IMPLEMENTING A GUI

To give you some idea of what's involved in making an interface work, here is *part* of a description of the development of the software to support an X Windows graphical interface. Notice the insistence on standards and design, and the final remarks on the efficiency. The original paper[IMP20] continues with descriptions of measures taken to speed up the interface to practical levels.

The DECwindows user interface language (UIL) and resource manager (DRM) are the tools which allow form and function to be separated. UIL is a specification language that describes the initial state of a user interface, i.e., it describes the objects used in the interface and the application callbacks to be invoked when the interface changes state. DRM provides the application with a run-time library for accessing the compiled UIL descriptions. DRM, therefore, builds the run-time structures necessary to actually create the user interface during execution of the application.

Conformance to the XUI Style. The toolkit had to support XUI style at a detail level in both look and feel. Supporting the look primarily meant setting default values for the many graphic aspects of a widget, such as the border width of a push-button. Supporting the feel meant establishing tables that translate user events, such as button press, into widget action, such as highlight. Defining the widgets that compose the toolkit was based on partitioning the XUI style look and feel demands into logical pieces and on predicting application needs.

Although a widget could have many customizable attributes, all of which could be controlled by the application, we wanted to make it easy for an application developer to design and implement a windows application that conformed to the XUI style. A widget should, by default, select conforming values for any attribute the application could have but did not set. Therefore, we implemented a default look and feel that matched the precise user interactions defined in the style guide and the precise graphic design that was defined for XUI by our graphic artists. However, we also made the widgets as flexible as possible. Although widgets defaulted to the XUI style, the customization methods inherent in the intrinsics, e.g., resource and translation management, could be used to customize a widget to another style. This design philosophy helped give applications a consistent look and feel but did not constrain user interface innovation.

Further, we decided to structure the set of widgets based upon the object's function as seen by the application's developer rather than as seen by the application's user. An example is the use of buttons in menus and dialog boxes. Both menus and dialog boxes contain buttons that directly invoke application actions (i.e., pushbuttons). However, the graphical appearance and user invocation syntax of the buttons is different depending upon whether the button is placed within a menu or a dialog box. The toolkit, however, presents only one push-button class to the application programmer. The buttons are dynamically configured based upon the environment in which they are placed. Thus, an application developer can change the environment of a widget without changing any other code.

Conformance to Standards The DECwindows program was intended to be based on MIT's X Window System standard. Therefore, the toolkit had to be based upon the standard X toolkit intrinsics. It was a challenge to do so because the toolkit and the intrinsics were designed, implemented, and standardized in parallel.

The standard language bindings for the intrinsics were designed for the C language. However, we were mindful of the requirements of other languages and attempted not to prohibit other language bindings from being possible. It is a well known technology to provide multiple language bindings, in the form of header file definitions and entry point names, for a single set of run-time routines. Digital used this approach in providing VAX procedure calling standard bindings for Xlib, the intrinsics, and the toolkit.

A special problem arose in defining the bindings for the intrinsics because the intrinsics would call back into the application code to provide notification of a user action such as a button press. The intrinsics, however, has no knowledge

of the language used in the called procedure. Therefore, we had to restrict the parameter passing mechanism in callbacks to the set that could be understood by most languages. Parameters to callbacks are passed by a reference mechanism as opposed to a value mechanism that is commonly used when calling C procedures.

Initial Implementation

The initial development of the toolkit presented the software engineers with a number of challenges. The major challenge was to develop several different layers of the architecture at the same time. Further, none of the layers had proven suitable for their designed task. Therefore, it was difficult to predict the performance characteristics of the layers.

To reduce the inherent risks of this situation, we established a development plan that allowed major functionality to become available for serious application development early in the product development cycle. We then used the applications to determine whether the goals of the DECwindows program, in general, and the toolkit, in particular, were being met.

Intrinsics and Toolkit Codevelopment

Our plan to design and implement the toolkit and the intrinsics simultaneously was further complicated by the fact that the layers below the intrinsics, i.e., Xlib and the X protocol, also were being changed Some of the changes were driven by the needs of the toolkit and intrinsics. Others were due to the lack of maturity of the Xll protocol. Because of these changes, we had to respond to a number of releases of the lower layers of the architecture.

The intrinsics design was changed several times during the first year of development as a result of two major factors. First, the problems and deficiencies of the intrinsics and the toolkit became apparent when we began to write serious applications. Second, other companies became more involved in the definition of the intrinsics standard. Therefore, we had to work with a formal process of proposing and reviewing changes to the standard and negotiating the inclusion of those changes with engineers from MIT and other companies. As each of these changes then became standardized, each would, in turn, cause changes in widget code, which caused changes in application code.

Each time a significant change in a layer of the architecture occurred, all of the layers above it had to change in a coordinated manner to provide a consistent development environment. Much time was spent in planning the management of these changes. Also, the changes necessitated rewriting code that had already been completed. We had not accounted for the time taken by these unanticipated changes in our original development plans.

Performance

Performance was the most serious problem encountered during early implementation. The first internal field test of the DECwindows software provided fairly complete functionality for the toolkit and the layers below it. However, the DECwindows developers, including the toolkit team, had devoted nearly all their efforts toward developing the functionality and postponed measuring, examining, and improving performance. Now that we had an existing collection of applications, serious work could begin on performance.

In the initial measurements of the system's performance against the goals described earlier, even the worst-case goal was missed in many areas. Early investigation also indicated that the performance problem did not seem to be localized. That is, the problems could not be isolated to a single component in the architecture. With this information, a task force with members from most DECwindows development groups was convened to determine where the performance problems were and what could be done about them.

We quickly learned that we could not determine where the performance problems were as easily as we could have in the typical engineering environment to which we were accustomed. Our experience was in evaluating isolated layered applications, such as compilers, and individual primitive operations, such as system calls. However, the user interface actions that were being measured involved the

issuance of possibly hundreds of X primitives, and the interaction of up to three separate processes (i.e., the application, the X server, and the window manager). Although the usual evaluation tools were of some help, additional tools were needed.

Existing tools, such as the VAX performance and coverage analyzer on the VMS system, were used to locate performance bottlenecks. These tools helped but did not provide the level of improvements that were necessary. A number of internal tools to aid in X performance analysis were used to supplement the traditional tools. These X performance tools included:

- An instrumented X server that counted the resources an application requested, such as graphic contexts, windows, and pixmaps
- A set of tests that measured the performance of Xlib primitive calls
- A protocol monitor that recorded the inter- actions between an application and the X server
- A tool that recorded the dynamic memory allocation of an application

By using these tools on the applications, a large amount of data was collected and evaluated. Some of the more important observations were:

- Applications were using more server resources than anticipated. The most common overuse was windows because each user interface object had its own X window. However, application use of other resources, such as graphic contexts, pixmaps, and fonts was also at a higher level than anticipated.
- Applications were using too much memory. The object-oriented design of the toolkit and the Xll Style Guide encouraged applications to use hundreds or thousands of widgets, and each widget was then using about 600 bytes of memory. A number of X toolkit intrinsics features, such as resource management and translation management, also used a large amount of memory.
- Application start-up was slow. Loading the large programming libraries, connecting to the X server, and creating widgets were some of the principal functions that slowed application start-up.
- The Digital Xll server design was optimized for graphic primitives, e.g., line and text drawing. The performance of these operations was very good. However, in optimizing the graphics aspect, the design had traded performance in windowing operations, for example, window creation and mapping. The analysis showed that windowing operation performance was important throughout much of the direct manipulation style user interface.
- Many context switches existed between the server and the application during time-critical operations. Even simple applications required the coordinated efforts of the application, a window manager, and a server. Careful analysis and planning were needed to minimize the communication traffic and switching among the processes.
- The basic round-trip time between the server and the application using the DECnet transport was higher than anticipated. This factor increased the need to reduce the amount of communication traffic between the application and the server.