Fresco™ Sample Implementation Reference Manual

X Consortium Working Group Draft
Version 0.4

October 21, 1993

Table of Contents

Introduction

Fresco is an object-oriented application programming interface (API) for graphical user interfaces that covers functionality in Xlib and Xt and adds support for structured graphics and application embedding. Fresco covers a broad range of functionality—buttons, menus, rectangles, white space, text editors, and movie players can all be Fresco objects. An application may choose to use part of the Fresco functionality within another environment, such as the structured graphics capabilities from within Xt, or use Fresco as a complete environment.

This manual defines the objects and operations that make up Fresco. In this introduction, we present the basic concepts of Fresco and the notation we use to specify the API. The remainder of the manual is a collection of definitions for the Fresco object interfaces, data types, and external functions. The definitions are presented in alphabetical order. Each interface definition in turn contains a list of its member definitions in alphabetical order.

Fresco has following three fundamental characteristics that distinguish it from Xt and most other APIs for graphical user interfaces:

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

We give an overview of the OMG object model in Section 5.1. The resolution-independence of Fresco coordinates means that graphical objects have the same size and appearance regardless of the dots-per-inch of a particular screen or printer.

Graphical embedding is the composition mechanism for building hierarchies of user interface objects. Fresco combines the traditional areas of structured graphics and application embedding into a single unified user interface object model. This integration allows embedded applications to appear and be appropriately transformed within graphical scenes.

In the next section, we present the basic types that are pervasive throughout Fresco. In Section 2, we describe the Fresco API for basic window system services, such as mapping window to a screen, handling device events, and coordinating with a window manager. Section 3 presents an overview of the primitive graphics state objects such as colors and a painter object for displaying graphical objects. Section 4 discusses graphical embedding in detail. Finally, in Section 5 we present the notation used to define the Fresco API, including an overview of the OMG Interface Definition Language (IDL).

1. Basic types

The basic Fresco types provide common functionality such as 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.

All objects have a type that minimally can be used to convert safely from a more general type to a more specific one. Types are themselves objects, represented by the FrescoType interface.

The Fresco type has one instance that provides access to other 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++.

1.1 Memory management

Fresco does not require garbage collection of objects, though some implementations may assume such an environment. The FrescoObject interface, from which all other interfaces are descended, defines operations for managing 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.

Our convention is that communicating a shared object from one object to another requires the sender hold the reference before sending the reference and the receiver release when it no longer needs the reference. In a reference counting implementation, holding a reference means incrementing the count and releasing means decrementing the count.

This convention guarantees that an object will not be destroyed accidentally when passed as a parameter, allows return values to be used directly as parameters (without the use of an explicit temporary), and allows an operation to create and return a temporary object. Because the caller is always responsible for releasing the reference, a temporary return value will be reclaimed when the caller is finished with it. The one inconvenience of this approach is that one cannot cascade calls that return references. For example, “a->f()->g()” will not release the reference to the return value from a->f().

This convention can be a little surprising at first. In this example below, notice that a function must release its object reference parameter:

```
void f(ObjectRef obj) {
    // do some stuff
    Fresco::unref(obj);
}
```

Similarly, a caller must hold an object reference that it wishes to hold before passing it:

```
void A::g() {
    // assume that obj_ is a member variable
    Fresco::ref(obj_);
    f(obj_);
}
```

However, if a function merely passes an object reference on to another function, then the first function need not hold or release the reference because the second function will take care of that:

```
// assume f above
void p(ObjectRef obj) {
    // do some stuff
    f(obj);
    // do some more stuff
}
```

Of course, if the reference is passed several times, then it must be held for all but the last call:

```
void p(ObjectRef obj) {
    Fresco::ref(obj);
    f(obj);
    // do some stuff
    Fresco::ref(obj);
    f(obj);
    // do some more stuff
    f(obj); // last call
}
```

In certain cases, this convention is inconvenient and inefficient. If the parameter is known to be shared across a set of calls, then one may decide that the calls will use a different convention that does no reference management. This exception occurs for a drawing traversal where the same traversal state object is passed to every object. For such a traversal, it is unnecessary to perform reference management on the common parameter.

1.2 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.

1.3 Coordinates

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), 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 (e.g., on a different screen or a printer).

The Axis type specifies a dimension of cartesian coordinates. Many user interface operations, especially involving layout, can be parameterized by dimension. The Transform type defines a matrix for scaling, rotating, or translating coordinates. The Region type defines a portion of the possible coordinate space.

1.4 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.

1.5 Attribute management

Many objects contain modifiable state they wish to make editable, such as the current font in a text editor or the current color in a graphics editor. We refer to the items of an object's visible state as attributes.

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 attributes, such as fonts or colors. We can narrow to determine whether a style supports a particular set of attributes. 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 an attribute may require searching up the Style hierarchy, looking for a context that both defines the interface supporting the attribute and defines a value for the attribute. Alternatively, attributes could be cached and a notification sent when the original attribute is modified.

2. Windowing

Interfaces: Display, DisplayStyle, Screen, Window, WindowStyle

3. Drawing

Interfaces: Brush, Color, Font, Pattern, Raster, Painter, DrawingKit

4. 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 can edit the text in place within the context of a graphical transformation.

4.1 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.

4.1.1 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.

4.1.2 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, for example, could extend outside the text object's allocation.

4.1.3 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.

4.1.4 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.

4.2 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.

4.2.1 Separating the user interface from the data

The key conceptual requirement 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.

4.2.2 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. Figure 5.1 shows an example viewer-data structure for a graphics object embedded inside a text object. The rectangles represent data objects; the ellipses represent viewers.

4.2.3 Transcription

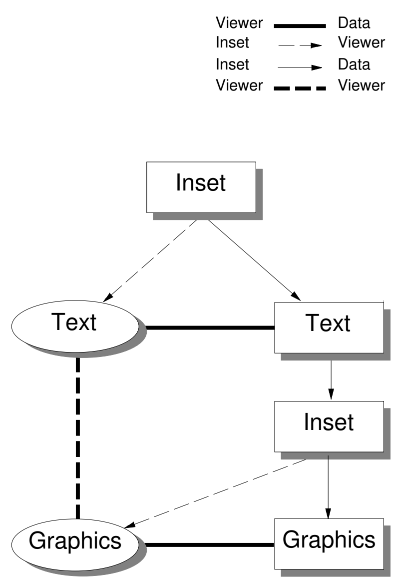
An object transcription is a persistent representation of the object in an externally-controlled format. The external definition of the format is what distinguishes transcription from persistence or object-oriented databases. In a persistent store or database, the system designer can choose one format 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.

4.2.4 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. 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 bound to several languages. OMG has defined a standard binding to C, and efforts are underway to define bindings 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.

An IDL object type is called an *interface.* Fresco provides two kinds of interfaces: *protocols* and *kits.* Both define a set of operations, but the operations in a protocol perform some function on an object, such as drawing or handling input. The operations on a kit create objects, such as push buttons or text editors. The idea of a kit is also sometimes referred to as an "object factory." Using kits hides the details of object construction, including subclass-instance trade-offs made by the implementation and location decisions in a distributed environment.

5.1 IDL overview

An IDL object type is called an *interface*. The definition of a type 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 a possible definition of a rectangle object in IDL as follows:

```
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);
};
```

This interface defines eight 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. Note that we do not propose this interface as the only or best interface for a rectangle; we simply use this example for convenience.

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 in-line 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, in this example one could not access a rectangle remotely using this class.

A more flexible C++ definition of this rectangle class is possible 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. The OMG specifies standard *language mappings* that translators can use to generate implementations that are compatible with the OMG object model.

5.1.1 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 a pointer to the pure virtual class. In the example above, the operation would be declared as

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

The rectangle implementation subclass must support both the rectangle operations and the ability to pass a rectangle reference to a remote operation.

5.1.2 Remote objects

Along with the translation of IDL to C++ classes, we assume that system support for distributed objects includes the generation of stub operations for marshalling and unmarshalling parameters into a message buffer, the network transmission of messages, a name service for finding objects, and automatic tracking of object location. In addition, we assume that specific object types may provide remote implementations that are tuned for their particular protocol. All but the last assumption are met by OMG specifications, implementations of which will be provided by CORBA-compliant system vendors. The last is the responsibility of the object implementor (us).

As users of distributed objects, we can simply define the Fresco objects with IDL, generate the pure virtual classes and stubs, and then provide C++ implementation subclasses. A call on an object reference to a remote object will look syntactically identical to a call on a local object. Figure 0.1 shows how CORBA implements remote objects. A client program holds an object reference, which in C++ is a pointer to an implementation class called a *stub.* The stub uses the transport mechanism to send a call to the implementation *skeleton*, which performs the appropriate operation on the object implementation.

5.1.3 Local Objects

Although we want the power to access remote objects, most accesses will be to local objects. If the object reference is to an object in the same address space as the caller, then we can simply call the operation directly using the normal C++ virtual function dispatching mechanism. The only issue is where the object’s data resides. If the data is allocated with the object, then migrating the object from being local to being remote is more difficult than if the data access is indirect.

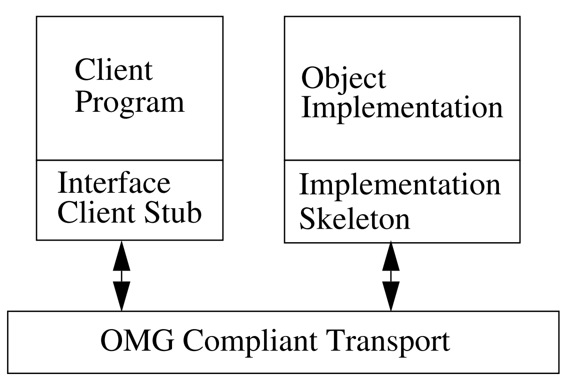


Figure 0.1 Implementation of remote objects in CORBA.

For local objects, the cost of using IDL is equivalent in C++ to using virtual functions for all calls and having an extra level of indirection to access data. Experience with InterViews has been that these conventions are good to follow anyway. Virtual functions allow extensibility, and indirect access to data prevents recompilation of code that uses a class when the class representation changes.

5.1.4 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.

Experience with InterViews has shown that factories offer advantages over C++ constructors even when creating objects in a single address space. A factory 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.

5.1.5 IDL usage conventions

IDL defines a number of features not directly available in C++, including additional builtin types, discriminated unions, modules, and contexts. Of these features, Fresco *only* relies on the existence of boolean, octet, and sequence types. We use the type String instead of the IDL type string so as to allow for better support for international text.

We assume that the 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 binding or implementation will handle this problem.

5.1.6 Notational conventions

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.

5.2 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).

5.3 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. In particular, the event loop, screen redraw, and application operations could run in three separate threads by default. The consequence of this use of 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.

5.4 Internationalization

Interfaces: String, CharCode

6. Summary

Sample Implementation

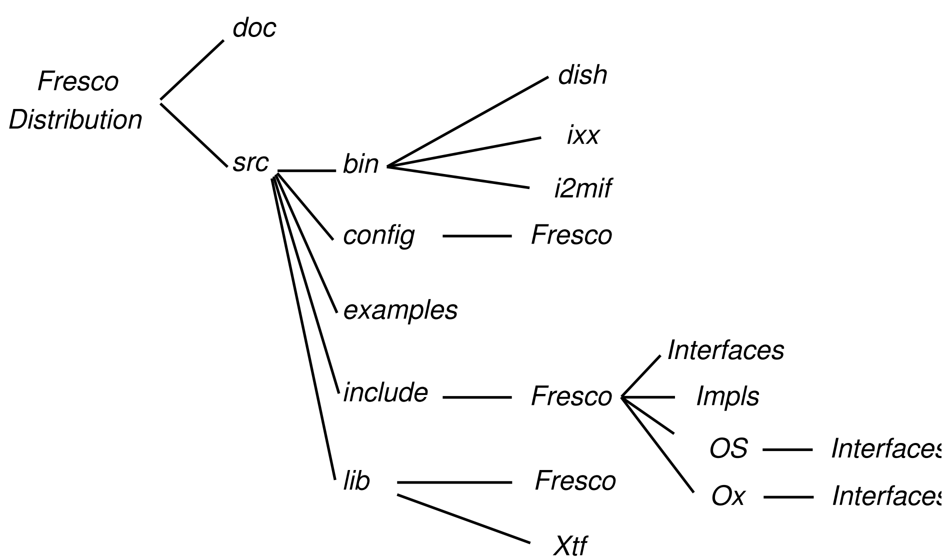
This manual contains information describing both the Fresco specification and the sample implementation provided by the X Consortium. The distinction between specification and implementation is important because one may encounter difficulties porting code to other Fresco implementations if the code relies on non-standard parts of the sample implementation.

In this chapter, we describe the directory organization and platform requirements of the sample implementation. We also describe two tools that assist in building the sample library and documentation. The interface translator, Ix, produces C++ header files from interface definitions and also can edit implementation source files to reflect a change in an interface. The MIF generator, I2mif, produces FrameMaker input files from the comments in the interface definitions.

The remainder of this manual contains the definitions of Fresco types and functions. Definitions for types specific to the sample implementation are included in the manual along with standard definitions. We denote non-standard definitions with a trailing asterisk (“*”) after the name. For example, ThreadKit* is the kit in the implementation for creating threads and locks, which is not part of the Fresco specification but is an important part of the sample implementation.

1. Directory organization

The Fresco distribution contains the directory structure shown in the figure below. The doc subdirectory contains the FrameMaker source for this manual, split into a file for each major section of the manual. The “book.fm” file contains this implementation manual; “spec.fm” only contains the sections relevant to the standard Fresco specification.



The src directory contains subdirectories for commands (bin), configuration files, example programs, include files, and libraries. The current commands are the interface translator (Ix), MIF generator (I2mif), and TCL-based dynamic invocation shell (Dish). The bin directory also contains a scripts subdirectory with various shell scripts. The config, include, and lib directories contain a subdirectory named Fresco to distinguish Fresco-specific source from the potential for other source that might be included in the sample implementation. The include directory is further split into a subdirectory for IDL interfaces and a subdirectory for common implementation classes. All of the files in the Interfaces subdirectory have a corresponding .h file in the parent directory; these .h files are generated automatically by Ix.

2. Platform requirements

The sample implementation requires C++ version 3.0 or later. The implementation currently does not use templates, but does require nested types. In particular, Ix generates struct declarations inside of classes that we refer to as *class-name::struct-name*. The same nesting is used for enumerations and typedefs.

Fresco can run on any X platform, but provides more functionality on platforms that 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.

The current implementation has been built on the following platforms:

- SGI IRIX 5.1, SGI C++ 3.2
- Solaris 2.2, SPARCompiler C++ 3.0.1
- DEC Alpha OSF/1 with CXX, GNU cpp.

3. Ix – Interface translator

The sample implementation includes a translator called Ix that reads interfaces from standard input and writes corresponding C++ class definitions to standard output. Ix recognizes but currently does not generate C++ definitions for IDL modules, constants, exceptions, contexts, or the type "any." In addition to being able to generate C++ classes corresponding to the interface definitions, Ix can "filter" C++ source files to assist with the coding of implementation classes. Normally, one need not run Ix manually, as it is called by the MakeHeader and EditImpl rules defined by the configuration files.

3.1 Command-line options

-f[ile] file
Only generate C++ class definitions for interfaces defined in the specified file, omitting any interfaces defined in #include'd interface files. This option should be used in conjunction with the -include option so that the generated file can access the C++ headers corresponding to the #include'd interface files.

-filter file
Instead of generating C++ class definitions, copy the file argument to standard output, adding additional code based on annotations embedded in the file and the interface definitions. The subsection below on filtering describes the possible annotations.

-i[nclude] name
Insert a #include with the given name at the beginning of the generated C++ output. This option cannot be used with -filter. Output begins with a comment warning that the source has been automatically generated, followed by a #ifndef and #define so that the source will not be

included more than once. The #include directive follows the #define. A #endif is placed at the end of the output to match the #ifndef.

-m[etaclass] name
Use the given name for the class providing runtime type information on the classes generated from the interface definitions and select implementation classes.

-r[equest] name
Specify the name of the class for creating dynamic invocation requests.

-s[uperclass] name
Use the given name as the public base class for any C++ class generated from a top-level interface (that is, an interface not derived from any other interfaces). The one exception is if the name itself is also the name of a top-level interface.

-stubclass suffix
Use the given suffix to generate a stub class for every interface. Stub generation is still experimental in the current version of Ix.

-stubfile name
Generate stubs and type information in the file with the given name.

-serverfile name
Generate server-side skeleton code in the file with the given name.

-stubinclude name
Put a #include directive with the given name in the stub output file.

-serverinclude name
Put a #include directive with the given name in the server output file.

3.2 Generated C++ class definitions

The name of an interface is used as the name of the corresponding C++ class. A derived interface maps to a class that publicly inherits from the corresponding base class. By default inheritance is *not* virtual to avoid some deficiencies in current C++ compilers. This approach requires that the base class specified by the -superclass argument have no data members.

Unless stubs are generated, all member functions are generated as pure virtuals. The member functions must be defined in implementation classes that derive from the generated abstract base classes. No data members are defined in the generated classes.

For each generated class, Ix defines a type for object references. The name of the new type is formed by appending “Ref” to the class name. For example, an interface named Glyph defines the class Glyph and the type GlyphRef. An object reference is equivalent to a C++ pointer, though using the GlyphRef may be more portable than Glyph* because other interface translators may have a different representation.

Predefined IDL types such as boolean and long are simply copied as is. An Ix user must therefore ensure that the included C++ files properly define the primitive types.

Parameters and return values that are interface types are always mapped to object references (pointers). Except for structures and unions, in parameters correspond to by value parameters, out and inout parameters are passed as C++ ref parameters. An in struct parameter is passed as a constant ref, out and inout structures are passed as refs.

3.3 Filtering

Ix assists the coding of C++ implementations classes that descend from a generated C++ class by processing annotations in the C++ source. This approach allows the automatic updating of implementation classes when either the interface changes or the binding from interface to C++ changes.

Annotations assume the syntax of a specialized C++ comment starting with the string "//+" and continuing until the end of the source line. Class annotations simplify the task of writing C++ class definitions. Implementation annotations simplify writing the C++ function headers for an implementation. Below is each annotation syntax with a description of what ix does during filtering.

3.3.1 Class annotations

Definition annotations affect the class definitions of implementation classes. They are used to specify redefinition of methods of classes generated from interfaces and methods that are common to all implementations.

[replaced] //+ interface::op

Replace the code in the line containing the annotation with the member function definition for the specified op in the specified interface.

[replaced] //+ interface::attribute=[identifier]

Replace the code in the line containing the annotation with the member function definition for setting the specified attribute in the specified interface. If an identifier is specified, it is used as the parameter name for the new value.

[replaced] //+ interface::attribute?

Replace the code in the line containing the annotation with the member function definition for retrieving the specified attribute in the specified interface.

//+ interface::*
[replaced]
//+

Replace all lines between the annotation and a closing "//+" with the member functions for all the operations and attributes defined by the specified interface.

//+ interface::_
[replaced]
//+

Replace all lines between the annotation and a closing "//+" with the functions common to all implementations of the specified interface. This list might include, for example, the function to attempt to narrow a base class object to the specified interface type.

//+ derived : base[, base1...]
[replaced]
//+

Replace all lines between the annotation and a closing "//+" with the beginning segment of a class definition that is publicly derived from the specified base classes. This segment includes the class identifier and base specifiers, and some common methods.

3.3.2 Implementation annotations

Implementation annotations affect member function definitions. They are used to automatically generate the signature and argument names of operations redefined in the implementation class. They can also be used to generate default definitions for methods of classes generated from interfaces, common implementation methods, and method redirections.

//+ class(interface::op) {
[replaced] {
: Replace input from the line after the annotation up to but not including a left brace (“{“) with the signature of the member function for the specified operation in the specified interface. The member function is qualified by the specified implementation class name. If left brace is followed immediately by a space and a right brace (“}”), then parameter names are not written as part of the generated signature (avoiding compiler warnings for unimplemented or otherwise empty functions).

//+ class(interface::attribute=identifier)
[replaced] {
: Replace input from the line after the annotation up to but not including a left brace (“{“) with the signature of the member function for setting the specified attribute in the specified interface. The member function is qualified by the specified implementation class name.

//+ class(interface::attribute?)
[replaced] {
: Replace input from the line after the annotation up to but not including a left brace (“{“) with the signature of the member function for retrieving the specified attribute in the specified interface. The member function is qualified by the specified implementation class name.

//+ interface
[replaced]
//+
: Replace the input after the annotation up to the closing “//+” with the member functions for the class generated from the specified interface. These functions include an empty constructor, destructor, copy constructor, and assignment operator.

//+ class(interface::_)
[replaced]
//+
: Replace the input after the annotation up to the closing “//+” with the member functions associated with the specified implementation class for the specified interface. These operations may include a function for narrowing a base class object to the specified interface type.

//+ class(interface::*receiver)
[replaced]
//+
: Replace the input after the annotation up to the closing “//+” with the member functions for all the operations and attributes defined by the specified interface. For each function, ix generates a body to call (and possibly return) the corresponding function on the specified receiver class. This feature effectively redirects calls to a base class and is useful when resolving ambiguities caused by multiple inheritance.

3.4 Running Ix under make

Instead of invoking ix directly, one normally invokes ix through the make utility that uses a make file partly generated from imake rules. Imake rules for ix either generate C++ headers from interface files or filter implementation files.

3.4.1 Header file generation

MakeHeader(header)
Generate a C++ header file in the current directory with the name header.h from the file Interfaces/header.idl. The generated file includes only the interfaces defined in header.idl, omitting interfaces found in any files #include'd by header.idl.

MakeHeaderFlags(name,flags)
Similar to MakeHeader, except the flags are passed to ix.

Running make using the resulting Makefile causes a header file to be regenerated whenever any of its depending interface files is modified.

3.4.2 Implementation file filtering

EditImpl(name,interfaces)
Filter the C++ implementation file name.ext by applying its annotations with respect to the definition in the space-separated list of interface files, where "ext" is the C++ file name suffix defined by the imake symbol CCSuffix.

EditImplHeader(name,interfaces)
Filter the C++ header file name.h by applying its annotations with respect to the definitions in the space-separated list of interface files.

Running "make all.sed" using the resulting Makefile in the directory containing the implementation header files or the object files compiled from the implementation files forces the refiltering of the implementation files. This behavior is desired because interface files typically change far less frequently than implementation files and different ix behavior (e.g. different options) may call for complete refiltering.

3.5 Example

Consider the following ix input, normally generated by the C preprocessor to handle #include directives:

```
# 1 "/a/b/example.idl"
interface B { void f(); };
```

The following output would be produced by running ix with the arguments "-include '"A.h"'" and "-superclass A":

```
/* DO NOT EDIT — Automatically generated from /a/b/example.idl */

#ifndef b_example_h
#define b_example_h

#include "A.h"

class B;
typedef B* BRef;

class B : public A {
protected:
    B();
    virtual ~B();
public:
    virtual void f() = 0;
private:
    B(const B&);
    void operator =(const B&);
};
```

4. I2mif – Interface to MIF generator

The sample implementation includes a program "I2mif" that produces FrameMaker Interchange Format (MIF) from the comments in the source include directory. I2mif reads standard input and produces the MIF to standard output. The Imakefile in the doc directory contains macros that call i2mif appropriately.

The source comments are denoted by a line containing either the string "//-" or the string "//.". A line containing "//-" followed by additional text denotes the beginning of a definition where the text is the name being defined. A top-level definition ends with a line containing "//-" with no trailing text or a line beginning with a right brace ("}"). A nested definition (such as of an operation within an interface) ends at the beginning of another definition or the end of the outer definition.

Lines that begin with "//." indicates text that should be written to the FrameMaker file. Index markers can be specify with the syntax "\marker{text}" where text is the index entry. Similarly, formatted text can be specified with the syntax "\emphasis{text}" for emphasis or "\bold{text}" for a boldface font.

5. Dish – Dynamic invocation shell

Dish is an application that allows invocation of Fresco objects through the Tcl scripting language. Users can bind Fresco objects to Tcl variables and commands, as well as directly invoke operations supported by these objects. The Tcl distribution and documentation are available via anonymous ftp from sprite.berkeley.edu.

The following commands are predefined by Dish:

bind *name object*
: Create the variable "name" with value equal to the given object and also define "name" as a command that invokes an operation (specified as the second argument) on the object.

unbind *name ...*
: Delete the variable and its associated commands. The object to which the name is bound is not deleted. Use the "unref" command to delete objects.

bindings
: List all binding names.

objects

Show a list of all accessible objects.

action ***tcl-command-line***
Create a Fresco Action object that executes the given Tcl command line when executed.

delay ***seconds***
Pause for the given number of seconds, which may be a floating-point number.

main ***viewer glyph***
Call the "main" operation on the Fresco object. This operation establishes a new display connection and maps a window containing the given viewer (for input) and glyph (for output).

ref ***object***
Call Fresco::ref on the given object, incrementing its reference count.

unref ***object***
Call Fresco::unref on the given object, decrementing its reference count and deleting it if the count reaches zero.

thread ***action***
Create and run a new thread that executes the given action.

Dish also predefines the objects drawing_kit, figure_kit, layout_kit, and widget_kit corresponding to the Fresco operations with the same name. Tcl strings are automatically converted to Fresco StringObj objects. Here are three example Dish scripts:

```
# Dish script for hello, world example
main 0 [figure_kit label [figure_kit default_style] "hello world"]

# Dish script for circle example
bind style [figure_kit default_style]
main 0 [figure_kit figure_root [figure_kit circle 0 style 0 0 100]]

# Dish script for button example
proc push_me {message} {
  puts $message
  flush stdout
}
set button [
  widget_kit push_button
    [figure_kit label [figure_kit default_style] "Push me"]
    [action push_me "pushed"]
]
ref $button
main $button [layout_kit margin $button 10.0]
```

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.

ActionImpl*

class ActionImpl : public Action

ActionImpl provides a common implementation for actions.

ActiveViewer*

class ActiveViewer : public ViewerImpl

The ActiveViewer class checks motion events by implementing ViewerImpl::move and ViewerImpl::drag to notice when the pointer enters or leaves the viewer.

enter

virtual void enter();

Notice that the pointer is now inside the viewer. The default is to do nothing.

leave

virtual void leave();

Notice that the pointer is now outside the viewer. The default is to do nothing.

Adjustment

interface Adjustment : FrescoObject

An adjustment is two segments that represent the possible and current range of values for a bounded value, such as would be used for scrolling.

begin, commit, cancel

void begin();
void commit();
void cancel();

These operations control when adjustment changes take effect. Operations to change the settings between a call to begin and commit will be treated as a single transaction in that the adjustment will be locked and no updates will be sent to any observers until the call to commit. The cancel operations stops the transaction and returns the settings to their value before the previous call to begin.

constrain

void constrain(inout Coord c);

Modify the given coordinate if necessary to keep it within the adjustment's settings. If the given value is below the adjustment's lower bound, then set it to the lower bound. If the given value plus the current length is large than the adjustment's upper bound, then set it to be the upper bound minus the current length.

get_settings

void get_settings(in Adjustment::Settings s);

Retrieve the current settings of the adjustment.

scale_to

void scale_to(in Coord length);

Request that the adjustment change its current length, effectively scaling the adjustment so that it covers more or less of the total segment.

scroll_by

void scroll_by(in Coord delta);

Request that the adjustment change its lower position to add the given delta (which may be negative).

scroll_forward, scroll_backward
page_forward, page_backward

void scroll_forward();
void scroll_backward();
void page_forward();
void page_backward();

Request that the adjustment change its current lower position setting. The scroll operations ask to move the position by the amount determined by the small_scroll attribute; the page operations by the large_scroll amount. The forward operations add the amount; the backward operations subtract the amount.

scroll_to

void scroll_to(in Coord lower);

Request that the adjustment change its lower position to the given coordinate.

Settings

struct Settings {
Coord lower, upper, length;
Coord cur_lower, cur_upper, cur_length;
};

The Settings structure defines the two ranges for an adjustment. The lower, upper, and length fields define the possible range of values; cur_lower, cur_upper, and define a current range. Providing both bounds and length is redundant information (since length is always equal to upper minus lower), but is more convenient than picking one representation over the other.

small_scroll, large_scroll

attribute Coord small_scroll;
attribute Coord large_scroll;

When used for scrolling, an adjustment provides attributes that define the size of movement for the scroll_forward, scroll_backward, page_forward, and page_backward operations. The small_scroll attribute is for use by the scroll operations; large_scroll for use by the page operations.

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.

Allocator*

class Allocator : public MonoGlyph

An Allocator is a glyph that always gives its child an allocation that matches the child's requisition. This functionality is useful as a gateway between figure objects, which ignore their allocation, and layout objects.

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.

BaseObject

class BaseObject

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

_type

virtual TypeObj _type();*

Return a reference to the object's type.

Create a request for performing dynamically-chosen operations on the object.

BaseThreadsObj

interface BaseThreadsObj

BaseThreadsObj only exists as a way to get reference counting for thread-related objects without deriving from FrescoObject. This interface may go away in the future.

Beveler*

class Beveler : public MonoGlyph

The beveler class draws a frame around its body and provides convenience operations for drawing bevelled objects.

Beveler

```
Beveler(
    Coord thickness, Alignment x = 0.0, Alignment y = 0.0,
    boolean hmargin = true, boolean vmargin = true
);
```

The constructor is given the frame thickness, optional x and y alignments, and optional margin flags. If the margin flags are true, then the beveler will allocate its body inside a frame of the given thickness; otherwise the body will have the same allocation as the beveler.

diamond

```
static void diamond(
    PainterRef p, Coord thickness,
    ColorRef light, ColorRef medium, ColorRef dark,
    Coord left, Coord bottom, Coord right, Coord top
);
```

Draw a diamond with sides that meet in the middle on all four edges.

down_arrow

static void down_arrow(
PainterRef p, Coord thickness,
ColorRef light, ColorRef medium, ColorRef dark,
Coord left, Coord bottom, Coord right, Coord top
);

Draw a beveled triangle with base parallel to the X-axis on the top and sides meeting in the middle on the bottom. This function is a bit misnamed, but such a triangle is typically used to represent a down-arrow.

fill_path

static void fill_path(PainterRef p, Vertex v, long n);*

Short-hand for filling a path containing the given vertices.

left_arrow

static void left_arrow(
PainterRef p, Coord thickness,
ColorRef light, ColorRef medium, ColorRef dark,
Coord left, Coord bottom, Coord right, Coord top
);

Draw a beveled triangle with base parallel to the Y-axis on the right and sides meeting in the middle on the left. This function is a bit misnamed, but such a triangle is typically used to represent a left-arrow.

rect

static void rect(
PainterRef p, Coord thickness,
ColorRef light, ColorRef medium, ColorRef dark,
Coord left, Coord bottom, Coord right, Coord top
);

Draw a beveled rectangle with the given thickness, upper-left color (light), middle color (medium), and lower-right color (dark). If medium is nil, then no drawing is done on the interior of the rectangle.

right_arrow

static void right_arrow(
PainterRef p, Coord thickness,
ColorRef light, ColorRef medium, ColorRef dark,
Coord left, Coord bottom, Coord right, Coord top
);

Draw a beveled triangle with base parallel to the Y-axis on the left and sides meeting in the middle on the right. This function is a bit misnamed, but such a triangle is typically used to represent a right-arrow.

up_arrow

static void up_arrow(
PainterRef p, Coord thickness,
ColorRef light, ColorRef medium, ColorRef dark,
Coord left, Coord bottom, Coord right, Coord top
);

Draw a beveled triangle with base parallel to the X-axis on the bottom and sides meeting in the middle on the top. This function is a bit misnamed, but such a triangle is typically used to represent an up-arrow.

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, which 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.

Button*

interface Button : Viewer

A button is a viewer that can perform an action when clicked (pressed and subsequently released with the pointer over the button's allocated area). Buttons manipulate a telltale that represents the current state of the button and determines the button's appearance.

click_action

attribute Action click_action;

The button will execute its action (if not nil) when clicked.

state

attribute Telltale state;

The button's state may affect the appearance of the button. If the state's enabled flag is on, then the button will change the state in response to input events.

ButtonImpl*

class ButtonImpl : public Button, public ActiveViewer

This class implements the Button interface using the ActiveViewer implementation for noticing when motion enters or leaves the button. The button modifies the telltale state in response to input, and observes the telltale state to cause a redraw whenever the state changes.

CharCode

typedef long CharCode

CharCode is a value that denotes a character from a particular character set (presumably Unicode?).

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 Color::Intensity r, out Color::Intensity g, out Color::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 DisplayObj::drawing_kit).

ConditionVariable*

interface ConditionVariable : BaseThreadsObj

A condition variable enables threads to block atomically until a condition is satisfied. The condition is tested under the protection of a mutual exclusion lock. When the condition is false, a thread typically blocks on a condition variable, which atomically releases the lock and waits for the condition variable to be notified. When another thread signals a change in the condition, one or more of the waiting threads may wake up.

broadcast

void broadcast();

Wakeup all threads waiting on this condition.

notify

void notify();

Wakeup one (or more) arbitrarily-chosen threads waiting on this condition.

wait

void wait(in LockObj lock);

Automatically release the given lock and sleep, then re-acquire the lock after waking. Threads that call the wait operation should re-check the condition after wait returns, as another thread could affect condition between the time when the waiting thread is awakened and actually starts execution.

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 user 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.

DamageObj

interface DamageObj : 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.

DebugGlyph*

class DebugGlyph : public MonoGlyph

DebugGlyph is a glyph that prints information about Glyph operations to standard output and then forwards the operations to its body. DebugGlyph is very useful for debugging layout management and screen update.

DebugGlyph

```
DebugGlyph(
    GlyphRef g, const char* msg,
    Flags flags = DebugGlyph::trace_request_traverse
);
```

The constructor takes as parameters the body, a message header to print at the beginning of each line of information, and the flags that determine which operations are of interest. The default flags are to record requests and all traversal operations.

Flags

```
enum Flags {
    trace_none = 0x0,
    trace_request = 0x1,
    trace_draw = 0x2,
    trace_pick = 0x4,
    trace_other = 0x8,
    trace_traverse = 0x1e,
    /* convenient shorthand */
    trace_request_traverse = 0x1f,
    trace_request_draw = 0x3,
    trace_request_pick = 0x5,
    trace_draw_pick = 0x6,
    trace_request_draw_pick = 0x7
};
```

The trace flags determine which operations will have information displayed.

print_region

static void print_region(RegionRef);

Print out information about an allocation. The format is “X(span), Y(span), Z(span)” where “span” is the output for the span along an axis.

print_requirement

static void print_requirement(Glyph::Requirement&);

Print out information about a requirement. If the requirement is undefined then “undef” is printed. Otherwise, the output format is “sizes @ alignment” where “@ alignment” is not displayed if the alignment is zero. If the minimum and natural sizes are the same then only the natural size is displayed. Similarly, if the natural and maximum sizes are the same then only the natural size is displayed. So if the requirement is rigid, only the natural size will be displayed.

print_span

static void print_span(const Region::BoundingSpan&);

Print out information about the span of a region. The format is “begin,end @ align” where “ @ align” is not displayed if the alignment is zero.

DeckObj*

interface DeckObj : Glyph

A deck is an aggregate that only shows one of its children at a time. The natural size of a deck is the maximum size of the natural sizes of its children. Regardless of the flexibility of their children, decks are rigid.

card

GlyphOffset card();

Return the offset for the currently-visible child (also known as the “top”).

flip_to

void flip_to(in GlyphOffset off);

Set the currently-visible child to the one that corresponds to the given offset.

DisplayObj

interface DisplayObj : 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.

add_filter, remove_filter

Tag add_filter(in Viewer v, in GlyphTraversal t);
void remove_filter(in Tag add_tag);

The add_filter operation requests that all events be passed to the given viewer with the given traversal before processing them further, even if the event would normally be delivered to a different viewer. If the viewer's handle operation returns true, then no other processing will occur. Otherwise, the event will be handled as if the filter were not present. If several filters are defined, they will be queried in the reverse order in which they are defined. To stop filtering, the tag returned by add_filter should be passed to the remove_filter operation.

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 pattern[16], in long mask[16]
);
Cursor cursor_from_bitmap(in Raster b, in Raster mask);
Cursor cursor_from_font(in Font f, in long pattern, in long mask);
Cursor cursor_from_index(in long index);

These operations return cursors that can be used on windows on the 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 indicates that the request have been processed.

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.

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.

screen, number_of_screens, default_screen

ScreenObj screen(in DisplayObj::ScreenNumber n);
DisplayObj::ScreenNumber number_of_screens();
ScreenObj 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.

ScreenNumber

typedef long ScreenNumber;

The ScreenNumber type defines the range of allowable integer values for screen numbers.

style

DisplayStyle style();

Return the style for the display.

DisplayStyle

interface DisplayStyle : StyleObj

A display style provides a type-checked interface to the predefined display attributes.

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.

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.

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.

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_file, bitmap_data, bitmap_char

Raster bitmap_file(in StringObj filename);
Raster bitmap_data(
in DrawingKit::Data data,
in Raster::Index rows, in Raster::Index columns,
in Raster::Index origin_row, in Raster::Index origin_column
);
Raster bitmap_char(in Font f, in CharCode c);

These operations return a bitmap created from one of three sources. The bitmap_file operation reads a file containing a bitmap definition in the format produced by the X bitmap program. If the file is not found or is in an invalid format, this operation returns nil.

The bitmap_data operation creates a bitmap given the binary data, sizes, and origin. 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.

color_rgb, find_color

Color color_rgb(in Color::Intensity r, in Color::Intensity g, in Color::Intensity b);
Color find_color(in StringObj 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.

Data

typedef sequence<long> Data;

This type defines a data structure for passing a sequence of 32-bit integer values.

find_font, find_font_match

Font find_font(in StringObj fullname);
Font find_font_match(
in StringObj family, in StringObj 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

PainterObj printer(in StringObj output);

Return a painter that generates printer output. The output parameter is the name of a file in which to generate PostScript.

raster_tiff

Raster raster_tiff(in StringObj filename);

Read a TIFF image file and convert it into a raster object.

resolve_color, foreground, background

Color resolve_color(in StyleObj s, in StringObj name);
Color foreground(in StyleObj s);
Color background(in StyleObj 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 StyleObj s, in StringObj name);
Font default_font(in StyleObj 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."

simple_brush, dither_brush, patterned_brush

Brush simple_brush(in Coord width);
Brush dither_brush(in Coord width, in long pattern);
Brush patterned_brush(in Coord width, in DrawingKit::Data pattern);

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.

style

StyleObj style();

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

transform, identity_transform

TransformObj transform(in TransformObj::Matrix m);
TransformObj identity_transform();

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

Event

interface Event : FrescoObject

An event is a device-level object that arrives a single stream per display.

button_is_down

boolean button_is_down(in Event::ButtonIndex b);

Return true if the specified button was down *before* the event occurred.

ButtonIndex

typedef long ButtonIndex;

A button index is an integer value that indicates to which button a down or up event refers.

keysym, character

Event::KeySym key();
CharCode character();

Return the keysym or character code associated with a key event. The result is undefined for non-key events.

KeySym, KeyChord

typedef unsigned long KeySym;
typedef sequence<KeySym> KeyChord;

A keysym is an integer value that identifies a key (as opposed to an interpretation of that key in a particular character set). X11 defines the standard keysym values supported by Fresco; some platforms may offer additional possible values.

Modifier

enum Modifier { control, shift, capslock, meta };

A modifier is a key that is normally held down when entering a second key.

modifier_is_down

boolean modifier_is_down(in Event::Modifier m);

Return true if the given modifier was down *before* the event occurred.

pointer_button

Event::ButtonIndex pointer_button();

Return the button associated with a down or up event. The return value is undefined for non-button events.

pointer_x, pointer_y

Coord pointer_x();
Coord pointer_y();

Return the pointer location for the event in coordinates relative to the lower-left corner of the main viewer in which the event occurred, or in the case of dragging the coordinates are relative to where the original down event occurred. Pointer locations are defined for motion, button, and keyboard events. The values for other events are undefined.

time

Event::TimeStamp time();

Return a time-stamp indicating when the event occurred.

TimeStamp

typedef unsigned long TimeStamp;

A time stamp on an event is an integral number of milliseconds.

type

Event::TypeCode type();

Return the event's type.

TypeCode

enum TypeCode {
motion, enter, leave, down, up, key_press, key_release,
focus_in, focus_out, other
};

The predefined event types are pointer motion, pointer enter or leave the main viewer, button down, button up, key press or release, and receiving or losing focus. Most platforms provide additional kinds of events, which will be delivered to a viewer with the type other. An application object wishing to process other events can narrow the event object to the platform-specific event type.

unread

void unread();

Put the event back on the input queue for re-delivery.

FigureKit*

interface FigureKit : FrescoObject

FigureKit provides operations for creating common two-dimensional figures, including as rectangles, circles, polygons, and B-splines. Once created, figures cannot be altered other than through the transformation matrix accessed by the Glyph::transform operation.

All figures have a style defined by the FigureStyle interface, which can be accessed by getting the Glyph::style attribute. Setting a figure's style using the Glyph::style attribute will actually set the parent style. Furthermore, because of the one-to-one relationship between a figure and its style, an implementation may choose to represent the figure and style as a single object that supports both interfaces (where retrieving a figure's style simply returns the figure itself).

circle

Glyph circle(
in FigureKit::Mode m, in FigureStyle s,
in Coord x, in Coord y, in Coord r
);

Return a circle with center (x,y) and radius r. The mode parameter controls whether the circle is filled, stroked, or both. The style parameter controls which brush and colors are used for drawing.

closed_bspline

Glyph closed_bspline(
in FigureKit::Mode m, in FigureStyle s, in FigureKit::Vertices v
);

Return a B-spline curve that uses the given vertices as control points. The curve is “closed” in that the last and first control points are automatically connected. The mode parameter controls whether the area is filled, stroked, or both. The style parameter controls which brush and colors are used for drawing.

default_style

FigureStyle default_style();

Return a default style to use for figures.

ellipse

Glyph ellipse(
in FigureKit::Mode m, in FigureStyle s,
in Coord x, in Coord y, in Coord r1, in Coord r2
);

Return an ellipse with center (x,y) and radii r1 and r2. The mode parameter controls whether the ellipse is filled, stroked, or both. The style parameter controls what brush and colors are used for drawing.

figure_root

Glyph figure_root(in Glyph child);

Return a glyph that centers its body by transformation. This kind of glyph should be put around a collection of figures that are inside a layout object such as a box.

fitter

Glyph fitter(in Glyph g);

Return a glyph that simply transforms its body to fit into the space available. That is, a fitter uses its allocation and its body's requested size to compute a transform to make the two sizes match. The fitter uses this transform during a traversal so that the body will be allocated its natural size, but will be transformed to drawn within the fitter's area.

group

Glyph group();

Return a group that can contain one or more figures.

label

Glyph label(in FigureStyle s, in StringObj str);

Return a text label with the given style.

line

Glyph line(
in FigureStyle s, in Coord x0, in Coord y0, in Coord x1, in Coord y1
);

Return a figure that draws a line from (x0,y0) to (x1,y1) using the brush and foreground color of the given style.

Mode

enum Mode { fill, stroke, fill_stroke };

Mode is an enumeration type the controls figure drawing. If the mode is “fill” then the figure fills its area with its style's foreground color. If the mode is “stroke” then the figure paints its outline with its style's brush and foreground color. If the mode is “fill_stroke” then the figure fills its area with its style's background color and paints the outline with the style's brush and foreground color.

multiline

Glyph multiline(
in FigureKit::Mode m, in FigureStyle s, in FigureKit::Vertices v
);

Return a list of lines defined by pair-wise of vertices, effectively yielding an "open" polygon. The mode parameter controls whether the area is filled, stroked, or both. The style parameter controls which brush and colors are used for drawing.

new_style

FigureStyle new_style(in StyleObj parent);

Return a new figure style that inherits from the given style.

open_bspline

Glyph open_bspline(
in FigureKit::Mode m, in FigureStyle s, in FigureKit::Vertices v
);

Return a B-spline curve that uses the given vertices as control points. The curve is "open" in that the last and first control points are not connected. The mode parameter controls whether the area is filled, stroked, or both. The style parameter controls which brush and colors are used for drawing.

point

Glyph point(in FigureStyle s, in Coord x, in Coord y);

Return a figure that draws a point in the foreground color of the given style.

polygon

Glyph polygon(
in FigureKit::Mode m, in FigureStyle s, in FigureKit::Vertices v
);

Return the polygon defined by the given vertices. The last and first vertices are automatically connected. The mode parameter controls whether the area is filled, stroked, or both. The style parameter controls which brush and colors are used for drawing.

rectangle

Glyph rectangle(
in FigureKit::Mode m, in FigureStyle s,
in Coord left, in Coord bottom, in Coord right, in Coord top
);

Return a rectangle with lower-left corner (left,bottom) and upper-right corner (right,top). The mode parameter controls whether the rectangle is filled, stroked, or both. The style parameter controls which brush and colors are used for drawing.

Vertices

typedef sequence<Vertex> Vertices;

The Vertices type defines a sequence of vertex structures for figures with an arbitrary number of vertices, such as polygons and B-splines.

FigureStyle*

interface FigureStyle : StyleObj

A figure style provides a type-checked interface to the predefined figure attributes.

background

attribute Color background;

Figures with a “fill_stroke” mode use the background color for filling.

brush

attribute Brush brush_attr;

Figures with a “stroke” or “fill_stroke” mode use the brush to draw the outline of the figure.

font

attribute Font font_attr;

Labels use the font attribute for drawing and computing the bounding box of labels.

foreground

attribute Color foreground;

Figures with a “fill” mode use the foreground color for filling. Figures with a “stroke” or “fill_stroke” mode use the foreground color to draw the outline of the figure.

Focus

interface Focus : FrescoObject

Every viewer hierarchy has a focus object at the root that holds shared information for all the viewers. In addition to maintaining which viewer has input focus, the focus object can coordinate input translation, shared menubar state, and shared tool palettes.

add_focus_interest

void add_focus_interest(in Viewer v);

Add the given viewer to the list of viewers that are on the path from the main viewer to the focus viewer.

map_keystroke, map_keychord

void map_keystroke(in Event::KeySym k, in Action a);
void map_keychord(in Event::KeyChord k, in Action a);

Specify a translation for a key event to an action. The map_keystroke operation defines the mapping for a given keysym, map_keychord maps a sequence of keysyms. The order of modifiers in a keychord is ignored; an event will be mapped if the modifiers in the sequence are down.

If the main viewer receives a key event that is mapped, it will perform the action immediately rather than passing it to the focus viewer.

receive_focus_below, lose_focus_below

void receive_focus_below(in Viewer v, in boolean temporary);
void lose_focus_below(in Viewer v, in boolean temporary);

Notify all the viewers on the interest list that they have received or lost the input focus.

Font

interface Font : FrescoObject

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

encoding

StringObj 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 StringObj 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

StringObj 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 StringRef 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.

create_request

virtual RequestObjRef create_request(BaseObjectRef) = 0;

Return a new request for dynamically invoking an operation on the given object.

drawing_kit

virtual void drawing_kit(DrawingKit) = 0;*
virtual DrawingKit drawing_kit() = 0;*

Set or get the kit creating drawing objects such as colors and fonts.

figure_kit

virtual void figure_kit(FigureKit) = 0;*
virtual FigureKit figure_kit() = 0;*

Set or get the kit for creating simple graphics figures.

layout_kit

virtual void layout_kit(LayoutKit) = 0;*
virtual LayoutKit layout_kit() = 0;*

Set or get the kit for creating simple layout manager objects.

main

virtual void main(Viewer v, Glyph* g) = 0;*

Start and run a “main loop” in the common case with a single viewer on a single display. This operation creates and maps a main viewer on the default screen of the main Fresco display (see Fresco_open or Fresco_open_display). The main viewer uses the given viewer for input (unless the viewer is nil, in which case input is ignored) and the given glyph for output (unless the glyph is nil, in which case the viewer is used for output). After mapping the viewer, main calls DisplayObj::run and waits for it to return.

open_display, open_default_display

virtual DisplayObj open_display(StringRef name) = 0;*
virtual DisplayObj 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, string_copy

static StringRef string_ref(const char);*
static StringRef 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.

style

virtual StyleRef style() = 0;

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

thread_kit

virtual void thread_kit(ThreadKit) = 0;*
virtual ThreadKit thread_kit() = 0;*

Set or get the kit for creating threads and synchronization objects.

widget_kit

virtual void widget_kit(WidgetKit) = 0;*
virtual WidgetKit widget_kit() = 0;*

Set or get the kit for creating simple widgets such as buttons, menus, and scrollbars.

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 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;
    TransformObj transform;
    DamageObj damage;
};
```

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

AllocationInfoList

typedef sequence<AllocationInfo> AllocationInfoList;

AllocationInfoList 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::AllocationInfoList 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.

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.

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.

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, extension, shape

void request(out Glyph::Requisition r);
void extension(in AllocationInfo a, in Region r);
void shape(in Region r);

These three operations query information about the size of a glyph. The request operation asks for a glyph's desired geometry. Request should be fast to allow a composite glyph to compute rather than store its components' requisitions. If a requisition might be expensive to compute, then a glyph should store the 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.

The extension operation computes the region bounds of where a glyph would naturally draw. An extension is normally the same as a requisition's natural size, only differing in cases where the drawing goes outside the object's allocation, such as a checkmark or slanted text.

The shape operation asks the glyph to generate the precise shape of where it would naturally draw. Calling shape may be more costly than calling extension (which just computes bounds) and therefore is only called by composites that wish to perform layout using the actual area where the object will draw.

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.

style

attribute StyleObj 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.

transform

TransformObj transform();

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.

visit_children, visit_children_reversed, visit_parents

void visit_children(in GlyphVisitor v);
void visit_children_reversed(in GlyphVisitor v);
void visit_parents(in GlyphVisitor v);

Visit each of a glyph's children or parents. The visit_children_reversed operation begins with the last child and iterates through the list backwards to the first child. Parents are not ordered, so no similar capability is necessary.

GlyphImpl*

class GlyphImpl : public Glyph

GlyphImpl is a default implementation for library glyph objects (those that live in the same address space as their creator). GlyphImpl should be not used for aggregates, only leaf glyphs.

init_requisition, default_requisition

static void init_requisition(Glyph::Requisition& r);
static void default_requisition(Glyph::Requisition& r);

These operations initialize the given requisition. The init_requisition operation sets the requirements along each axis to be undefined. The default_requisition operation sets each requirement to have a rigid, zero size and zero alignment.

require

```
static void require(
    Glyph::Requirement& r,
    Coord natural, Coord stretch, Coord shrink, float alignment
);
```

Set the given requirement to the given sizes and alignment.

require_lead_trail

```
static void require_lead_trail(
    Glyph::Requirement& r,
    Coord natural_lead, Coord max_lead, Coord min_lead,
    Coord natural_trail, Coord max_trail, Coord min_trail
);
```

Set the given requirement based on the given sizes that define the natural, maximum, and minimum sizes before (lead) and after (trail) the requirement's origin.

requirement

static Requirement requirement(Glyph::Requisition& r, Axis a);*

Return the requirement for the given requisition along the given axis.

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::AllocationInfoList 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.

insert

void insert(in Glyph g);

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

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.

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

Region allocation();
boolean bounds(out Vertex lower, out Vertex upper, out Vertex origin);
boolean origin(out Vertex origin);
boolean span(in Axis a, out Region::BoundingSpan s);

These operations return the current allocation or information about the allocation. The bounds, origin, and span operations are short-hand for retrieving the allocation region and retrieving information from it. The bounds, origin, and span operations return false if the allocation is not defined (which could happen during a non-allocating traversal).

begin_trail, end_trail

void begin_trail(in Viewer v);
void end_trail();

Record the beginning and end of a trail associated with the given viewer.

clear

void clear();

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

damage

DamageObj damage();

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

display, screen
DisplayObj display();
ScreenObj screen();

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

forward, backward
void forward();
void 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.

glyph
Glyph current_glyph();

Return the current glyph being traversed.

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.

offset
GlyphOffset offset();

Return the current offset being traversed.

op, swap_op
GlyphTraversal::Operation op();
GlyphTraversal::Operation swap_op(in GlyphTraversal::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.

painter
attribute PainterObj painter;

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

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.

viewer

Viewer current_viewer();

Return the viewer at the head of the current trail.

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.

GlyphVisitor

interface GlyphVisitor : FrescoObject

A glyph visitor is essentially a callback for iterating over a list of glyph children or parents.

visit

boolean visit(in Glyph glyph, in GlyphOffset offset);

Visit the given glyph. During a child iteration, the glyph is the offset's child. During a parent iteration, the glyph is the offset's parent. The visit operation should return whether or not the iteration should continue (returning true means continue, false means stop).

GlyphVisitorImpl

class GlyphVisitorImpl : public GlyphVisitor

GlyphVisitorImpl provides a default implementation for GlyphVisitor. Other implementations can subclass from GlyphVisitorImpl and just redefine visit.

LayoutKit*

interface LayoutKit : FrescoObject

LayoutKit provides operations for creating glyphs that are useful in managing the arrangement of one or more glyphs. Many of the LayoutKit objects are modelled after Knuth's TeX document processing system.

back

Glyph back(in Glyph g, in Glyph under);

Return a glyph that draws the second parameter glyph first, then draws the body specified by the first parameter. Structural operations such as Glyph::append are delegated to the body glyph.

between

Glyph between(in Glyph g, in Glyph under, in Glyph over);

Return a glyph that draws the second parameter glyph first, then the body glyph specified by the first parameter, and then the third parameter glyph. Structural operations such as Glyph::append are delegated to the body glyph.

bmargin, bmargin_flexible

Glyph bmargin(in Glyph g, in Coord natural);
Glyph bmargin_flexible(
in Glyph g, in Coord natural, in Coord stretch, in Coord shrink
);

These operations return a glyph with a bottom margin. The bmargin operation is equivalent to "vmargin(g, natural, 0)." The bmargin_flexible operation is equivalent to "vmargin(g, natural, stretch, shrink, 0, 0, 0)."

center, center_aligned, center_axis

Glyph center(in Glyph g);
Glyph center_aligned(in Glyph g, in Alignment x, in Alignment y);
Glyph center_axis(in Glyph g, in Axis a, in Alignment align);

Return a glyph that positions a given glyph according to a given alignment. The center operation uses an alignment of 0.5, which effectively centers the interior glyph inside an allocation.

deck

DeckObj deck();

Return a new deck. See the DeckObj interface definition.

fil

attribute Coord fil;

The "fil" attribute defines an "infinite" coordinate for the purposes of specifying stretchability that is large relative to other objects.

fixed, fixed_axis, hfixed, vfixed

Glyph fixed(in Glyph g, in Coord x, in Coord y);
Glyph fixed_axis(in Glyph g, in Axis a, in Coord size);
Glyph hfixed(in Glyph g, in Coord x);
Glyph vfixed(in Glyph g, in Coord y);

These operations return a glyph that makes a given glyph act rigid even if it is flexible. Fixed glyphs are ideal for specifying sizes that are otherwise undefined, such as the initial dimensions of a file chooser. The fixed operation returns a glyph fixed with given width and height natural sizes. The fixed_axis operation returns a glyph fixed in one dimension. The hfixed and vfixed operations return fixed glyphs for the horizontal and vertical axes, respectively.

flexible, flexible_fil, flexible_axis, hflexible, vflexible

Glyph flexible(in Glyph g, in Coord stretch, in Coord shrink);
Glyph flexible_fil(in Glyph g);
Glyph flexible_axis(
    in Glyph g, in Axis a, in Coord stretch, in Coord shrink
);
Glyph hflexible(in Glyph g, in Coord stretch, in Coord shrink);
Glyph vflexible(in Glyph g, in Coord stretch, in Coord shrink);

These operations return a glyph that makes a given glyph act flexible even if is rigid. The flexible operation returns a glyph with a given stretchability and shrinkability along all axes. The flexible_fil operation is equivalent to calling flexible with fil stretchability and zero shrinkability. The hflexible and vflexible operations return a glyph with a given flexibility in the X or Y axis, respectively.

front

Glyph front(in Glyph g, in Glyph over);

Return a glyph that draws the body glyph first, specified by the first parameter, and then draws the second parameter glyph. Structural operations such as Glyph::append are delegated to the body glyph.

glue

Glyph glue(
    in Axis a,
    in Coord natural, in Coord stretch, in Coord shrink, in Alignment align
);

Glue is the TeX term for flexible spacing between other glyphs. LayoutKit provides a variety of operations for creating glue objects. The glue operation is the most general, specifying the axis of flexibility, the natural size, stretchability, shrinkability, and alignment along that axis. A glue object's requirement along other axes is undefined.

glue_requisition

Glyph glue_requisition(in Glyph::Requisition r);

Return a glue object that uses the given requisition as its desired size.

hbox, vbox

Glyph hbox();
Glyph vbox();

An hbox tiles its children left-to-right along the X axis and aligns their origins along the Y axis. A vbox tiles its children top-to-bottom along the Y axis and aligns their origins along the X axis. The natural, minimum, and maximum sizes along the tiling axis are the sums of the child natural, minimum, and maximum sizes, respectively. The natural size along the alignment axis is the maximum of the child natural sizes. The minimum size along the alignment axis is the maximum of the child minimum sizes. The maximum size along the alignment axis is the minimum of the child maximum sizes.

The X alignment of an hbox is the left side of the box. The Y alignment of a vbox is the top side of the box.

hbox_aligned, vbox_aligned

Glyph hbox_first_aligned();
Glyph vbox_first_aligned();

These operations are variations on hbox and vbox, returning objects that only differ in their alignment along the tiling axis. Both hbox_first_aligned and vbox_first_aligned use the alignment of their first component for their alignment along the tiling axis.

hcenter, hcenter_aligned, vcenter, vcenter_aligned

Glyph hcenter(in Glyph g);
Glyph hcenter_aligned(in Glyph g, in Alignment x);
Glyph vcenter(in Glyph g);
Glyph vcenter_aligned(in Glyph g, in Alignment y);

Return a glyph that positions a given glyph in one dimension according to a given alignment. The hcenter and vcenter operations use an alignment of 0.5, which effectively centers the interior glyph along the X or Y axis, respectively.

hglue, hfil, hglue_fil, hglue_aligned, hspace

Glyph hfil();
Glyph hglue_fil(in Coord natural);
Glyph hglue(in Coord natural, in Coord stretch, in Coord shrink);
Glyph hglue_aligned(
    in Coord natural, in Coord stretch, in Coord shrink, in Alignment a
);
Glyph hspace(in Coord natural);

Return different kinds of horizontal glue. The hfil and hglue_fil operations return glue that is infinitely stretchable and not shrinkable; hfil has zero for its natural size. The hglue and hglue_aligned operations return horizontal glue with the given sizes. The hspace operation returns rigid glue.

hmargin, hmargin_lr, hmargin_lr_flexible

Glyph hmargin(in Glyph g, in Coord both);
Glyph hmargin_lr(in Glyph g, in Coord lmargin, in Coord rmargin);
Glyph hmargin_lr_flexible(
    in Glyph g,
    in Coord lmargin, in Coord lstretch, in Coord lshrink,
    in Coord rmargin, in Coord rstretch, in Coord rshrink
);

These operations are short-hand for specifying margin glyphs with zero-size top and bottom margins. The hmargin operation is equivalent to "margin_lrbt(g, both, both, 0, 0)." The hmargin_lr operation is equivalent to "margin_lrbt(g, lmargin, rmargin, 0, 0)." The hmargin_lr_flexible operation is equivalent to "margin_lrbt_flexible(g, lmargin, lstretch, lshrink, rmargin, rstretch, rshrink, 0, 0, 0, 0, 0, 0)."

hstrut

Glyph hstrut(
in Coord right_bearing, in Coord left_bearing,
in Coord natural, in Coord stretch, in Coord shrink
);

lmargin, lmargin_flexible

Glyph lmargin(in Glyph g, in Coord natural);
Glyph lmargin_flexible(
in Glyph g, in Coord natural, in Coord stretch, in Coord shrink
);

These operations return a glyph with a left margin. The lmargin operation is equivalent to "hmargin(g, natural, 0)." The lmargin_flexible operation is equivalent to "hmargin(g, natural, stretch, shrink, 0, 0, 0)."

margin, margin_lrbt, margin_lrbt_flexible

Glyph margin(in Glyph g, in Coord all);
Glyph margin_lrbt(
in Glyph g,
in Coord lmargin, in Coord rmargin, in Coord bmargin, in Coord tmargin
);
Glyph margin_lrbt_flexible(
in Glyph g,
in Coord lmargin, in Coord lstretch, in Coord lshrink,
in Coord rmargin, in Coord rstretch, in Coord rshrink,
in Coord bmargin, in Coord bstretch, in Coord bshrink,
in Coord tmargin, in Coord tstretch, in Coord tshrink
);

These operations return a glyph that extends the size of a glyph to leave room for a margin of a given size. The margin operation returns a glyph with the same fixed size for the left, right, bottom, and top margins. The margin_lbrt operation returns a glyph with specified fixed sizes for each margin. The margin_lrbt_flexible operation is the most general, returning a glyph with independent sizes and flexibility for each margin.

natural, natural_axis, hnatural, vnatural

Glyph natural(in Glyph g, in Coord x, in Coord y);
Glyph natural_axis(in Glyph g, in Axis a, in Coord size);
Glyph hnatural(in Glyph g, in Coord x);
Glyph vnatural(in Glyph g, in Coord y);

These operations return a glyph that makes a given glyph have a specified natural size instead of the glyph's original natural size. Natural glyphs do not affect the rigidity of the given glyph. The natural operation returns a glyph with a given natural width and height. The natural_axis operation returns a glyph with a given natural size along a given axis. The hnatural and vnatural operations return a glyph with the given natural size along the X or Y axis, respectively.

overlay

Glyph overlay();

An overlay aligns the origins of its children along both the X and Y axis. Overlays draw their children first-to-last, guaranteeing that the later glyphs will appear "above" earlier glyphs with the same z-coordinate.

rmargin, rmargin_flexible

Glyph rmargin(in Glyph g, in Coord natural);
Glyph rmargin_flexible(
in Glyph g, in Coord natural, in Coord stretch, in Coord shrink
);

These operations return a glyph with a right margin. The rmargin operation is equivalent to “hmargin(g, 0, natural).” The rmargin_flexible operation is equivalent to “hmargin(g, 0, 0, 0, natural, stretch, shrink).”

shape_of, shape_of_xy, shape_of_xyz

Glyph shape_of(in Glyph g);
Glyph shape_of_xy(in Glyph gx, in Glyph gy);
Glyph shape_of_xyz(in Glyph gx, in Glyph gy, in Glyph gz);

Return a glyph that has the same desired size as the given glyph or glyphs, but does not perform any drawing. The shape_of operation has the same sizes as a given glyph along all axes, shape_of_xy uses different glyphs for the X and Y axis, and shape_of_xyz also uses a different glyph for the Z axis (shape_of_xy returns a glyph whose Z requirement is undefined).

spaces

Glyph spaces(
in long count, in Coord each, in Font f, in Color c
);

Return glue that corresponds to a given number of spaces of a given width in a given font. The font determines the origin of the glue. Unlike other glue objects, which despite begin generally referred to as “white space” do not actually have an appearance, spaces are drawn in the given color.

strut

Glyph strut(
in Font f, in Coord natural, in Coord stretch, in Coord shrink
);

tmargin, tmargin_flexible

Glyph tmargin(in Glyph g, in Coord natural);
Glyph tmargin_flexible(
in Glyph g, in Coord natural, in Coord stretch, in Coord shrink
);

These operations return a glyph with a top margin. The tmargin operation is equivalent to “vmargin(g, 0, natural).” The tmargin_flexible operation is equivalent to “vmargin(g, 0, 0, 0, natural, stretch, shrink).”

vglue, vfil, vglue_fil, vglue_aligned, vspace

Glyph vfil();
Glyph vglue_fil(in Coord natural);
Glyph vglue(in Coord natural, in Coord stretch, in Coord shrink);
Glyph vglue_aligned(
in Coord natural, in Coord stretch, in Coord shrink, in Alignment a
);
Glyph vspace(in Coord natural);

Return different kinds of vertical glue. The vfil and vglue_fil operations return glue that is infinitely stretchable and not shrinkable; vfil has zero for its natural size. The vglue and vglue_aligned operations return vertical glue with the given sizes. The vspace operation returns rigid glue.

vmargin, vmargin_bt, vmargin_bt_flexible

Glyph vmargin(in Glyph g, in Coord both);
Glyph vmargin_bt(in Glyph g, in Coord bmargin, in Coord tmargin);
Glyph vmargin_bt_flexible(
in Glyph g,
in Coord bmargin, in Coord bstretch, in Coord bshrink,
in Coord tmargin, in Coord tstretch, in Coord tshrink
);

These operations are short-hand for specifying margin glyphs with zero-size left and right margins. The vmargin operation is equivalent to “margin_lrbt(g, 0, 0, both, both).” The vmargin_bt operation is equivalent to “margin_lrbt(g, 0, 0, bmargin, tmargin).” The vmargin_bt_flexible operation is equivalent to “margin_lrbt_flexible(g, 0, 0, 0, 0, 0, 0, bmargin, bstretch, bshrink, tmargin, tstretch, tshrink).”

vstrut

Glyph vstrut(
in Coord ascent, in Coord descent,
in Coord natural, in Coord stretch, in Coord shrink
);

LockedFrescoObjectImpl*

class LockedFrescoObjectImpl : public SharedFrescoObjectImpl

This class provides common code for implementing basic FrescoObject operations that assume mutual exclusion for accessing the object state. Instances are normally included inside other object implementations. The container object should set the lock instance variable appropriately.

LockObj*

interface LockObj : BaseThreadsObj

Locks implement mutual exclusion for serialization of thread execution. Locks typically are used to ensure that only one thread executes a critical section of code at any one time.

acquire

void acquire();

If no thread currently holds it, then the calling thread acquires the lock. Otherwise, the caller will block until the lock is released by the current holder. No guarantee is made that the first of several threads to block on a lock will be the first to be unblocked.

release

void release();

If the calling thread holds the lock, then the lock is released and a thread waiting to acquire the lock will be unblocked. If the caller does not hold a lock, then this operation may fail in a system-dependent way. [Note: we should define an exception to raise in this situation.]

try_acquire

boolean try_acquire();

Attempt to acquire the lock, returning true if the lock is successfully acquired. If the lock is currently held by a thread, then this operation will return false immediately rather than wait for the lock.

MonoGlyph*

class MonoGlyph : public GlyphImpl

A MonoGlyph delegates some of its operations to another glyph, called the body.

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 (“:”).

PainterObj

interface PainterObj : 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->end_path();
painter->fill();
```

brush, color, font

attribute Brush brush_attr;
attribute Color color_attr;
attribute Font font_attr;

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.

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, push_clipping, pop_clipping, is_visible

void clip(in Coord x0, in Coord y0, in Coord x1, in Coord y1);
void push_clipping();
void pop_clipping();
boolean is_visible(in Region r);

The clip operation restricts subsequent drawing to be inside 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. The is_visible operation returns true if any portion of the given region is within the drawing area defined by the clipping region.

comment

void comment(in StringObj s);

This operation may write an annotation, which can be useful when generating a printable representation.

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.

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.

matrix, push_matrix, pop_matrix, transform

attribute TransformObj matrix;
void push_matrix();
void pop_matrix();
void transform(in TransformObj t);

The 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 transform operation performs a premultiply on the current matrix with the given one.

page_number

void page_number(in StringObj 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.

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.

PolyGlyph*

class PolyGlyph : public GlyphImpl

PolyGlyph is a default implementation for aggregate glyphs. PolyGlyph inherits most operations from GlyphImpl, redefining only those operations relating to aggregates. PolyGlyph represents children as a list of PolyGlyphOffset objects that denote the child and the child's index into the list.

change

virtual void change(long index);

Note that the request of the indexed child may have changed.

child_allocation

virtual void child_allocation(long index, Glyph::AllocationInfo& a);

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

children_requests

Glyph::Requisition children_requests(Glyph::Requisition* req, long n);*

Return an array of requisitions for the polyglyph's children. If the given integer is as large as the number of children, then the given pointer will be used for the result array. Otherwise, an array will be dynamically-allocated that the caller should delete.

fixup

virtual void fixup(long start, long delta);

This operation is called when a list item is added or removed, meaning that the indices in all offsets must be updated.

modified

virtual void modified();

Note that the child list has been modified. A polyglyph will normally cache its requisition; if so this operation should invalidate the cache.

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. 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

Raster::Index origin_x();
Raster::Index origin_y();

Return the origin of the raster.

peek, poke

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

Raster::Index rows();
Raster::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.

RasterBitmap*

class RasterBitmap : public RasterImpl

RasterBitmap provides a raster implementation for bitmaps using X images.

RasterChannels*

class RasterChannels : public RasterImpl

RasterChannels is an implementation of Raster supporting different numbers of channels, where 1 channels means grey, 2 means grey with alpha, 3 means RGB, and 4 means RGB with alpha.

RasterImpl*

class RasterImpl : public Raster

RasterImpl is a base class for common raster implementations on X. It does not implement the raster data (i.e., peek and poke), but provides shared operations for mapping raster coordinates to screen coordinates.

RasterLUT*

class RasterLUT : public RasterImpl

Raster implementation using a color lookup table.

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.

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.

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_intersect on the given region.

merge_intersect, merge_union, subtract

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.

transform

void transform(in TransformObj 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.

RegionImpl*

class RegionImpl : public Region

RegionImpl is a simple implementation of the Region interface, managing only the bounding box of the region.

notify

virtual void notify();

Note that the region has been modified by the merge, subtract, or transform operations.

RegionImpl

RegionImpl();

The constructor initializes an empty bounding box with all coordinates zero.

RequestObj

interface RequestObj

A request object is used to perform dynamic invocation of operations on objects. These objects are useful for script or other kinds of interpreters.

add_arg

void add_arg(in string value);

Add the argument to the operation. The request object will try to convert the string to the type of the corresponding formal parameter.

CallStatus

```
enum CallStatus {
    initial, ok,
    unknown_operation, bad_argument_count, bad_argument_type
};
```

This type defines the possible return status values for a call.

invoke

CallStatus invoke();

Call the operation on the object.

op_info

boolean op_info(out TypeObj::OpInfo op);

Retrieve the information of the operation associated with the request, if any. The return value is true if the request corresponds to a valid operation.

result_string

string result_string();

Return a string containing the value returned by the operation. This operation returns a nil string if the operation has not been invoked.

set_operation

void set_operation(in string s);

Specify the operation for which this request is intended.

ScreenObj

interface ScreenObj : FrescoObject

A screen represents a physical array of pixels.

application

Window application(in Viewer v);

Create a new application window for the given viewer. An application should create a single application window, which may make information about command-line arguments available to a session manager.

display

DisplayObj 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.

icon

Window icon(in Viewer v);

Create a new icon window for the given viewer.

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.

popup

Window popup(in Viewer v);

Create a new popup window for the given viewer. A popup window is unmapped directly to a screen without window manager interaction (or knowledge). In the X window system, a popup window will override the normal redirection of map requests to window managers. Popups on X also will request that the pixels under the popup be saved to avoid a subsequent exposure when the popup is unmapped. Popups should only be used for temporary windows, such as popup or pulldown menus. Because they do not go through the window manager, popups must be placed explicitly.

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.

top_level

Window top_level(in Viewer v, in Window group_leader);

Create a new top-level window for the given viewer. A top-level window can have a “group leader,” which is typically an application window. A window manager might allow quick control of all windows with the same group leader, such as when iconifying or deiconifying.

transient

Window transient(in Viewer v, in Window transient_for);

Create a new transient window for the given viewer. A transient window is often treated specially by window mangers. It may be decorated differently (or not at all), or automatically unmapped when the main window is iconified. Transient windows are used for temporary controls, such as dialog boxes, but not for unmanaged windows such as popups or pulldowns. A transient is usually associated with another managed window.

width, height

Coord width();
Coord height();

Return the screen dimensions in resolution-independent coordinates.

Semaphore*

interface Semaphore : BaseThreadsObj

Semaphores can coordinate access to resources. A semaphore is created with an initial count that corresponds to the number of free resources. Threads can then atomically increment the count with the P operation and atomically decrement the count with the V operation.

P

void P();

Wait until the semaphore's count is positive, then decrement it.

V

void V();

If one or more threads are waiting on the semaphore, then wake one up (chosen arbitrarily). Otherwise, increment the count.

SharedFrescoObjectImpl*

class SharedFrescoObjectImpl : public SharedBaseObjectImpl

This class provides common code for implementing the basic FrescoObject operations. Instances are normally included inside other object implementations. This class purposely does not define any virtual functions, so be careful about subclassing from it.

StringObj

interface StringObj

A string represents a sequence of characters, where a character is a logical value rather than a specific physical size. The set of string operations may vary for a particular implementation language, but one should minimally be able to set or get a Unicode value for each character in the string.

StyleObj

interface StyleObj : 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 StringObj s);

Add an alias to the list for the style.

bind

StyleValue bind(in StringObj 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 StringObj 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 StyleObj 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 StyleObj 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 StringObj 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 StyleObj s);

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

name

attribute StringObj 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

StyleObj new_style();

Return a newly-created style.

parent

StyleObj parent_style();

Return the style's parent style, if any.

resolve

StyleValue resolve(in StringObj 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 StringObj name, in StyleObj 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 StringObj 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”.

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

StringObj 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 StringObj s);
void write_string(in StringObj 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 StringObj name);
boolean visit_attribute(in StyleValue a);
boolean visit_style(in StyleObj 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.

Telltale*

interface Telltale : FrescoObject

A telltale reflects the current state of a button in terms of a set of flags that are on or off. A telltale may also belong to a telltale group, which can ensure that only one telltale in the group has the chosen flag on.

clear

void clear(in Telltale::Flag f);

Turn off the given flag.

current

attribute Telltale current;

The current attribute is by default a nil reference. This attribute is intended to support radio buttons or other mutually-exclusive choice settings.

If set, then current attribute refers to a “group” telltale that contains a currently-chosen telltale in the group's current attribute. When a telltale's chosen flag is changed from clear to set and its current attribute is not nil, the current attribute refers to a group telltale. The group's current attribute refers to the previously-chosen telltale, which will have its chosen flag cleared. Then the group's current attribute will be changed to refer to the newly-chosen telltale.

Flag

enum Flag {
enabled, visible, active, chosen, running, stepping,
choosable, toggle
};

The telltale flags represent different states associated with a button. The enabled flag is on when the button is handling input. The visible flag is on when the pointer is inside the button's output area. The active flag is on between the time the button receives a down event and when the corresponding up event occurs. The chosen flag indicates the button is currently selected, a state normally associated with radio buttons or check boxes. The running flag is on while the button is executing its action. The stepping flag is on while the button is autorepeating (if the button supports autorepeat). The choosable flag indicates whether the button can be set to chosen or not. Push buttons are normally not choosable because a chosen button will not execute its action when clicked. The toggle flag indicates the state should flip between chosen and not chosen when clicked.

set

void set(in Telltale::Flag f);

Turn on the given flag.

test

boolean test(in Telltale::Flag f);

Return true if given flag is currently on.

TelltaleImpl*

class TelltaleImpl : public Telltale

The TelltaleImpl class implements the Telltale interface.

flags

TelltaleImpl::FlagSet flags();

Return the current set of “visibility” flags, which only includes enabled, visible, active, and chosen. This function is useful for indexing an array of colors, one per set of states.

FlagSet

typedef short FlagSet;
enum {
off = 0,
enabled_bit = 1 << Telltale::enabled,
visible_bit = 1 << Telltale::visible,
active_bit = 1 << Telltale::active,
chosen_bit = 1 << Telltale::chosen,
running_bit = 1 << Telltale::running,
stepping_bit = 1 << Telltale::stepping,
choosable_bit = 1 << Telltale::choosable,
toggle_bit = 1 << Telltale::toggle
};

We use a short to represent the set of possible telltale flags, meaning we must define the enumeration as a separate type (because C++ enumerations are always 32 bits). The encoding of bits here assumes that the Telltale::Flag enumeration defines constants that start at 0 and increment by 1.

ThreadKit*

interface ThreadKit : BaseThreadsObj

ThreadKit provides operations for creating threads, mutual exclusion locks, condition variables, and semaphores.

condition

ConditionVariable condition();

Create a condition variable.

lock

LockObj lock();

Create a mutual exclusion lock.

mutex_semaphore

Semaphore mutex_semaphore();

Create a semaphore with an initial count of one.

semaphore

Semaphore general_semaphore(in long count);

Create a general semaphore with the given initial count.

thread

ThreadObj thread(in Action a);

Create a thread that will execute the given action as its main function when the thread is run. If the host system does not support threads, this operation will return nil.

wait_semaphore

Semaphore wait_semaphore();

Create a semaphore with an initial count of zero.

ThreadObj*

interface ThreadObj : BaseThreadsObj

The ThreadObj interface defines a simple set of operations that are intended to be similar to a subset of the POSIX thread functionality.

run

void run();

Execute the thread's main function, as defined by the action passed when the thread was created. The thread will exit when the function completes.

terminate

void terminate();

Stop execution of the thread and reclaim its storage. The thread cannot be restarted after termination.

wait

void wait(out long status, out long exitcode);

Wait for the thread to exit. If the thread was terminated by a signal, the status will be non-zero and contains the signal number. If the status is zero, then the exitcode contains the thread's exit code.

TransformAllocator*

class TransformAllocator : public Allocator

A TransformAllocator maps its allocate to a translation during traversal and always gives its child the child's natural allocation. This functionality is useful as a gateway between layout objects and figure objects (which ignore their allocation).

TransformImpl*

class TransformImpl : public TransformObj

TransformImpl is a default implementation of the Transform interface. The current implementation only supports 3x2 matrices, but this will be replaced with 4x4 matrix code in the near future.

TransformObj

interface TransformObj : 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 TransformObj 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, load_identity, store

void load(in TransformObj::Matrix m);
void load_identity();
void store(out TransformObj::Matrix m);

Copy the transform from (load operation) or to (store operation) a given Matrix. The load_identity operation sets the transform to be the identity matrix.

Matrix

typedef Coord Matrix[4][4];

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

premultiply, postmultiply

void premultiply(in TransformObj t);
void postmultiply(in TransformObj 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.

transform, inverse_transform

void transform(inout Vertex v);
void inverse_transform(inout Vertex v);

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

TypeObj

interface TypeObj

TypeObj represents the type of an object.

is_interface

boolean is_interface();

Return whether this type is an interface, as opposed to a concrete or structured type.

name

string name();

Return the name of the type. The name is as defined in the source, not qualified by any containing scopes.

op_info

boolean op_info(out TypeObj::OpInfo op, in long index);

Retrieve the information about the operation referred to by the given index, if any. This operation returns true if the index is valid.

OpInfo

struct OpInfo {
string name;
long index;
TypeObj result;
sequence<ParamInfo> params;
};

This structure contains information about an operation or attribute.

ParamInfo

struct ParamInfo {
string name;
ParamMode mode;
TypeObj type;
};

This structure contains information about a particular parameter.

ParamMode

enum ParamMode { param_in, param_out, param_inout };

ParamMode is an enumeration for the various parameter passing modes.

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.

Viewer

interface Viewer : Glyph

The Viewer interface defines the mechanisms for the logical composition of user interface objects. Viewer is a Glyph subtype, and as such delegates the presentation and geometry of the viewer to another glyph. A viewer implementation should define the pick operation to be opaque in the sense that the viewer should appear as a leaf glyph relative to pick calls. This approach hides the viewer's structure from surrounding objects.

Unlike glyphs, which may be organized with sharing to form a directed-acyclic graph, viewers must be organized in a strict hierarchy. Using a hierarchy simplifies input handling and focus management.

Every viewer defines a unique style with an alias identifying the type of viewer. Application objects should set the name of a viewer's style to provide a convenient mechanism for customizing a particular instance. Setting a viewer's style using the Glyph::style attribute will actually set the parent style. By default, the parent style is the style of the viewer's parent. Furthermore, because of the one-to-one relationship between a viewer and its style, an implementation may choose to represent the viewer and style as a single object that supports both interfaces (where retrieving viewer->style() simply returns the viewer itself).

close

void close();

Respond to the user closing the viewer.

first_focus, last_focus, next_focus, prev_focus

boolean first_focus();
boolean last_focus();
boolean next_focus();
boolean prev_focus();

These operations ask that this viewer "move" the input focus inside itself, such as tabbing through the children in the hierarchy. The first_focus and last_focus operations ask focus to enter the viewer at the beginning or end, respectively. The next_focus and prev_focus operations ask to move focus to the right or left sibling, respectively.

handle

boolean handle(in GlyphTraversal t, in Event e);

Respond to the given input event. If the viewer cannot handle the event for any reason, then this operation should return false so that the caller may attempt to find another viewer to handle the event.

insert_first_viewer, insert_last_viewer

void insert_first_viewer(in Viewer v);
void insert_last_viewer(in Viewer v);

These operations modify the viewer hierarchy by adding this viewer at the beginning (insert_first_viewer) or end (insert_last_viewer) of the given viewer's list of children. Inserting a viewer will perform an insertion operation on the parent after the hierarchy has been modified.

insert_next_viewer, insert_prev_viewer

void insert_next_viewer(in Viewer v);
void insert_prev_viewer(in Viewer v);

These operations modify the viewer hierarchy by inserting this viewer before (insert_prev_viewer) or after (insert_next_viewer) a given viewer. Inserting a viewer will perform an insertion operation on the parent after the hierarchy has been modified.

insertion, removal

void insertion(in Viewer v);
void removal(in Viewer v);

Notify a viewer that its list of children have changed. The insertion operation is called after the given viewer has been inserted. The removal operation is called before the given viewer is removed.

link_next, link_prev

void link_next(in Viewer v);
void link_prev(in Viewer v);

These operations explicitly modify the links between sibling viewers. The link_next operation sets the right sibling, the link_prev operation sets the left sibling. These operations should not be called directly, as they do not guarantee structural consistency. The operations are necessary to implement the insertion operations.

lose_focus

void lose_focus(in boolean temporary);

Notify a viewer that it is losing input focus. If the temporary parameter is true, then the viewer will regain shortly, as soon as a transient such as a dialog finishes.

parent_viewer, next_viewer, prev_viewer first_viewer, last_viewer

Viewer parent_viewer();
Viewer next_viewer();
Viewer prev_viewer();
Viewer first_viewer();
Viewer last_viewer();

These operations provide hierarchy information starting from a viewer, returning the parent, right sibling, left sibling, first child, and last child, respectively. If the requested viewer does not exist, such as the right sibling for the last viewer in a list, then the operation returns a nil reference.

receive_focus

boolean receive_focus(in Focus f, in boolean primary);

Notify a viewer that it has acquired input focus. The operation returns whether the viewer accepts the input focus. If the viewer does accept, then it should perform the appropriate operations on the focus object, such as installing key translations. If the primary parameter is true, then this viewer is the lowest level viewer receiving focus. If primary is false, then this viewer contains a descendant that is the new focus.

remove

void remove();

Remove this viewer from its place in the hierarchy. This operation calls the removal operation on the parent before the hierarchy is modified.

request_focus

Focus request_focus(in Viewer requestor, in boolean temporary);

Ask for input focus. Normally, a viewer will call this operation on its parent. The parent, in turn, may propagate the call to its parent. The original caller passes itself as the requestor, and the same requesting viewer is passed if the calls is propagated.

If focus is granted to the requestor, the operation returns the focus object (normally associated with the root viewer of the hierarchy. If focus is not granted, then the operation returns nil.

Transient dialogs normally request focus temporarily by passing true as the second parameter to request_focus. This flag lets the current focus viewer know that it will probably regain focus shortly and might not wish to change its decoration in the same manner as a normal focus change.

ViewerImpl*

class ViewerImpl : public Viewer, public StyleObj

ViewerImpl is an implementation of the viewer interface. Because every viewer has one and only one distinct style, ViewerImpl also implements the StyleObj interface directly. This approach means there is only one object for a viewer and its style.

ViewerImpl also adds some implementation-specific virtual functions for input handling. ViewerImpl implements Viewer::handle by inspecting the event type and calling the appropriate virtual function.

double_click

virtual boolean double_click(GlyphTraversalRef t, EventRef e);

Respond to a button up event that rapidly follows a previous up event. The default is to call release and return the value from release.

drag

virtual boolean drag(GlyphTraversalRef t, EventRef e);

Respond to a pointer motion event while a button is pressed. The default is to do nothing and return false.

focus_event

virtual boolean focus_event(GlyphTraversalRef t, EventRef e);

Handle an event from a focusable device, such as a keyboard. This operation looks at the type of event and calls key_press, key_release, or whatever the appropriate function is.

grab

virtual void grab(GlyphTraversalRef t);

Set up a filter on the display (see Display::add_filter) so that further positional events will be passed directly to this viewer's handle_position_event operation.

handle_position_event

virtual boolean handle_position_event(GlyphTraversalRef t, EventRef e);

This operation looks at the type of event and calls move, press, drag, or whatever the appropriate function is. The chosen function depends on some state as well as the event type. If a pointer button has been pressed, (and is still down), then a motion event will call drag, otherwise it will call move. A button-down event will be ignored if a button is already down, otherwise it will call press and grab. A button-up event will call ungrab and either release or double_click, depending on whether the timestamp of the up event is within a preset threshold.

inside

virtual boolean inside(GlyphTraversalRef t);

Test whether the given traversal state has clipping set that intersects with this viewer. This operation is useful for testing if a positional event is inside a viewer when the positional event is the result of a grab (other positional events will necessarily be inside the viewer, otherwise it wouldn't be given the event to handle).

key_press

virtual boolean key_press(GlyphTraversalRef t, EventRef e);

Respond to a key being pressed. The default is to do nothing and return false.

key_release

virtual boolean key_release(GlyphTraversalRef t, EventRef e);

Respond to a key being released. The default is to do nothing and return false.

move

virtual boolean move(GlyphTraversalRef t, EventRef e);

Respond to a pointer motion event. The default is to do nothing and return false.

other

virtual boolean other(GlyphTraversalRef t, EventRef e);

Respond to an implementation-dependent event. The default is to do nothing and return false.

position_event

virtual boolean position_event(GlyphTraversalRef t, EventRef e);

Handle a event for a positional device, including as pointer motion, button press, and button release. The default implementation does a pick to see if the event is contained in a nested viewer; if so, then the event is passed to the nested viewer. Otherwise, this operation calls handle_position_event.

press

virtual boolean press(GlyphTraversalRef t, EventRef e);

Respond to a button press event. The default is to do nothing and return false.

release

virtual boolean release(GlyphTraversalRef t, EventRef e);

Respond to a button up event. The default is to do nothing and return false.

ungrab

virtual void ungrab();

Remove the filter for this viewer. If the same viewer calls several grabs, then only one filter will be created but it will not be removed until a corresponding number of ungrab calls are made.

WidgetKit*

interface WidgetKit : FrescoObject

WidgetKit provides operations for creating common types of buttons, menus, and scrolling objects, as well as styled frames and labels.

check_box

Button check_box(in Glyph g, in Action a);

Return a check box with the check to the left of the given glyph. A check box is a toggle button: choosing it when already chosen will cause it to become unchosen.

inset_frame, outset_frame, bright_inset_frame

Glyph inset_frame(in Glyph g);
Glyph outset_frame(in Glyph g);
Glyph bright_inset_frame(in Glyph g);

These operations provide a beveled frame around the given glyph. The inset_frame operation uses dark shading in the upper left and light shading in the lower right to make the contents of the frame appear to be recessed. The outset_frame operation reverses the shading to make the contents appear to project out of the frame. The bright_inset_frame operation is like inset_frame but uses a brighter background color and is a thinner frame.

push_button, default_button, palette_button

Button push_button(in Glyph g, in Action a);
Button default_button(in Glyph g, in Action a);
Button palette_button(in Glyph g, in Action a);

Return a button that frames the given glyph and executes the given action when clicked. A default button adds some decoration to indicate to the user that it is the common choice. Unlike a push or default button, a palette button can be chosen.

radio_button

Button radio_button(in Glyph g, in Action a, in Telltale group);

Return a radio button with the given glyph to the right of the flag that indicates whether the button is chosen. A radio button has a telltale group so that only one button in a group will be chosen at any given time.

telltale_group

Telltale telltale_group();

Return a telltale object suitable for use with radio buttons or other groups of mutually-exclusive choice objects.

WidgetKitImpl*

class WidgetKitImpl : public WidgetKit

WidgetKitImpl provides an implementation of the WidgetKit interface that creates objects with a Motif-like look and feel. The bevelled look is slightly enhanced by using several levels of shading instead of a single color change.

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.

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.

handle_event

void handle_event(in Event e);

Respond to an event. A window typically will simply pass these operations on to the main viewer. This operation is normally called by a display object and need not be called directly from an application.

iconify, deiconify

void iconify();
void deiconify();

These operations request that a window be iconified (sometimes also known as “minimized”) or deiconified.

main_viewer

Viewer main_viewer();

Return the top-level viewer inside the window.

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.

screen

ScreenObj screen();

Return the screen with which the window is associated.

style

WindowStyle style();

Return the style for the window.

WindowStyle

interface WindowStyle : StyleObj

A window style provides a type-checked interface to the predefined window attributes.

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 StringObj 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.

icon, icon_bitmap, icon_mask, icon_name, icon_geometry, iconic

attribute Window icon;
attribute Raster icon_bitmap;
attribute Raster icon_mask;
attribute StringObj icon_name;
attribute StringObj icon_geometry;
attribute boolean iconic;

These attributes specify information about a window, called an icon, that is normally mapped when this window is iconified and unmapped when the window is deiconified. If the icon attribute is defined, then it specifies the icon window. Otherwise, the icon_bitmap and icon_mask attributes define the icon appearance. The icon_mask defines the area to be drawn and the bitmap defines the foreground and background components. Pixels that correspond to a one in the bitmap and a one in the mask are drawn with the foreground color. Pixels that correspond to a zero in the bitmap and a one in the mask are drawn with the background color. Pixels that correspond to a zero in the mask are not drawn.

If neither a window nor rasters are set, then the icon_name attribute defines a string to be used as a label on a generic icon. If the iconic flag is true, then calling Window::map will actually map its icon window instead.

title

attribute StringObj title;

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

XtfFresco

class XtfFresco : public FrescoImpl

The XtfFresco class implements the Fresco class interface for interoperability with Xt. An application that wishes to place a Fresco viewer or glyph in an Xt widget hierarchy should create an XtfFresco object instead of a Fresco object.

init_widget, init_widget_glyph

void init_widget(Widget, Viewer);*
void init_widget_glyph(Widget, Glyph);*

These operations initialize an XtfViewer widget to contain a given Fresco viewer or glyph. The XtfViewer widget should be placed in the Xt hierarchy in the normal Xt manner.

XtfFresco_init

extern XtfFresco XtfFresco_init(*
XtAppContext, XDisplay,*
const char name,*
*int& argc, char** argv, Option* options = nil*
)

Return an XtfFresco object suitable for placing Fresco objects in an Xt hierarchy. XtfFresco_init must be passed an application context, display, session name, and command-line arguments.

Index

Index (Cont.)