Fresco is currently under development by an X Consortium working group. A rough draft of the specification should be available within the working group in early '93. Some relevant implementation work has been done, but no schedule has been set for a complete sample implementation.

References

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

[2] P. Calder and M. Linton. Glyphs: Flyweight objects for user interfaces. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Snowbird, Utah, October 1990, pp. 92-101.

[3] P. Klemond. Taking the Bull by the Horns: Investigating Object Linking and Embedding. *Microsoft Systems Journal*, Vol 7., No. 2, March, April 1992

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

[5] M. Linton. Encapsulating a C++ Library. *Proceedings of the USENIX C++ Conference*, Portland, Oregon, August 1992.

[6] B. Martin. The Separation of Interface and Implementation in C++. *Proceedings of the USENIX C++ Conference*, Washington, D.C., April 1991.

[7] Object Management Group, Common Object Request Broker Architecture and Specification, OMG Document Number 91.12.1, Revision 1.1.

[8] J. Waldo. Controversy: The Case for Multiple Inheritance. *Computing Systems 4.2* (1991).

This approach has a number of limitations. The user model is limited to "drill-down" editing. In other words, the embedded data can only be edited via an external dialog, then "drilled down" into the embedding application after the editing operation is complete. In addition, both client and server are limited in the kinds of data than can be passed between applications (or a conversion service must be provided). Finally, the client and server can be separated no farther than the reach of the messaging service.

An approach that has been discussed specifically for X Intrinsics-based toolkits is the use of "embedded shell widgets" to allow applications to cooperate in the presentation of one dialog. This approach attempts to address the need for edit-in-place editing versus the drill-down model. Unfortunately, the other issues raised, including activation, storage, and change propagation, are beyond the scope of embedded shells and must be addressed with additional protocols.

Concurrency issues

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 interactivity.

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 at some level to ensure safe access to data. Creating existing 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.

Conclusions

Fresco will solve a number of problems found in current toolkits. Our focus in this paper is on the problem of distributed embedding; more specifically, how to provide network-transparent composition of user interface components. We have presented the underlying distributed object mechanisms provided by CORBA, including IDL. We have shown how using IDL to define the toolkit API allows distributed access to user interface objects. The Fresco embedding architecture then simply treats the components as traditional toolkit objects, as opposed to semi-autonomous applications.

as the “desktop” address space). These components can be connected into the GUI hierarchy of the embedder, using IDL defined references to objects provided by the embedder. This allows the embedder to control the process without requiring the embedder to have detailed knowledge of the specific components being created.

This approach differs from systems like OLE 1.0[3], which requires the embedder to create a presentation based on some data passed to it by the embeddee (via the clipboard). Fresco will accomplish this in a manner that allows the embedder complete control over the resulting presentation.

The advantage of the Fresco approach is that the embedder need have no knowledge of how to present the embeddee. Systems like OLE require that data be of some common format which all applications know how to present (typically a bitmap image).

Creating Embeddee-Related Editing Dialogs and Menus

Fresco interfaces will allow the developer to express relationships between dialogs versus individual dialog components. Objects supporting these interfaces are provided by the embedder to embeddees, allowing the embeddee to define associations between the principal dialog in which the embeddee is presenting and auxiliary dialogs used specifically for operations on the embeddee.

As a result, full edit-in-place can be supported. The fact that dialogs used to edit the embedded object appear to be associated with the embedder (e.g., when the embedder is iconified, so are the embeddee dialogs) gives the GUI a seamlessness that is not achievable with less robust architectures. Because the objects supporting these interfaces are provided by the embedder, the embedder can take full control of the association process.

Integrating the Embeddee with the Embedder

Fresco defines protocols for propagating dialog specific information among the nodes composing the GUI component hierarchy. These protocols propagate such things as user preference information (colors, fonts, etc.), access to things like window system resources, and the location of kits for the construction of supporting objects.

These protocols are defined using IDL, and developers are free to provide new implementations of these interfaces. Since these protocols are publicly specified in IDL, independently developed GUI component implementations can be integrated into an environment seamlessly. This is critical if we are to support the ability to construct hierarchies which span processes and networks. This capability necessarily supposes that GUI component hierarchies will have nodes that do not share a common implementation.

Other Embedding Architectures

A limited form of embedding can be added to existing toolkits. This can be accomplished by introducing an inter-application messaging service (e.g., Microsoft’s DDE or SunSoft’s Tooltalk) and passing the identity of the source application when data is cut-and-pasted between applications. The identity passed with the data must be something recognizable by the messaging service. When the user wishes to edit the embedded image later, the application can activate the proper editing program via the messaging service, using the identity passed when the data was pasted.

The Problem

Suppose we want to embed a portion of a spreadsheet inside a document. We would like to be able to edit the spreadsheet seamlessly - if the editing operation can be done naturally by having the user directly manipulate the spreadsheet, we'd like to be able to allow this. Additionally, if the editing operation requires additional tools, like palettes, menus, and dialogs, these tools should appear when needed in the same way tools used to edit the rest of the document appear.

We want to provide embedding with the following constraints:

- We should be able to develop the document system without any knowledge of the spreadsheet system. We should not be required to include spreadsheet support in our document system. Similarly, the spreadsheet implementation should not need to know anything about documents.
- The user of the document system should not be required to have an implementation of the spreadsheet system installed in order for the embedding to succeed.
- The document and the spreadsheet should not be required to reside in the same address space or on the same machine.
- The GUI components used to display the spreadsheet inside the document are not expected to be pre-loaded, or dynamically loaded, into the document process.
- The GUI components used to present the spreadsheet inside the document should have the same look and feel as the document components.
- Menu commands available to the document user, such as Print, should propagate seamlessly and correctly to the embedded spreadsheet.
- Changes made to the spreadsheet from other spreadsheet editors should be able to be reflected immediately in the document.
- If the document is saved and edited later, the spreadsheet should still appear within the document, and spreadsheet editing should still be available as before.

The Fresco Approach

Fresco meets these goals in two ways. First, we leverage the OMG CORBA system to handle activation, inter-client messaging, and the other problems of distribution using the CORBA object infrastructure. Fresco does not define a GUI toolkit-specific architecture for inter-application messaging as related to embedding. Communicating between applications is just another form of performing operations on objects. Second, the Fresco embedding architecture is designed to minimize the impact of network delays and failures. While individual components of a GUI hierarchy may be distributed around a network, the need to propagate messages through this distributed hierarchy is minimized.

For purposes of discussion, we will refer to the object being embedded as the embeddee, and the object which contains the embedded object as the embedder. In other words, the spreadsheet is the embeddee and the document is the embedder.

Creating the Embedded Presentation

Fresco defines IDL interfaces which provide the ability to create GUI components in either the address space of the embeddee, the address space of the embedder, or in a third address space (such

We provide basic types supporting an observability mechanism for notifying one or more objects when another object changes. Our mechanism contributes to the separation of user interface objects from application data, providing a way to update the interface when the data changes.

Windowing

This layer provides access to basic window system services, such as mapping window to a display and screen, handling device events, and coordinating with a window manager. Object types defined in this layer include Display, Screen, Window, and Event. A Fresco implementation can build the windowing layer directly on X, though an implementation should also be possible on other platforms.

Rendering

The rendering layer defines individual graphics state objects, such as colors and fonts, and a rendering context analogous to a graphics context in X. Object types in this layer include Brush, Color, Font, Bitmap, Raster, and Renderer.

An important requirement for the rendering layer is to support printing. In particular, a Fresco graphical object should be able to implement a single draw operation that works on either a screen or a printer. The key to printing support is the use of resolution-independent coordinates and calling operations for which both screen and printer implementations are provided.

Embedding

The embedding layer defines the composition mechanisms for building hierarchies of user interface objects. Embedding encompasses presentation, which involves the *physical* composition of user interfaces, and editing, which involves the *logical* composition of objects. For presentation, Fresco defines a type called Glyph that supports layout, drawing, and hit detection. For editing, Fresco defines a type called View that supports input focus management, event routing and translation, attribute propagation, and transcription to persistent storage. Because user interfaces frequently edit graphical objects, we also provide a type for representing graphical data.

An important characteristic of Fresco embedding is the decoupling Fresco objects from X windows. For example, one would like to be able to embed text inside a drawing and use the same view to edit the text as when editing a text file. In the drawing, the text may appear transformed or in front of background graphics. The text view therefore cannot be allocated an X subwindow and cannot update the screen directly. Screen update must be coordinated so that all objects in a given area are redrawn. In the case of the embedded text, the graphics under the text may need to be redrawn when the text changes.

An Example of Distributed Embedding

Distributed embedding is a general term that describes several interesting problems solved by distributed user interfaces. The most commonly cited problem is that of composing a document out of separately developed components - the “spreadsheet inside a document” scenario. This is the one we will use to illustrate the concepts and problems introduced. We will show the approach taken by Fresco to provide support for this capability, and compare this approach to that taken by other systems.

Our InterViews experience[5] has convinced us 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.

Fresco

The goal of the Fresco effort is to provide a "next-generation" window system toolkit with functionality beyond what is available from Xlib or Xt. Specifically, we want Fresco to support the following new features:

- Lightweight objects, such as InterViews glyphs
- Structured graphics
- Resolution-independence (and therefore printing)
- Natural C++ programming interface
- Edit-in-place embedding
- Distributed user interface components
- Multithreading

We are guided in our design of the first four features by lessons learned in InterViews[2,4]. Our embedding architecture has roots in the Andrew Toolkit[1]. Our distribution model conforms to CORBA specifications, using IDL, with distributed user interface specifications based on technology derived from SunSoft's Project DOE. Multithreading minimally means that concurrent applications can safely use the toolkit; in addition, we would like to take advantage of concurrency where possible in the implementation.

The Fresco specification has not yet been finalized, so we present an overview with the caveat that some of the names and details may change. We split Fresco functionality into three layers: *windowing*, *rendering*, and *embedding*. Before discussing these layers, we should present some of the basic types that Fresco defines.

Basic Types

Several types are common throughout the Fresco API. The Coord type represents a position or distance in floating point units of printer's points (one point is 1/72 of an inch). The PixelCoord type is an integer type for pixel-based coordinates.

The Region type defines an area within a coordinate space and an origin as a point of reference. 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.

reference to a remote object will look syntactically identical to a call on a local object. Figure 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.

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.

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. Our experience with InterViews[2] has been that we follow these conventions anyway. We use virtual functions heavily to allow extensibility, and use indirection for data to prevent recompilation of users of a class when the class representation changes.

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 of 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. We retain the term "factory" as a general term for those objects that create other objects as a secondary function. For example, we might refer to an object of type Font as a factory for other Font objects if a Font operation create_new_size returned a new font with the same characteristics except for a specified point size. On the other hand, a layout kit is an object that creates common layout management objects.

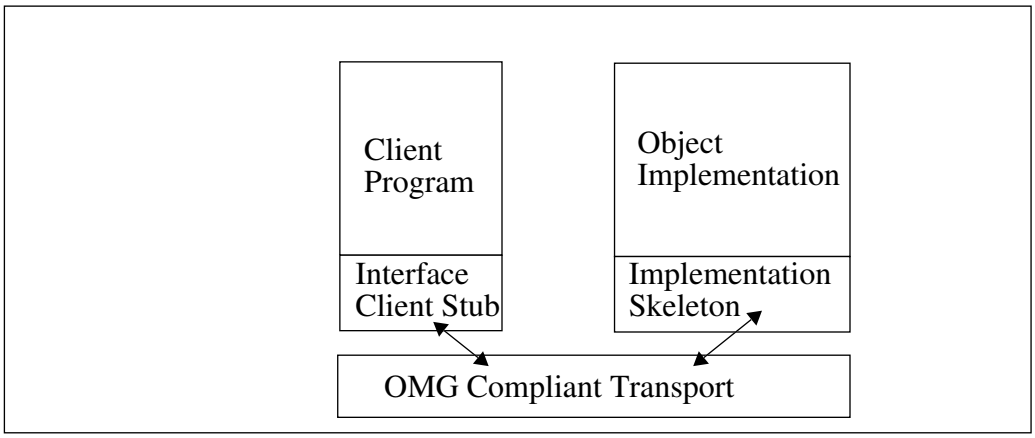

Figure 1. Implementation of Remote Objects in CORBA.

used to describe an object implemented in C, C++, or Smalltalk, in spite of the fact that C, C++, and Smalltalk do not share a common object implementation model. This notion of complete separation of interface and implementation in C++ has been explored in [6] and further discussed in [8].

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.

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

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 {
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;
};
```

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. Currently a standard mapping exists for C, and a mapping for C++ is being reviewed.

Interfaces and Implementations

A fundamental principle of object-oriented design is to separate an object's external interface from the object's implementation and to allow multiple implementations for the same interface to exist at runtime. We have shown how an IDL interface can be mapped to a C++ class and support at least two implementations, one for local and one for remote objects. The reverse direction is also possible; that is, a single implementation can support multiple interfaces at once. An interface does not define the boundary of an object implementation - it merely specifies a type that an object can support.

A strong separation of interfaces and implementations allows us to support multiple languages, even if the languages have differing object models. For example, a single IDL interface can be

In this paper, we describe how Fresco solves the problem of distributed components by relying on the specification of object interfaces in a form that supports both distribution and multiple languages. We first introduce the mechanism for defining interfaces to objects, then present a brief overview of the Fresco programming interfaces. We discuss the use of distribution in the specific case of embedding several remote objects in the same window. We briefly present some of the concurrency issues associated with a distributed environment and conclude with some observations about our approach.

Defining 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)[7] 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.

In the remainder of this section, we present an overview of the IDL features, give some examples, and discuss some of the issues of using C++ as an implementation language. We also discuss the use of IDL for accessing local and remote objects.

IDL

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

Building Distributed User Interfaces with Fresco

Mark Linton and Chuck Price

Abstract

The X Window System provides network-transparent access from an application to a user's display. However, X does not provide any convenient mechanism for distributing components of a user interface across a collection of machines.

Fresco is an X Consortium effort (formerly known as XC++) to define a new set of programming interfaces covering and expanding current X functionality. To allow the distribution of user interface components in a network, we are using the OMG Interface Definition Language (IDL) to specify the Fresco protocol for linking and embedding components. This approach provides a simple, uniform framework for integrating application objects in a distributed, heterogeneous environment.

Introduction

An X Window System user can run an application on a remote machine and direct the application to connect to the user's local display. This functionality allows flexibility in the organization of a site and allows some users to access applications that might not otherwise be available to them.

With user environments becoming more object-oriented by replacing the concept of an application with that of an object, users want the same capability for distribution to be available for individual components of a user interface. For example, one might want to compose a single document containing text, a spreadsheet displaying information from a financial database, and a 3D drawing from a CAD database. We cannot a priori predict whether the databases will be on the same machine as the user, or even the same machine as each other, but the user will want to compose these objects into a single page. X supports the assembly of an entire application that can be run remotely, but currently provides no mechanism for embedding several user interface components in a single window, where some of the components are running remotely.

Mark Linton (linton@sgi.com) is a Principal Scientist at Silicon Graphics.

Chuck Price (price@sun.com) is a Staff Engineer at SunSoft.

Fresco is a registered trademark of the MIT X Consortium.