

Coord

typedef float Coord

User interface objects specify positions and sizes in terms of coordinates. The Coord type represents a position or distance in units of printer's points (1/72 of an inch). Coord might better be defined as an interface from the perspective of abstraction, but for practical reasons it is defined as a typedef. For applications with simple graphics needs, the use of floating point coordinates is typically not a problem. Applications with more sophisticated text or graphics usually need to manipulate floating point coordinates anyway.

Cursor

interface Cursor : FrescoObject

A cursor is an image that tracks a pointing device's position. Cursors may be defined by two bitmaps and a “hot spot.” The mask bitmap specifies which pixels are to be drawn, and the pattern bitmap specifies which pixels are in foreground color and which are in background color. The hot spot specifies the location of the pointing device relative to the cursor's lower-left corner.

Damage

interface Damage : FrescoObject

A damage object represents the smallest unit of screen update. Typically, there will be one damage object for each main viewer. Before a draw traversal begins, the damage object is normally used to set the painter's clipping region and then reset to be empty.

current

Region current();

Return the current region that has been requested for update.

extend

void extend(in Region r);

Extend the current update area to include the given region.

incur

void incur();

Request an update on the entire region defined by the damage.

Display

interface Display : FrescoObject

A display is the unit of window system control; typically it consists of a single screen, keyboard, and a mouse or other pointing device. Application objects typically need not perform operations directly on a display.

auto_repeat

attribute boolean auto_repeat;

If the value of this attribute is true, then the display's keyboard will auto-repeat keys that are depressed.

close

void close();

Terminate the connection to the display. Once a connection is closed, it cannot be re-opened.

cursor_from_data, cursor_from_bitmap
cursor_from_font, cursor_from_index

```
Cursor cursor_from_data(
    in short x, in short y, in long pat[16], in long mask[16]
);
Cursor cursor_from_bitmap(in Raster b, in Raster mask);
Cursor cursor_from_font(in Font f, in long pat, in long mask);
Cursor cursor_from_index(in long n);
```

These operations return cursors that can be used on windows on the display.

display_screen, number_of_screens, default_screen

```
Screen display_screen(in Display::ScreenNumber n);
Display::ScreenNumber number_of_screens();
Screen default_screen();
```

For displays with multiple screens, the screen operation returns the screen object corresponding to the given index. The number_of_screens returns the total number of screens the display has; a screen number should be between zero and the number of screens minus one. The default_screen operation returns the default screen as defined by the display.

display_style

Style display_style();

Return the style associated with a display.

drawing_kit

DrawingKit drawing_kit();

In addition to screen access and update, a display provides a name service for drawing objects such as colors and fonts. The drawing_kit operation on a display returns a drawing kit that provides this service.

flush, flush_and_wait

```
void flush();
void flush_and_wait();
```

Repair all damaged windows on the display and ensure that any pending requests have been sent to the window system. The flush_and_wait operation additionally waits for an acknowledgment from the window system indicating that the request have been processed.

key_click_volume

attribute float key_click_volume;

The key click volume is specified as a fraction of the possible loudness. For example, a value of 0.75 specifies a volume that is three-quarters as loud as is possible. If the value is not positive, no sound will be made. If the value is greater than or equal to one, then the volume will be as loud as possible.

need_repair

void need_repair(in Window w);

Notify the display that the given window has been damaged and should be repaired the next time the display is repaired. If called during event processing, the need_repair operation will put the window on a list to be repaired before blocking for input. Otherwise, if need_repair is called asynchronously while the input thread is blocked, then the display will be repaired immediately.

pointer_acceleration, pointer_threshold

attribute float pointer_acceleration;
attribute long pointer_threshold;

These two attributes control the pointer responsiveness. Pointer movement will be multiplied by the acceleration when it moves more than the number of pixels specified by the threshold.

pointer_double_click_time

attribute float pointer_double_click_threshold;

This attribute holds time difference, in seconds, between two clicks below which the second click may be interpreted as a double click event. The default value is 250 milliseconds.

repair

void repair();

Repair all damaged windows on all screens on the display. This operation usually need not be called directly from application objects. Incurring or extending damage is normally sufficient to cause a repair to happen in the near future.

ring_bell

void ring_bell(in float pct_loudness);

Sound the system bell. The parameter specifies the volume as a fraction of the possible loudness. For example, a value of 0.75 specifies a volume that is three-quarters as loud as possible. If the value is not positive, no sound is made. If the parameter value is greater than or equal to one, then the volume is as loud as possible.

run, running

void run(in boolean b);
boolean running();

Given a value of true as its parameter, the run operation executes a main loop that reads and dispatches input events, terminating when the run operation is called with a value of false. The run operation is also responsible for handling screen update, though redisplay will typically be handled by a separate thread. The running operation returns the current state of the loop.

ScreenNumber

typedef long ScreenNumber;

The ScreenNumber type defines the representation of screen numbers.

DrawingKit

interface DrawingKit : FrescoObject

The DrawingKit interface defines operations for creating common drawing objects, such as brushes, colors, and fonts. The initial creation of these objects may be display-dependent because the naming of objects is often resolved per-display. However, the objects themselves are not display-dependent; they may be used on displays other than the one associated with the drawing kit.

bitmap_from_data

```
Raster bitmap_from_data(
    in Data8 d,
    in Raster::Index rows, in Raster::Index columns,
    in Raster::Index origin_row, in Raster::Index origin_column
);
```

color_rgb, find_color

```
Color color_rgb(in Color::Intensity r, in Color::Intensity g, in Color::Intensity b);
Color find_color(in CharString name);
```

These operations return color objects. The objects may be newly-created or may be shared with other users of the same color. The color_rgb operation returns a color with the given RGB intensities, as defined relative to the drawing kit's display. The find_color operation returns the color object (if any) associated with the given name. Color name tables may be managed per-display, so the result of this operation also depends on the drawing kit's display.

Data8, Data32

```
typedef sequence<octet> Data8;
typedef sequence<long> Data32;
```

These types define sequences of 8-bit or 32-bit values.

drawing_style

```
Style drawing_style();
```

Return the default style for the drawing kit, as defined by the display from which the drawing kit was created.

find_font, find_font_match

```
Font find_font(in CharString fullname);
Font find_font_match(
    in CharString font_family, in CharString font_style, in Coord ptsize
);
```

The find_font operation returns the font object defined by the given full name. The find_font_match operation finds the best-matching font for the given family, style, and point size. Both operations require a lookup that may be display-dependent.

printer

```
Painter printer(in CharString output);
```

Return a painter that generates printer output. The interpretation of the parameter string is system-dependent. It could, for example, denote the name of a file in which to generate PostScript. Alternatively, it could contain the name of a system printer where output should be sent.

raster_open

Raster raster_open(in CharString name);

Return a raster image specified by the given name. Typically, the name is a file in some known image format such as an X bitmap or TIFF file.

resolve_color, foreground, background

Color resolve_color(in Style s, in CharString name);
Color foreground(in Style s);
Color background(in Style s);

These operations return a color defined for a particular style. The operations resolve a name in the style to a string value and then do the equivalent of a find_color operation on that value. The resolve_color operation looks for a color by name. The foreground operation is equivalent to resolve_color with the name “foreground,” the background operation looks for the name “background.”

resolve_font, default_font

Font resolve_font(in Style s, in CharString name);
Font default_font(in Style s);

The resolve_font operation returns a font specified for a particular style. This operation is equivalent to resolving the name in the style to a string value and calling find_font on that value. The default_font operation is equivalent to calling resolve_font on the name “font.”

Return a bitmap created from the given data, sizes, and origin.
bitmap_from_char

Raster bitmap_from_char(in Font f, in CharCode c);

The bitmap_char operation creates a bitmap for a given character in a font. In this case the bitmap width and height will reflect the actual size of the character and the bitmap origin will correspond to the character's left bearing and baseline.

simple_brush, dither_brush, patterned_brush

Brush simple_brush(in Coord width);
Brush dither_brush(in Coord width, in long p);
Brush patterned_brush(in Coord width, in Data32 p);

These operations create brushes. The brush operation creates a solid brush with a given width, the dither_brush operation creates a brush with a line style specified by a bit vector. The least significant 16 bits of the pattern parameter are interpreted as a bit pattern, with ones specifying foreground segments and zeroes specifying background segments.

The patterned_brush operation defines the line style from an array of 32-bit integer that specify the length of successive foreground and background segments. Even-numbered array indices (starting from zero) specify the length of the foreground segments, odd-numbered indices specify the background segments. The count is the number of entries in the array. The count can be zero, which specifies a solid brush.

solid_pattern, halftone_pattern,
stipple, stipple_4x4

Pattern solid_pattern();
Pattern halftone_pattern();
Pattern stipple(in Data32 d);
Pattern stipple_4x4(in unsigned long p);

transform_matrix, identity_matrix

Transform transform_matrix(in Transform::Matrix m);
Transform identity_matrix();

Return a new transformation matrix. The transform operation is given a complete matrix, the identity_transform returns an identity matrix.

Font

interface Font : FrescoObject

A font defines a mapping between character codes and their appearance on the screen.

encoding

CharString encoding();

Return the character set identification, such as “iso8859” for ISO Latin.

equal, hash

boolean equal(in Font f);
unsigned long hash();

The equal operation tests whether two font objects refer to the same font information. The hash operation will return the same value for two equal fonts, though there is also some chance it will return the same value for two unequal fonts.

font_info, char_info, string_info

void font_info(out Font::Info i);
void char_info(in CharCode c, out Font::Info i);
void string_info(in CharString s, out Font::Info i);

Return font metric information. The font_info operation returns summary metrics for all characters, char_info returns metrics for a particular character, and string_info returns metrics for a string of characters.

Info

```
struct Info {
    Coord width, height;
    Coord left_bearing, right_bearing;
    Coord ascent, descent;
    Coord font_ascent, font_descent;
};
```

The Info structure defines font metric information for a character, string, or summary information for the entire font. The width and height fields define the overall dimensions. The left_bearing and right_bearing fields denote the left and right edges inside the width. The ascent and descent fields define the distance to the baseline from the top and bottom, respectively. The font_ascent and font_descent fields return the summary baseline information for the font, as opposed to the character-specific distances provided by ascent and descent.

name

CharString name();

Return the full name of the font. The syntax of the name is platform-dependent.

point_size

Coord point_size();

Return the point size of the font.

Fresco

class Fresco : public BaseObject

Fresco is a C++ class that defines operations for accessing other Fresco objects. It is defined as a class rather than an interface to provide support for the C++ "char*" data type and static memory management operations. The Fresco class also provides access to non-standard kits.

Typically, there will only be one Fresco object per address space that is created when the address space is initialized. The Fresco object will hold references to kits, which in turn may hold other references. Therefore, to avoid a reference count circularity objects should not hold a reference to the Fresco object. Instead, objects can assume that the Fresco object will live at least as long as any object.

argc, argv

virtual long argc() = 0;
*virtual char** argv() = 0;*

Return the argument list used when Fresco object was opened. This information is normally only needed for registration with a session manager.

class_name

virtual CharStringRef class_name() = 0;

Return the class name associated with a Fresco object. This operation is normally only needed when opening a display and initializing the display's style information.

delay

static Boolean delay(Float seconds);

Pause execution the given number of seconds or fraction of a second. Return true if execution was successfully suspended the requested duration.

fresco_style

virtual StyleRef fresco_style() = 0;

Return the root Fresco style. This style will contain default and command-line attributes, but not display-specific or user-specific attributes.

get_string, get_tmp_string

static CharStringRef get_string(Style_in, const char name);*
static CharString_var get_tmp_string(Style_in, const char name);*

Return the string bound to the given name in the given style. The get_tmp_string operation returns a managed rather than a raw reference. These operations are effectively short-hand for creating a string, calling resolve on the style, and calling read_string on the result if non-nil. Both operations return a nil string if either the name is not defined in the style.

open_display, open_default_display

virtual Display open_display(CharStringRef name) = 0;*
virtual Display open_default_display() = 0;*

Open a display by name or open the default display. The name of the default display is platform-specific; on POSIX it can be set by the DISPLAY environment variable.

Display names are of the form N:D.S, where N is an optional network host name. If omitted, then the display is assumed to be on the local host. D is a display number on the host, and S is the number of the screen to use on the display by default.

ref, unref

static void ref(BaseObjectRef);
static void unref(BaseObjectRef);

The ref and unref operations respectively increment and decrement the reference count of a Fresco object. If the count is not positive after an unref, then the object may be destroyed. These operations are static rather than operations on BaseObject to allow for nil references, which are ignored.

string_ref, tmp_string_ref, string_copy, tmp_string_copy

static CharStringRef string_ref(const char);*
static CharStringRef string_copy(const char);*
static CharString_var tmp_string_ref(const char);*
static CharString_var tmp_string_copy(const char);*

Create a string from a null-terminated array of characters. The string_ref operation simply references the data; the string_copy operation copies the data into its own storage that is freed when the string is deallocated. The tmp_string_ref and tmp_string_copy operations return a managed rather than raw object reference.

Fresco_init

extern Fresco Fresco_init(*
const char name, int& argc, char** argv, Option* options = nil*
)

This function returns a Fresco object with which one can access other objects. The name parameter specifies the name of the root style. The argc and argv parameters should be the command-line parameters, the same as one would pass the function “main” in C. Unlike Fresco_open, this function does not open a display and does not create an initial drawing kit.

Fresco_open, Fresco_open_display

extern Fresco Fresco_open(*
const char name, int& argc, char** argv, Option* options = nil*
)
extern Fresco Fresco_open_display(*
const char display_name,*
const char name, int& argc, char** argv, Option* options = nil*
)

These functions returns a Fresco object with which one can access other objects. The name parameter specifies the name of the root style. The argc and argv parameters should be the

command-line parameters, the same as one would pass the function “main” in C.

Both functions open a display and create an initial drawing kit from the display. Fresco_open uses the default display name; Fresco_open_display uses the given display name. Both functions return nil if they cannot successfully open the display.

FrescoObject

interface FrescoObject

All Fresco objects support the FrescoObject interface, which defines operations for accessing an object's type and for holding a reference to an object.

attach, detach, disconnect

Tag attach(in FrescoObject observer);
void detach(in Tag attach_tag);
void disconnect();

To avoid circularities, an object may have an “uncounted” reference to another object. To ensure the reference is valid, the holder of an uncounted reference may “attach” to the referenced object, and “detach” when no longer storing a reference to the object or be notified via the “disconnect” operation if the object is destroyed. The tag returned from the attach operation is the value that should be passed to the detach operation.

One may call the attach and detach operations on any object, but the object's implementation may choose to ignore these requests. Thus, this functionality has no *required* overhead per object. This approach allows a simple interface to objects while giving flexibility to implementations.

notify_observers, update

void notify_observers();
void update();

Uncounted references (see the attach operation) provide a simple mechanism for update notification. All objects support a “notify_observers” operation for the object being observed and an “update” operation for the observer.

Glyph

interface Glyph : FrescoObject

The Glyph interface defines the common operations for all objects that have a geometry and may be drawn. The name “glyph” is borrowed from typography to connote simple, lightweight objects with an instance-specific appearance. Glyphs also have a size and origin for the purposes of layout. The Glyph interface provides operations for geometry management, drawing, picking, and a directed-acyclic structure (not strictly hierarchical).

add_parent, remove_parent

Tag add_parent(in GlyphOffset parent_offset);
void remove_parent(in Tag add_tag);

Add or remove a parent for this glyph. Because a glyph could possibly be added more than once as a child of the same parent, an offset represents the parent edge. The add_parent operation

returns a tag that the offset retains to remove itself from the child's parent list when the edge is removed. The child should not hold a reference to its parent (otherwise it would create a cycle), so the parent_offset is not passed with a reference and should not be unref'd by the add_parent operation.

AllocationInfo

```
struct AllocationInfo {
    Region allocation;
    Transform transformation;
    Damage damaged;
};
```

The AllocationInfo structure contains specific cumulative geometry information. This information is used to determine what regions to damage in order to redraw a glyph.

AllocationInfoSeq

```
typedef sequence<AllocationInfo> AllocationInfoSeq;
```

AllocationInfoSeq is a list of AllocationInfo structures to handle the possibility that a glyph appears several places at once on the screen.

allocations

```
void allocations(out Glyph::AllocationInfoSeq a);
```

Compute cumulative information for each allocation of the glyph. This operation normally need not be called directly; instead application objects usually call need_redraw or need_resize.

append, prepend

```
void append(in Glyph g);
void prepend(in Glyph g);
```

Add the given glyph to this composite. If the receiver is not actually a composite, the operation will be ignored. If the receiver has a body glyph, it should pass the operation on to the body. In this way, “wrapper” glyphs can alter appearance without affecting structural operations.

body

```
attribute Glyph body;
```

The body attribute defines a glyph as the recipient of operations. Glyphs that have a body defined are typically “filters” that decorate, align, or otherwise adjust the appearance of the body.

clone_glyph

```
Glyph clone_glyph();
```

Return a logical copy of this glyph. Style information for the target glyph is also copied, but inherited styles are not copied. The target data may be locked while the copy is in progress.

draw

```
void draw(in GlyphTraversal t);
```

Display this glyph using the painter and allocation specified by the given traversal. No clipping or translation is automatic for this operation--the allocation specifies the position and size of this glyph for the purposes of layout. However, a composite glyph may set clipping or transformations on the traversal's painter before the composite draws its children.

There is no guarantee that a call to draw will affect the screen immediately because of the possibility (indeed, likelihood) that the window is double-buffered. Furthermore, drawing order

is important for glyphs when they overlap in the same plane. A glyph's draw operation therefore should not be called directly, but implicitly will be called as part of the screen update process.

extension

void extension(in Glyph::AllocationInfo a, in Region r);

This operation extends the given region by the bounds of where this glyph would draw when given the allocation and transform defined by the AllocationInfo.

first_child_offset, last_child_offset

GlyphOffset first_child_offset();
GlyphOffset last_child_offset();

Return the offset of the first or last of this glyph's list of children. Both operations will return nil if the glyph has no children.

glyph_style

attribute Style glyph_style;

A glyph may use the style attribute to customize the font, colors, and other characteristics of the glyph's appearance. Usage of the style attribute is optional--a glyph may choose to ignore setting of the style and may return nil when the style is retrieved.

need_redraw, need_redraw_region

void need_redraw();
void need_redraw_region(in Region r);

These operations request that a glyph cause its appearance on the screen to be refreshed. If the glyph appears in multiple places in a window or in several windows, the glyph should be redrawn in all the places. A glyph may choose not to implement these operations under the assumption that update is either not possible or will be handled by higher-level glyphs.

The need_redraw operation asks that the entire glyph be redrawn; need_redraw_region specifies a portion of the glyph relative to the glyph's coordinate space. Note these operations do not actually cause an immediate screen update; they merely notify the system that a draw traversal is desired in the near future.

need_resize

void need_resize();

This operation is similar to need_redraw, but includes the additional possibility that the glyph's natural size has changed. This operation implies two redraw requests (which will be handled with a single draw traversal), one for the glyph's original area and one for the glyph's new area.

OffsetSeq

typedef sequence<GlyphOffset> OffsetSeq;

OffsetSeq is a list of glyph offsets.

parent_offsets

void parent_offsets(inout OffsetSeq parents);

Get the offsets of the parents of this glyph. The parameter is inout to allow the caller to allocate a fixed size array for the result. If this glyph has more parents than the maximum size given for the list, then it will allocate new storage for the list. The caller therefore must check to see whether the resulting size is smaller or now to deallocate the list elements correctly.

pick

void pick(in GlyphTraversal t);

Find the glyphs that intersect a point or rectangle specified in window-relative coordinates. Conceptually, picking is like drawing and determining what glyphs intersect the point or region. When pick returns, the traversal contains a list of the glyph trails that were hit.

request

void request(out Glyph::Requisition r);

This operation asks for a glyph's desired geometry. The computation should be fast to allow a composite glyph to compute rather than store its components' requisitions. In general, if a requisition might be expensive to compute then a glyph should store its own requisition. A composite glyph therefore generally should store its requisition rather than traverse its components. A composite can assume that GlyphOffset::notify will be called if the requisition of one of its children changes.

Requirement

```
struct Requirement {
    boolean defined;
    Coord natural, maximum, minimum;
    Alignment align;
};
```

If the defined flag is true, a requirement specifies a natural size, maximum size, minimum size, and alignment. A composite may “stretch” a requirement from its natural size to its maximum size or “shrink” the requirement from the natural to minimum size. It is possible to have negative sizes, such as for “backwards” space. The alignment is a fraction that indicates the origin of the area. An alignment of zero means the origin is at the lower coordinate value; an alignment of one means it is at the upper coordinate value. Alignments can be negative or larger than one to indicate the origin is outside the natural position.

Requisition

```
struct Requisition {
    Requirement x, y, z;
    boolean preserve_aspect;
};
```

A requisition contains a requirement for each axis and a flag indicating whether the natural aspect ratios should be preserved if the glyph is stretched or shrunk.

restore_trail

boolean restore_trail(in GlyphTraversal t);

This operation attempts to restore the traversal state (allocation and drawing state) for the given traversal The return value indicates whether restoration is successful.

shape

void shape(in Region r);

This operation extends the given region by the precise shape where this glyph would naturally draw. Calling shape may be more costly than calling extension (which just computes bounds) and therefore should only be called by composites that wish to perform layout using the actual area where the object would draw.

transformation
Transform transformation();

Return a transformation matrix associated with the glyph. If a glyph implementation does not provide a transform, this operation will return nil.

traverse
void traverse(in GlyphTraversal t);

Start traversing a glyph and its components. The traversal object defines the operation on each glyph. Normally, a composite will call traverse recursively on each of its components.

N.B.: Because traversal state must be held by the caller past a call to traverse, an extra reference to the traversal parameter object is *not* passed to the traverse operation. Therefore, contrary to normal conventions, an implementation of the traverse operation should *not* unref the parameter.

GlyphOffset

interface GlyphOffset : FrescoObject

A glyph offset is a reference to a glyph within an aggregate. Because a glyph might appear twice within the same composite, a glyph itself is insufficient to identify its position within its parent.

allocations
void allocations(out Glyph::AllocationInfoSeq a);

Compute cumulative information for each allocation of the glyph. This operation normally need not be called directly; it is normally called by an implementation Glyph::allocations.

child_allocate
void child_allocate(inout Glyph::AllocationInfo a);

Determinte the allocation for the child given the allocation information for the parent.

insert
void insert(in Glyph g);

Insert the given glyph into the composite before the glyph referred to by the target offset.

next_child, prev_child
GlyphOffset next_child();
GlyphOffset prev_child();

Return the next or previous offset within the list of children.

notify
void notify();

Notify the composite that the desired size of the glyph at this offset has changed.

parent, child
Glyph parent();
Glyph child();

Return the composite parent and child glyph to which the offset refers.

remove
void remove();

Take the glyph at this offset out of the composite. The offset will now refer to the following glyph. If the offset refers to the end of the composite, then this operation will have no effect.

replace
void replace(in Glyph g);

Put the given glyph in place of the glyph at this offset. The replace operation is equivalent to an atomic transaction that removes the old glyph and inserts the new one.

visit_trail
void visit_trail(in GlyphTraversal t);

Notify the composite that the desired size of the glyph at this offset has changed.

GlyphTraversal

interface GlyphTraversal : FrescoObject

The GlyphTraversal interface defines the mechanism for visiting a collection of glyphs. A traversal is passed to a glyph's traverse operation and maintains common information as well as the stack of information associated with each level of the traversal.

allocation, bounds, origin, span, is_visible
Region allocation();
boolean bounds(out Vertex lower, out Vertex upper, out Vertex origin);
boolean allocation_is_visible();

These operations return the current allocation region or information about it. A traversal may not define an allocation, in which case a nil region is returned. If the allocation region is defined, the bounds operation retrieves the corners and origin and returns true. Otherwise, bounds just returns false without modifying its parameters.

The allocation_is_visible operation returns true if the allocation is defined and is visible in the current drawing state (see Painter::is_visible).

clear
void clear();

Remove the picked reference to hit objects, potentially releasing the associated traversals.

current_display, current_screen
Display current_display();
Screen current_screen();

These operations return the display or screen where the traversal is rooted.

current_glyph, current_offset, current_viewer
Glyph current_glyph();
GlyphOffset current_offset();
Viewer current_viewer();

Return the information for the current element along the trail.

current_painter

attribute Painter current_painter;

This attribute defines the drawing state for glyphs during the traversal.

current_transform

Transform current_transform();

Return the cumulative transformation matrix for the current glyph in the traversal.

damaged

Damage damaged();

Return the damage area that glyphs in the traversal may use to cause a screen update.

forward, backward

boolean forward();
boolean backward();

These operations move the current glyph along the current trail. The forward operation moves the current glyph away from the viewer; the backward operation moves toward the viewer. Both operations return true if successful at moving the current glyph.

hit

void hit();

Add the current trail to the list of picked trails during a pick traversal.

hit_info

attribute long hit_info;

Associated with a hit object may be an additional integer containing arbitrary information. For example, a character object would use this information to indicate whether pick occurred on the left or right of the character.

op, swap_op

Operation op();
Operation swap_op(in Operation op);

Both op and swap_op return the current traversal operation. The swap_op method additionally changes the operation to the given operation.

Operation

enum Operation { draw, pick_top, pick_any, pick_all, other };

An operation distinguishes what kind of traversal is being performed. The common traversals are draw and three kinds of picking, which should result in calls to a draw or pick glyph operation on each leaf during a traversal. The “other” operation will call the traversal's visit operation on each leaf. The kind of pick specifies the desired result: pick_top indicates that the caller only wishes to find the top-most object that is hit, pick_any returns when any object is hit (possibly avoiding further traversal), and pick_all finds all hit objects.

picked

GlyphTraversal picked();

Return the traversal associated with the most-recently hit object.

trail

GlyphTraversal trail();

Copy the current traversal state information. The current trail, transformation, and allocation is copied, but graphics state is not. Thus, the new trail may be suitable for certain update or picking operations, but is not usable for use in drawing.

traverse_child

void traverse_child(in GlyphOffset o, in Region allocation);

Continue traversing down to the given child glyph defined by the given offset. If the given region is nil then the parent's allocation will be used for the child.

visit

void visit();

Call the traversal-specific operation on the current glyph. This operation should be called on a leaf glyph (or a composite that wants to do more than traverse its children) when the traversal operation is other.

Option

struct Option

An option describes a possible command-line argument in terms of a name that matches the argument string, a path that defines the root-style attribute to set, a format, and an optional implicit value. If the format is “implicit” then the attribute will be set to the implicit value. If the format is “value” then the attribute will be set to the next argument on the command-line. If the format is “path_value” then the next argument is a string containing the attribute path and value separated by a colon (“:”).

Painter

interface Painter : FrescoObject

The painter interface provides a very simple set of operations to perform resolution-independent drawing. The intent is to provide enough functionality to implement common two-dimensional text and structured graphics objects, but the operations should be considered neither complete nor particularly convenient. Some glyphs or viewers will need to use more sophisticated graphics libraries, such as for 3D graphics or image blending. It is important that a painter implementation be able to use such graphics libraries directly and thus allow mixing of painter-based objects and objects that make direct calls to another graphics library.

A painter object represents the state necessary to perform drawing. If an application wishes to have multiple threads drawing at the same time, then each thread should have a separate painter. Unlike X, where a graphics context may be used in different windows, a painter is bound to one and only one output area (either a portion of the screen or a device for printing).

begin_path, move_to, line_to, curve_to, close_path

void begin_path();
void move_to(in Coord x, in Coord y);
void line_to(in Coord x, in Coord y);
void curve_to(
in Coord x, in Coord y, in Coord x1, in Coord y1,
in Coord x2, in Coord y2
);
void close_path();

These operations define a path for drawing as in PostScript. The move_to operation sets the current position of the path, line_to extends the path to a new position along a straight line, and curve_to extends the path along a Bezier curve.

As an example of the drawing operations, the following code draws a filled triangle with corners (x1,y1), (x2,y2) and (x3,y3):

```
painter->begin_path();
painter->move_to(x1, y1);
painter->line_to(x2, y2);
painter->line_to(x3, y3);
painter->close_path();
painter->fill();
```

character

void character(in CharCode ch, in Coord width, in Coord x, in Coord y);

Draw the specified character from the given font in a given color, aligning the origin of the character at the given position. The width parameter is merely a hint as to the width of the given character in the current font. A painter may use the hint to buffer several characters in a single drawing request, doing so without explicitly computing the character's width.

clip, clip_rect, push_clipping, pop_clipping

void clip();
void clip_rect(in Coord x0, in Coord y0, in Coord x1, in Coord y1);
void push_clipping();
void pop_clipping();

The clip operation restricts subsequent drawing to be inside the current path. The clip_rect operation restricts drawing within the given rectangle. Clipping is cumulative; that is, two consecutive clip operations will result in a clipping region that is the intersection of the paths specified by the two requests. The push_clipping and pop_clipping operations can be used to save and restore the current clipping region.

comment

void comment(in CharString s);

This operation may write an annotation in the stream of output commands. As such, it has no effect on the actual image generated by the output commands.

current_brush, current_color, current_font, current_pattern

attribute Brush current_brush;
attribute Color current_color;
attribute Font current_font;
attribute Pattern current_pattern;

These attributes are used by the drawing operations. It is the responsibility of objects that perform drawing to set these attributes explicitly, or save and restore them if desired.

current_matrix, push_matrix, pop_matrix, premultiply

attribute Transform current_matrix;
void push_matrix();
void pop_matrix();
void premultiply(in Transform t);

The current_matrix attribute defines the current transformation matrix applied to coordinates when drawing. The push_matrix and pop_matrix operations can be used to save and restore the matrix. The premultiply operation modifies the current matrix to contain the result of multiplying it by the given transformation matrix.

image, stencil

void image(in Raster r, in Coord x, in Coord y);
void stencil(in Raster r, in Coord x, in Coord y);

The image operation draws the given raster with its origin at the given position. The stencil operation draws the current color through the mask defined by the raster, aligning its origin with the given position.

is_visible, visible

boolean is_visible(in Region r);
Region visible();

The is_visible operation returns true if any portion of the given region is within the drawing area defined by the clipping region. The visible operation returns the current clip region.

line, rect, fill_rect

void line(in Coord x0, in Coord y0, in Coord x1, in Coord y1);
void rect(in Coord x0, in Coord y0, in Coord x1, in Coord y1);
void fill_rect(in Coord x0, in Coord y0, in Coord x1, in Coord y1);

These operations are short-hand for drawing common paths. The line and rect operations stroke a line segment and rectangle, respectively. The fill_rect operation fills a rectangle path.

page_number

void page_number(in CharString s);

This operation generates page number information where meaningful, such as when generating PostScript.

stroke, fill

void stroke();
void fill();

The stroke operation draws the current color along the current path using the current brush. The fill operation draws the current color inside the current path.

to_coord, to_pixels, to_pixels_coord

Coord to_coord(in PixelCoord p);
PixelCoord to_pixels(in Coord c);
Coord to_pixels_coord(in Coord c);

These operations convert between pixels and coordinates on the painter's screen. They are equivalent to the corresponding screen operations and are provided for convenience.

Pattern

interface Pattern : FrescoObject

A pattern describes a simple way to fill an area. Pixels that correspond to zeros in the pattern are not drawn; pixels that correspond to ones are drawn in the current color.

equal, hash

boolean equal(in Pattern p);
unsigned long hash();

The equal operation tests whether two patterns objects refer to the same information. The hash operation will return the same value for two equal patterns, though there is also some chance it will return the same value for two unequal patterns.

stipple

void stipple(out DrawingKit::Data32 d);

Retrieve the stipple data associated with this pattern.

PixelCoord

typedef long PixelCoord

The PixelCoord type represents a position or distance in units of screen pixels. PixelCoord values are inherently resolution-dependent and generally should be avoided. However, some user interfaces need to guarantee that they produce output that is precisely balanced in terms of pixels. Coord values are resolution-independent, but one cannot rely on precise output at low resolutions. Coordinates are normally rounded-off to guarantee that objects that abut in coordinates will abut on the screen.

This approach introduces the possibility that the same Coord value might be converted to two different PixelCoord values. To avoid such an occurrence, an application object should convert the coordinate value to a PixelCoord and then back to a Coord before using it.

Raster

interface Raster : FrescoObject

A raster is a two-dimensional array of primitive elements, which are either colors that define a color image or boolean values that define a bitmap.

Element

struct Element {
boolean on;
Color pixel;
float blend;
};

A raster element is defined by a boolean value, a color, and a blending factor. The boolean value is synonymous with a blending factor of zero (if false) or one (if true). This value provides a convenient way to manipulate bitmaps. The blending value defines a factor for alpha blending. The value is between zero and one inclusive, where zero means nothing is blended and one means everything is blended.

equal, hash

boolean equal(in Raster r);
unsigned long hash();

The equal operation tests whether two raster objects refer to the same image or bitmap data. The hash operation will return the same value for two equal rasters, though there is also some chance it will return the same value for two unequal rasters.

Index

typedef long Index;

The Index type defines the range of values for indices into the array of elements.

origin_x, origin_y

Index origin_x();
Index origin_y();

Return the origin of the raster.

peek, poke

void peek(in Index row, in Index column, out Element e);
void poke(in Index row, in Index column, in Element e);

The peek operation returns the element at a given position in the array; the poke operation sets the element. The implementation of these operations should be cheap, which in the case of poke means that several operations may be deferred until the raster is next displayed.

rows, columns

Index rows();
Index columns();

Return the number of rows and columns in the raster.

scale

attribute Coord scale;

Define the mapping between raster indices and coordinates. If the scale is zero (the default value), then one index will correspond to one pixel on the output device. Otherwise, the scale defines the size of a raster element in resolution-independent units.

Region

interface Region : FrescoObject

A region is an area or volume within an object's coordinate space. A region may be described by a surface, curve, path, or other geometric means. Regions define an origin as a point of reference (the origin is not required to be contained inside the region). A region implementation may choose only to support a bounding box representation.

Regions are used for a variety of purposes, including the allocation an object receives as part of layout, the extension that an object covers when drawn, the portion of a window that needs to be updated, and the current clipping area. Some regions, such as the update area in a window, are 2-dimensional and aligned to the xy-plane. We refer to these regions as “xy-regions” and consider them to have a z-coordinate with value zero and no thickness.

apply_transform

void apply_transform(in Transform t);

Set this region to be the region that results from the given transformation applied to the original region. Because a region may be implemented using bounding boxes, transformations are not necessarily invertible. That is, transforming a region first by a matrix and then by the inverse of the matrix may yield a larger (but not smaller) region that the original.

BoundingSpan

```
struct BoundingSpan {
    Coord begin, end;
    Coord origin, length;
    Alignment align;
};
```

A bounding span is a one-dimensional coordinate range with an origin value as a reference. The begin, end, and align fields define a bounding span. The length is always equal to end minus begin, and the origin is always equal to begin plus length times align.

bounds, origin, span

```
void bounds(out Vertex lower, out Vertex upper);
void origin(out Vertex v);
void span(in Axis a, out Region::BoundingSpan s);
```

Return information about the region. The bounds operation returns two vertices that contain the lower and upper coordinate values, respectively. The origin operation returns the region's origin. The span operation returns the span along a particular axis.

contains, contains_plane, intersects

```
boolean contains(in Vertex v);
boolean contains_plane(in Vertex v, in Axis a);
boolean intersects(in Region r);
```

Determine is a region intersects a point or another region. The contains_plane operation only compares coordinates *not* in the given axis. For example, to test if a given point is in the xy-plane (therefore ignoring its z-coordinate), one would pass Z_axis as the second parameter.

copy

void copy(in Region r);

Copy the given region into this region. This operation is equivalent to setting this region to the empty region and then calling merge_union on the given region.

merge_intersect, merge_union

void merge_intersect(in Region r);
void merge_union(in Region r);

Modify a region to merge the values of another region. The merge_intersect operation sets the region to be the intersection of itself with the given region; merge_union sets the region to cover both regions. For both operations, the origin of the region is unchanged.

subtract

void subtract(in Region r);

Set this region to be the difference between itself and the intersection of itself and the given region. The origin of the region is unchanged.

Screen

interface Screen : FrescoObject

A screen represents a physical array of pixels.

display

Display screen_display();

Return the display that contains the screen.

dpi

attribute Coord dpi;

This attribute contains the resolution, or “dots-per-inch” (dpi) of the screen. Initially, the value is the hardware resolution. It can be changed to a different value for use in the conversion to and from pixel coordinates.

move_pointer

void move_pointer(in Coord x, in Coord y);

Change the current position of the input pointer to the given position relative to the root window of the screen. This operation can surprise users and generally should be avoided.

to_pixels, to_coord, to_pixels_coord

PixelCoord to_pixels(in Coord c);
Coord to_coord(in PixelCoord p);
Coord to_pixels_coord(in Coord c);

These operations convert coordinates between points and pixels using the screen's resolution. The to_pixels_coord operation is equivalent to calling to_coord(to_pixels(c)), which effectively rounds the coordinate out to the nearest pixel boundary.

width, height

Coord width();
Coord height();

Return the screen dimensions in resolution-independent coordinates.

Style

interface Style : FrescoObject

A style is similar to an environment in a text formatting system such as Scribe. Styles may be nested hierarchically, and attributes defined in an outer context are visible in an inner context if not otherwise defined. A style may have a name, a list of aliases for wildcard-matching, a collection of attributes (name-value pairs), a collection of nested styles, and a parent style.

Attribute names may contain "*" characters to specify wildcard matching. A name of the form A*B will match an attribute B in a nested style named A. Wildcard names also may begin with a "*," which matches in any descendant style. Thus, *A*B will match an attribute B in any descendant style named A. Because attributes are inherited, specifying the wildcard name *B is identical to specifying the normal attribute B.

In addition to a name, styles may have a list of associated aliases. Wildcard matches search using a style's name first, then search using the style's aliases in the reverse order in which they are defined. Aliases are typically used for indicating a subtype relationship and allowing styles to inherit attributes specified for a supertype.

For example, suppose the root style defines the following attributes:

```
*Mover*autorepeat: off
*UpMover*autorepeat: on
```

Consider descendant styles S and T: S's aliases are UpMover and Mover; T's aliases are DownMover and Mover. The name "autorepeat" will be resolved to the value "on" for S and the value "off" for T.

The wildcard matching algorithm is intended to be compatible with the X resource manager (Xrm) and support the same user customization functionality. Wildcard attributes typically are defined only on the root style, as loaded from the window system, application defaults files, or command-line arguments.

alias

void alias(in CharString s);

Add an alias to the list for the style.

bind

StyleValue bind(in CharString name);

If the given name is already defined in this style, then return the associated attribute. Otherwise, create and return a new attribute in the style. The priority of the new attribute will be initialized to zero.

is_on

boolean is_on(in CharString name);

Determine whether the given name is defined as an attribute and whether the value is set to "on" or "true." This operation is short-hand for resolving the name to an attribute and calling StyleValue::is_on on the result.

link_child, unlink_child

Tag link_child(in Style child);
void unlink_child(in Tag link_tag);

The link_child operation adds the given style to the list of nested styles of this style and returns a tag to be passed when the child is to be removed from the list.

link_parent, unlink_parent

void link_parent(in Style parent);
void unlink_parent();

The link_parent operation sets the parent of a style for the purposes of inheriting attributes. The unlink operation unsets the parent so that attributes are no longer inherited.

lock, unlock

void lock();
void unlock();

The lock operation acquires exclusive access to the style until a subsequent call to unlock.

match

long match(in CharString name);

Check to see if a given name matches the style's name or any of its aliases. This operation returns zero if there is no match, one if the name matches, or two plus the alias number in the case of an alias match. A smaller index value indicates a “closer” match if the aliases represent an ancestry of superclasses.

merge

void merge(in Style s);

Copy all the attributes, aliases, and nested styles in the given style into this style.

name

attribute CharString name;

The name attribute defines the style's name for the purposes of wildcard matching. In Xrm terms, this attribute is the “instance name.”

new_style

Style new_style();

Return a newly-created style.

parent

Style parent_style();

Return the style's parent style, if any.

resolve

StyleValue resolve(in CharString name);

Search for an attribute with the given name (including inherited attributes), returning the associated attribute if found or nil otherwise. This operation also searches wildcard attributes using the name and alias of this style and, in the case of inherited attributes, the names and aliases of all the styles between this style and the style where the inherited attribute is defined.

resolve_wildcard

StyleValue resolve_wildcard(in CharString name, in Style start);

Look for a wildcard match in this style, using the given style as the starting point for name and alias comparisons. This operation is primarily for the style implementation to search its ancestors; application objects usually need not call this operation directly.

unbind
void unbind(in CharString name);

Remove the attribute with the given name, if any is defined on this style.

visit_aliases, visit_attributes, visit_styles
void visit_aliases(in StyleVisitor v);
void visit_attributes(in StyleVisitor v);
void visit_styles(in StyleVisitor v);

These operations iterate through the aliases, attributes, and nested styles in this style.

StyleValue

interface StyleValue : FrescoObject

An attribute is a (name,value) pair, where the value may be written or read in one of several forms. The value access functions come in pairs, one to read the value and one to write the value. When an attribute is written, it immediately notifies any observers.

is_on
boolean is_on();

Return true if the value can be interpreted as a boolean; that is if value is the string “on” or the string “true”. The string comparision is case-insensitive.

lock, unlock
void lock();
void unlock();

Acquire or release exclusive access to the attribute. It is the responsibility of attribute clients to ensure mutual exclusion during updates.

name
CharString name();

Return the attribute's name. Because the name cannot be changed, this operation can be called without locking the attribute.

priority
attribute long priority;

An attribute priority is simply a hint to attribute users as to whether they should override an already-defined value. The value is not used by the attribute in any operations. Typically, a style manager will check the priority and only update the attribute's value if the existing priority is lower than the priority of the new value. Setting an attribute's priority does not cause it to notify its observers.

read_boolean, write_boolean
boolean read_boolean(out boolean b);
void write_boolean(in boolean b);

Interpret the value as a boolean that is true or false. If the value is actually a string, this operation returns true if the string is “on” or “true.” The string comparison is case-insensitive.

read_coord, write_coord
boolean read_coord(out Coord c);
void write_coord(in Coord c);

If the value is a string, then interpret the string as a real number (see the read_real operation) followed by an optional unit specification. The remaining of the string can be “in” for inches, “cm” for centimeters, “mm” for millimeters, or “pt” for the width in points. For example, the value “10mm” converts to 28.3.

read_integer, write_integer
boolean read_integer(out long i);
void write_integer(in long i);

Interpret the value as a 32-bit (optionally signed) integer.

read_real, write_real
boolean read_real(out double d);
void write_real(in double d);

If the value is a string, interpret the string as a floating point number. The number has an optional leading plus or minus, zero or more digits to the left and right of the decimal point (the point itself is optional), and an optional exponent specification.

read_string, write_string
boolean read_string(out CharString s);
void write_string(in CharString s);

Interpret the attribute value as a string.

read_value, write_value
boolean read_value(out FrescoObject s);
void write_value(in FrescoObject s);

Interpret the attribute value in general as a reference to an object. Readers will typically need to try to narrow the result to the type that they expect.

uninitialized
boolean uninitialized();

Return whether the attribute value has been defined yet or not. This test allows attribute update based on priority to handle the case when the attribute is first bound and the priority has not been set.

StyleVisitor

interface StyleVisitor : FrescoObject

A style visitor is essentially a callback for iterating over the elements of a style. The visitor can be used to access the aliases, attributes, and style children of a particular style.

visit_alias, visit_attribute, visit_style
boolean visit_alias(in CharString name);
boolean visit_attribute(in StyleValue a);
boolean visit_style(in Style s);

Each of the visit operations returns a boolean value indicating whether the iteration should continue or not.

Tag

typedef unsigned long Tag

A tag represents a value used for associative access. Typically, tags are returned when an association is created and passed to remove or otherwise refer to the association. Tags are always *relative* to the object that creates them; they are not unique ids.

Transform

interface Transform : FrescoObject

A transform represents a (logically) 4x4 matrix for use in translating coordinates. A 2-dimensional implementation may store and manipulate a 3x2 matrix rather than a full 4x4 matrix.

equal, identity, det_is_zero

boolean equal(in Transform t);
boolean identity();
boolean det_is_zero();

The equal operation tests if a transformation matrix is the same as a given transform within a standard coordinate tolerance. The identity operation tests if the matrix is equal to an identity matrix (also with the standard tolerance). The det_is_zero operation tests if the determinant of the matrix is zero within the standard tolerance.

invert

void invert();

Set the matrix to its inverse.

load

void load(in Transform t);

Copy the matrix data from the given transform.

load_identity

void load_identity();

Set this transform's matrix to the identity matrix.

load_matrix

void load_matrix(in Matrix m);

Copy the given matrix data into this transform.

Matrix

typedef Coord Matrix[4][4];

The Matrix type defines the data that may be contained in a transform.

premultiply, postmultiply

void premultiply(in Transform t);
void postmultiply(in Transform t);

These operations set the matrix to be the result of multiplying the matrix and the given transform. Two operations are necessary because matrix multiplication is not necessarily

communative. Premultiply means the current matrix is on the left-hand side of the multiplication, postmultiply means the current matrix is on the right.

scale, rotate, translate

void scale(in Vertex v);
void rotate(in float angle, in Axis a);
void translate(in Vertex v);

Modify the matrix to perform coordinate scaling, rotation, and translation. The rotation angle is given in degrees. A 2-dimensional implementation only implements rotate about the z-axis.

store_matrix

void store_matrix(out Matrix m);

Store this transform's matrix to the given matrix data.

transform_vertex, inverse_transform_vertex

void transform_vertex(inout Vertex v);
void inverse_transform_vertex(inout Vertex v);

Multiply the given vertex by the transformation matrix to compute transformed or inverse-transformed coordinates.

Vertex

struct Vertex { Coord x, y, z; }

A vertex is a 3-dimensional point. A Fresco implementation may only implement 2d operations and therefore will ignore the z-field of vertices.

Window

interface Window : FrescoObject

A window is supported by the underlying window system and can be manipulated by the user through a (possibly-separate) window manager.

configure, get_configuration

void configure(in Window::Placement p);
void get_configuration(in boolean position, out Window::Placement p);

The configure operation sets the desired placement for a window, the get_configuration operation returns the actual placement. If the position flag is true, then get_configuration will retrieve both the size and position of the window; otherwise, only the size fields of the placement structure will be set.

configure_notify, move_notify

void configure_notify(in Coord width, in Coord height);
void move_notify(in Coord left, in Coord bottom);

Notify the window that its size or position has changed. These operations are intended for use by the display object in response to events generated by window management, and need not be called directly by applications.

default_cursor, cursor_foreground, cursor_background

attribute Cursor default_cursor;
attribute Color cursor_foreground;
attribute Color cursor_background;

These attributes define the default cursor and cursor colors to use when the pointer is inside the window.

double_buffered

attribute boolean double_buffered;

This attribute determines whether drawing to a window is double-buffered or not.

geometry

attribute CharString geometry;

Geometry specifications are strings of the form "WxH+X+Y" where W is the width, H the height, X the left corner, and Y the top corner of the window. Either the position or the size may be omitted, and the position can use "-" instead of "+" to denote a distance from the opposite side of the screen to the opposite side of the window. For example, a -X value specifies the right side of the window should be a distance X from the right edge of the screen.

grab_pointer, ungrab_pointer

void grab_pointer(in Cursor c);
void ungrab_pointer();

The grab_pointer operation takes control of input events for the entire display until a subsequent call to ungrab_pointer. The cursor parameter, unless nil, is used during the grab when the pointer is outside the window.

icon, iconic

attribute Window icon;
attribute boolean iconic;

If the icon attribute is specified then a window manager will normally map the icon window when this window is unmapped. If the iconic flag is true, then calling Window::map will actually map its icon window instead.

map, unmap, is_mapped

void map();
void unmap();
boolean is_mapped();

The map operation requests that the window be mapped onto its screen and become visible. The unmap operations requests the window be unmapped and no longer be visible. The is_mapped operation returns whether the window is currently mapped. Because the map and unmap operations may be asynchronous, one cannot assume that a window will be mapped or unmapped immediately after the corresponding call.

map_notify, unmap_notify

void map_notify();
void unmap_notify();

Notify a window that is has been mapped or unmapped. These operations are intended for use by the display object in response to events generated by window management, and need not be called directly by applications.

Placement

```
struct Placement {
    Coord x, y, width, height;
    Alignment align_x, align_y;
};
```

The placement type describes a window position. The x and y fields specify the screen position for the window. The align_x and align_y fields are fractions that indicate where the window should appear relative to the x and y coordinates. For example, an alignment of 0.0,1.0 means the position specifies the upper-left corner of the window. An alignment of 0.5,0.5 means the coordinates specify the center of the window. Unless specified, the alignment will be 0.0,0.0, meaning the coordinates specify the lower-left corner of the window.

raise, lower

```
void raise();
void lower();
```

These operations request a change in the stacking order of this window with respect to other windows. The raise operation asks that this window be the highest of all windows at the same level; the lower operation asks to be the lowest.

redraw

```
void redraw(
    in Coord left, in Coord bottom, in Coord width, in Coord height
);
```

Damage a rectangular region of a window. This operation is intended for use by the display object in response to exposure events. Application objects should perform operations on a traversal's damage object rather than calling redraw directly.

repair

```
void repair();
```

Perform a draw traversal in the window beginning with the main viewer. This operation is normally called by the display object and need not be called from an application.

title

```
attribute CharString title;
```

This attribute defines a string that may be used by a window manager in a title bar above a window.

window_screen

```
Screen window_screen();
```

Return the screen with which the window is associated.

window_style

```
Style window_style();
```

Return the style for the window.

xor_pixel

```
attribute long xor_pixel;
```

This attribute defines the pixel value to use when performing xor drawing operations. Using xor on a color display is problematic and should generally be avoided.

Index

A

Action 19
Alignment 19
AllocationInfo 31
AllocationInfoSeq 31
Axis 19

B

BoundingSpan 42
Brush 19

C

CharCode 20
CharData 20
CharString 20
Color 21
concurrency 10
Coord 22
Cursor 22

D

Damage 22
Data 20
Data32 25
Data8 25
deadlock 10
Display 23
distribution 10
DrawingKit 25

E

editing 13
Element 41
embedding 13
encapsulating 9

F

Font 27
Fresco 7, 28
Fresco_init 29
Fresco_open 29
Fresco_open_display 29
FrescoObject 30

G

Glyph 30
GlyphOffset 34
GlyphTraversal 35
granularity 10

H

HashValue 21

I

identifiers 10
IDL 7
Index 41
Info 27
Intensity 21
interface 8
InterViews 7

K

kit 9

L

logical composition 13

M

Matrix 48

O

object factory 9
object reference 9
OffsetSeq 32
OMG 7
Operation 36
operations 10
Option 37

P

Painter 37
parameters 10
Pattern 40
PixelCoord 40
Placement 51
presentation 13
pure virtual 8

R

race condition 10
Raster 40
Region 42
Requirement 33
Requisition 33
Return 26

Index (Cont.)

S

Screen 43
ScreenNumber 24
Style 44
StyleValue 46
StyleVisitor 47

T

Tag 48
thread 10
Transform 48

V

Vertex 49

W

Window 49