Experience with the Cedar Programming Environment for Computer Graphics Research

Richard J. Beach
Computer Science Laboratory
Xerox Palo Alto Research Center

Abstract: Cedar is an integrated programming environment for building experimental computer systems. The environment consists of a well-coordinated collection of tools and packages and a language that encourages and enforces their coordination. Cedar incorporates a device-independent imaging model for presenting all information and relies on input interaction techniques to control the environment.

The computer graphics research accomplished with Cedar covers a broad range from basic computer graphics techniques to various design tools with sophisticated graphical interfaces to graphic-arts-quality typeset documents with imbedded color illustrations.

Our experience with Cedar confirms the benefits to a software researcher of shared module interfaces, compiler type-checking, automatic storage management, interpretive graphics programming languages, and device-independent imaging models. The 'object-oriented' programming style and the integration of graphics within Cedar below the screen window manager and document formatter have led to more effective software designs than those designed with traditional languages and programming environments. Cedar provides a software research environment where one quickly integrates the work of others and redesigns one's own work after experimenting with it in a functional prototype.

Résumé: Cedar est un environnement de programmation destiné à la construction de systèmes informatiques expérimentaux. Il est constitué d'une collection proprement structurée d'outils et de modules de programmes, ainsi que d'un langage encourageant et supportant une telle organisation. Cedar utilise un modèle graphique indépendant du dispositif d'affichage pour présenter toutes les informations, et s'appuie sur des techniques d'entrée interactive pour controler l'environnement.

La recherche graphique réalisée avec Cedar couvre un large domaine, depuis des techniques élémentaires de graphique par ordinateur jusqu'aux différents outils de conception dotés des interfaces graphique sophistiquée, en passant par la composition de documents de très haute qualité avec des illustrations en couleur.

Notre expérience avec Cedar confirme le bénéfice qu'un concepteur de programmes peut tirer des outils suivants: des interfaces entre modules partagés, un vérificateur de types au niveau du compilateur, une gestion de mémoire automatique, des langages interprétés de programmation graphique, et enfin des modèles d'images indépendants de l'affichage. Le style de programmation par objet et l'intégration d'outils graphiques en Cedar à un bas niveau (inférieur à la gestion de fenêtres ou au compilateur de documents) ont conduit à des programmes plus efficaces que ceux conçus avec des langages et des environnements de programmation traditionnels. Cedar fournit un environnement de recherche à l'intérieur duquel il est possible d'intégrer rapidement les réalisations d'autrui, et de modifier son propre travail après avoir expérimenté avec plusieurs prototypes.

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques; D.2.6 [**Software Engineering**]: Programming Environments; D.3.3 [**Software Engineering**]: Language Constructs; H.0 [**Information Systems**]: General; I.1.3 [**Computing Methodologies**]: Languages and Systems; I.3.2 [**Computer Graphics**]: Graphics Systems; I.3.6 [**Computer Graphics**]: Methodology and Techniques; K.8 [**Personal Computing**].

Introduction

This paper outlines experience with the Cedar programming environment and its support of computer graphics research. The first section provides an overview of Cedar: its origins and goals, some important language features, the integrated environment, and some productive software development features. The second section outlines several computer graphics research and development projects undertaken in relation to the Cedar project. These projects cover a broad range from basic computer graphics techniques to interactive design tools to graphic arts quality typeset documents with imbedded illustrations. Subsequent sections describe our experiences with the language features and with producing software in the Cedar environment.

Cedar: An Environment for Experimental Programming

Cedar is a programming environment for developing experimental programs. We define *experimental programming* to mean the production of moderate-size software systems that are usable by moderate numbers of people in order to test our ideas about such systems. The ambitious project of building such an environment was undertaken by the Computer Science and Imaging Sciences Laboratories at the Xerox Palo Alto Research Center (PARC) over the past five years.

The requirements for an experimental programming environment [10] evolved from an assessment made of the three computing environments in use at PARC in 1978: Smalltalk [13, 14, 16], Interlisp [33], and Mesa [11, 23]. This report prioritized a long catalog of capabilities sought in a programming environment. Many requirements focused on system issues such as object management, static type-checking, memory management, abstraction mechanisms, fast turn-around for program changes, run-time efficiency, and software development techniques. Another group of requirements dealt with information presentation through formatted documents, scanned or synthetic graphical objects, and uniform screen management.

The resulting Cedar environment supplies an integrated software development facility that incorporates a graphics package, a window manager, a structured editor, document preparation tools, and a refined set of software development tools.

The PARC Computing Environment

Computing Resources

Cedar runs on the family of Xerox Scientific Workstations. The most powerful of these is the Xerox 1132, the Dorado [18, 26] which is the personal computer available to all computer systems researchers at PARC. This family of high-performance personal computers shares a common hardware architecture and can execute all of the PARC computing environments in addition to Cedar. All PARC workstations are connected via Ethernets [22] to each other and to network services.

Imaging Devices

Each Cedar workstation has a high-resolution large-format black-and-white display. The display is bit-mapped with 1024 by 808 pixels resolution and is large enough to present the formatted information of almost two 8.5 by 11 inch sheets of paper side by side. Additional color displays can be configured by software in several resolutions and pixel densities. Common configurations are NTSC-compatible resolutions of 640 by 480 pixels at 8- or 24-bits per pixel, and a high-definition television resolution of 1024 by 768 pixels at 8-bits per pixel.

Each workstation provides a keyboard and mouse pointing device [20]. Additional input devices such as a digitizing tablet or a five-finger keyset can be connected to the workstation. These input devices are remarkable since the workstation can sense any key transition. This permits n-key rollover and chording actions. For example, pointing actions can be modified by depressing the CONTROL or SHIFT keys, or by the frequency or duration of key clicks.

No special graphics hardware is required for Cedar graphics. The only microcode assist is a BITBLT operation [17] equivalent to RasterOp [25]. The memory for the graphic displays is allocated from the Cedar virtual memory and accessed through display control blocks. Obviously one necessary feature of the workstation architecture is a high memory bandwidth to support these displays. The cursor has been implemented both in microcode, using a 16 by 16 bitmap for the black-and-white display, and in software, providing an arbitrary full-color cursor for the color display.

Hardcopy devices are made available through the network for shared access as spooled printers. Raster printers of various resolutions are used, ranging through Versatec color plotters, medium resolution Xerox product laser printers, to experimental high resolution typesetter quality printers. Using color separations printed on these laser printers, we can generate high-quality color images.

Network Services

Cedar both accesses network services on behalf of its users as well as relying on the Ethernet [22] for its own operation. For example, Cedar users are authenticated through the use of distributed databases called Grapevine registries [4]. Electronic message traffic among Cedar users and other workstations anywhere in the Xerox Research Internet [6] is transported by Grapevine protocols [31]. The Internet spans several thousand Xerox workstations throughout the world. There are in fact two electronic mail message systems implemented in Cedar, a file-based message manager and a more elaborate one that relies on the Cedar database server [7, 8] for the organization of message sets.

The workstation file system is properly thought of as a local cache of remotely stored files. Thus, the Cedar user will retrieve files stored anywhere in the Internet. Users normally operate without regard for the amount of free disk space on the local workstation, since remotely stored files can be flushed from the local cache to make space available. Users depend heavily on version control tools [30] to manage related sets of files, for example to bring into the cache collections of files in a software package or to store changed versions back onto a remote file server.

Several interesting research projects based on Cedar rely on these network services. Among them are transmission of voice information over an Ethernet [12, 32] and distributed computing protocols [5]. The latter project defines a protocol for workstations or servers to cooperate remotely over the network using the familiar concept of a procedure call with arguments. This remote procedure call mechanism permits experimental distributed services to be designed and first implemented within a local workstation environment then to have services distributed via the network.

Cedar Language Features

The Cedar language is an extension of Mesa [23, 11]. Both Mesa and Cedar are strongly-typed, statically-checked languages in the Pascal family. Mesa added the concepts of explicit module interfaces, with type checking across module boundaries, exception handling, lightweight processes, and monitors. Further Cedar extensions provide automatic storage management, safe pointers, immutable strings, lists, atoms, and object-oriented programming support by delayed binding.

Module Interfaces

The module interface concept is a 'programming in the large' mechanism for subdividing large programming projects. Modules come in two flavours, *interface definitions* and *program* modules. A *client* program *imports* the interface definitions module to access the procedures and data types of an abstraction. An *implementor* of a module interface *exports* the program code that will execute when a procedure is called. The module interface establishes a contract between the client program and the implementor program. Once the interface has been compiled, it establishes in writing (with compiled symbol table and version stamps) the agreement between those programs. The binder and loader further verify the correct use of an interface by comparing version stamps of the imported and exported modules. Using software development and version management tools based on these interface concepts, we undertake extensive changes to experimental software with relative ease.

Automatic Storage Management

Providing automatic storage management in a strongly-typed language results in several benefits. The foremost benefit is eliminating the *concern over ownership* of shared objects. When a module interface returns a new object (a pointer to some newly allocated memory space) there is certainty as to who owns the space: it is neither the client nor the implementor programs; it is always the storage management system. The client program may use the pointer as it sees fit, pass it on to another client, retain it, or dispose of it. Since the storage management system can discover what storage is accessible, only when no remaining valid references to the object exist will the storage be reclaimed. A secondary benefit is the *simplified exception handling* in modules using allocated storage; no action to dispose of allocated storage is needed since it will be reclaimed automatically.

Object-oriented programming support

Another benefit of the Cedar storage management scheme is the runtime type mechanism and its support for object oriented programming. The storage manager uses a type table to identify pointers during garbage collection. Exposing this typing information to the programmer permits the language to support generic pointers without violating any type checking. An object may be declared to contain a generic pointer, called a REF ANY in Cedar, with only a few generic operations possible such as assignment or passing it as a procedure argument. When the generic pointer is to be used to reference data, first it must be narrowed from a generic to a specific type of pointer. The Cedar NARROW operation uses the runtime type mechanism to verify the expected type specification and produces either a pointer of that type or an error.

Thus Cedar objects are a source of extensibility. A Cedar object that contains a generic pointer to *client-specified data* can be customized by a client without violating the type checking or obliging the interface definition to be changed. The Cedar language also supports an object-message programming style by providing an object instance, message, parameters format.

Integrated Environment

Cedar programmers are concerned about effective user interfaces. Graphical techniques exist at a low level in the Cedar environment to ensure that good visual feedback techniques are possible, that good typography is available with a selection of fonts and embellishments, and that pictures can be used whenever appropriate. User input interactions through the keyboard and pointing devices are supported and connected to the visual feedback mechanisms. These techniques are more effective when they are uniform throughout the environment.

Uniform User Interface

The text editor and window manager provide good facilities for dealing with text and interaction. Most new applications extend them rather than present different user interfaces. Furthermore, when a new consensus develops within the laboratory, the expectation is that these pivotal packages will be changed. When new interaction techniques are implemented, they eventually augment the window manager interface.

Generally, if a client program wants to deal with text interaction, a text window backed by the text editor will be provided. Client programs can deposit text in the window for display or examine text to obtain user-supplied options or parameters. Thus the same editing skills and actions for editing documents are useable in the user interface of this new program. When a client program needs to provide a menu or control panel, the window manager supplies the interfaces to lay out a menu and to handle the input events on behalf of the new program. Thus the same cursor positioning skills and feedback mechanisms will transfer to the operation of this new program.

Embedded Graphics Package

A key factor in the effectiveness of our user interface is visual interaction. The graphics package resides at a very low level in the Cedar system architecture, above the virtual memory, storage manager, and file system, but below the window manager, text editor and other applications. The window manager has access to the graphics package for presenting text in various fonts, for drawing lines around window borders and headers, for feedback cues by changing colors, and for drawing iconic pictures. In particular, the window manager can use graphics clipping regions for restricting client window refresh procedures. Embedding the graphics package at this level provides graphics facilities to all the clients of the window package as well.

Window Manager and Text Editor

The window package provides the essential abstraction of a virtual display, keyboard and pointing device for each program. The window manager allocates screen real estate to open windows using two columns and collects icons for closed windows at the bottom of the display. Each window supplies hints about its preferred size when it is open. Operations common to each window such as opening, closing, or adjusting the size hints for a window are implemented by the window manager. Operations specific to a client such as repainting the window contents or handling input events are implemented by clients of the window package.

The text editor is a client of the window package and implements a document window class. Each document window is an instance of that class and uses the same text editor implementation but with document-specific data. Menus and buttons are two more examples of window classes. Each button has the same event handler but a different command name and action procedure. Window classes may contain other window classes nested within them, enabling a window designer to build on existing software and to extend the uniform user interface easily.

Recall that workstations may have both a black-and-white and a color display. The device-independent graphics package permits the window manager to treat both displays equally. In fact windows may be moved from one display to another and the cursor slides smoothly from device to device. The user can declare whether the color display is placed to the right or left of the standard black-and-white display.

Software Development Productivity

Creating experimental software is a major goal of the Cedar programming environment. Producing software quickly is essential. Concurrency, a large virtual memory, distributed development and a large collection of preprogrammed packages and tools contribute to the productivity of Cedar programmers.

Concurrency and Lightweight Processes

The normal way to use Cedar is to do several things at once. Multiple windows display documents for editing, programs and tools operate in parallel without preempting your attention, and several things can happen in the background such as the posting of timely reminders or the automatic retrieval of new mail messages. The lightweight process mechanism [19] in Cedar supports this concurrency. Interactive user interfaces and graphics applications benefit from cheap process creation and synchronization functions. Processes abound in Cedar for controlling input and output devices, repainting windows with a process for each window, and executing programs with perhaps a new process for each menu button activation.

Virtual Memory and Runtime binding

A large virtual memory was one of the priority requirements for a new programming environment. Cedar provides a virtual memory as a single 24-bit address space. Since all programs running in Cedar share this virtual memory space, they coexist together and can more easily communicate and depend on shared interfaces. The concurrency of multiple processes is enhanced by this ability to switch between processes without loss of state within the virtual memory. Applications built on top of Cedar further extend this integration resulting in a very broad spectrum of facilities built independently that appear coordinated and designed together.

Distributed development

Connecting all the Cedar workstations to a ubiquitous network creates a community of developers. Sharing software packages, communicating insights, and reporting problems are all easily accomplished over a network with suitable protocols and servers. Our remote file system coupled with good version control tools ensures that consistent sets of files can be shared under appropriate access controls. This encourages researchers to build on the work of others, to share their packages, and to repair code for clients, especially when the compiler and binder help by checking the consistency of module interfaces and versions.

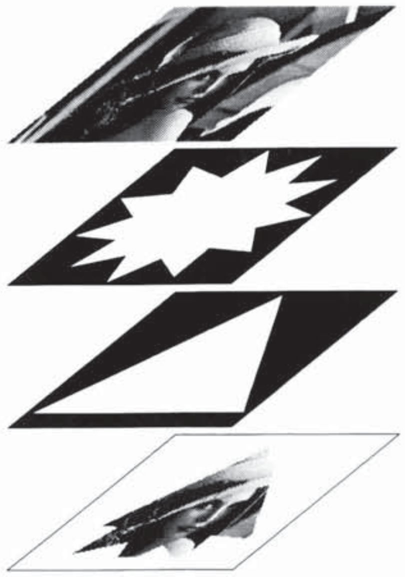
Catalog of Packages & Tools

As Cedar matures, the catalog of packages continually grows. Basic system interfaces provide access to the virtual memory, runtime type system, file system, and the graphics package. Additional packages exist for symbol tables, network communication, compiling and binding, timing, database storage. Built on those interfaces are the window manager and the text editor, followed by software development tools, document preparation tools, mail systems, illustrators, games, and so on. Generally, each application package, in addition to a user interface, also provides a program-accessible interface that permits integration into some further development project.

Computer Graphics Research

The computer graphics research accomplished in the Cedar environment spans a very broad range. Basic computer graphics algorithms are incorporated into the Cedar graphics package [34] that has a device independent and resolution independent imaging model, shown in Figure 1. A language for bitmap manipulation [15] was studied with a prototype implementation in Cedar. Interaction techniques using cursors, menus, buttons, and sliders are incorporated into the

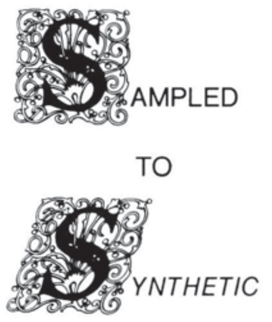
Figure 1. The imaging model used in the Cedar graphics package [34] includes an imaging source (top) passed through a mask (top middle) and clipped to a region (bottom middle) to produce the final image (bottom).



window package. An experimental debugging package using graphics to display data structures was created on an early version of Cedar [24].

Several illustrators are available for creating synthetic graphics. The Griffin illustrator [1] has been converted from an older Mesa implementation. With automatic storage management and the Cedar file system few restrictions on image complexity remain in Cedar Griffin, as shown in Figure 2. The spline outlines for the elaborate 'S' were computed from a scanned image using curve fitting techniques [27] that we are applying to font generation techniques. Another illustrator uses constraints on points, lines and curves to establish the geometry of a line drawing. Two bitmap editors are available to create iconic pictures and to tune bitmap fonts. Illustrators and design tools for integrated circuits and VLSI layouts [28] are in daily use.

Figure 2. The 'Fancy S' was scan converted from artwork and fitted to splines with approximately 850 control points. The curve fitting algorithm first determined an outline contour of the sampled image and then chose spline control points to efficiently and smoothly represent the outlines synthetically. The Griffin illustrator took the outline representation and applied various transformations.



Three-dimensional solid modelling and rendering algorithms are being implemented using Cedar. Solid shaded objects can be created and rendered with the SolidViews illustrator package [3]. A new texturing algorithm for rendering anti-aliased images was implemented using summed area tables [9].

High-quality document formatting and documentation graphics are important research areas within PARC. The text editor provides a sophisticated document structure for editing and formatting. This structure has been extended to accommodate illustrations [2]. Digital printing techniques are being investigated for producing halftones, color, and black-and-white images on laser printers. Color image processing, the calibration of color imaging devices, color specification models and the interrelationship between various models are also being explored within the Cedar environment.

Experience with the Cedar Language

As described earlier, the Cedar language evolved from Mesa [23, 11] which has been referred to as "industrial strength Pascal." Mesa is the implementation language for the Xerox Star office workstation while the Cedar programming environment is implemented in the extended Cedar language. Both are strongly typed languages with module interface specifications, while Cedar provides automatic storage management and supports an object-oriented programming style.

Strong Typing

The Cedar language provides static checking of data types at compile time both within modules and across module interfaces. For example, the Cedar Graphics package [34] interface defines several abstract data types for graphical objects, such as vectors, rectangular boxes, colors, texture patterns, paths, device objects, fonts, and procedures for graphical operations such as transformations and clipping. Clients of the graphics package obviously must specify the appropriate data types as arguments to graphics procedures. Generally this rigor eliminates oversights and coding errors before compilation is successful, so much so that Cedar programmers tend to rely on the compiler to catch their mistakes. When a module compiles correctly, many programs run correctly on the first attempt. In designing experimental programs this saves both time and effort in testing software.

Providing all the necessary type specifications can be a tedious overhead. A few programming aids that integrate with the editor relieve some of the coding overhead. The text editor knows how to expand abbreviations, and programmers frequently use a collection of abbreviations that expand into templates for Cedar program structures and type declarations. The editor deals with structured documents composed of a hierarchal tree of nodes, each node being a paragraph or heading. Cedar programs are represented as a forest of document trees, one tree for each procedure or declaration, with subtrees for nested control structures or record fields, and nodes for each statement or field declaration. Most coding is done by filling in templates or by copying from elsewhere. Long names are easy to use; once entered, they may be quickly copied from their declarations.

For example, the following code fragment contains one node per line, and nested nodes are children of the node above. Appropriate typography is used throughout Cedar to make program code more readable. Types and procedure names are bold; keywords are small capitals; comments are in italics. The compiler sees only the uncommented text without any of the typographic embellishments.

```
ColorTag: TYPE = {rgb, stipple};
    An enumerated type with for two color schemes

Color: TYPE = RECORD[
    A variant record based on the ColorTag
    spec: SELECT tag: ColorTag FROM
        rgb => [red, green, blue: [0..256)], -- 8 bits of intensity
        stipple => [ARRAY[0..16) OF BOOLEAN], -- a 16 bit pattern
        ENDCASE];
```

```
ColorToIntensity: PROC [color: Color] RETURNS [intensity: REAL]
        = {
    Compute the monochrome intensity of a color
    rgb uses NTSC luminance: stipples use black percentage
    WITH c: color SELECT FROM
        rgb => {
            i: REAL;
            IF c.blue=c.red AND c.green=c.red THEN
                i ← c.red
            ELSE
                The NTSC luminance formula:
                i ← 0.30*c.red+0.11*c.blue+0.59*c.green;
            intensity ← i/255.0 };
        stipple => intensity ← StippleToIntensity[c];
        StippleToIntensity is defined elsewhere.
        ENDCASE => intensity ← 0;
    RETURN[intensity];
    };
```

Module Interfaces

To understand a system as large as Cedar, you need an effective chunking mechanism. The module interface defines the data and procedures for an abstraction. In the graphics package, there is the abstraction of graphical objects and procedures to transform and display them. In the window manager, it is the virtual screen abstraction of a window, its class definition and procedures to manipulate windows. Such abstractions occur frequently in Cedar and they increase the number of packages used in experimental software.

The Cedar compiler and binder both check types and versions across module interfaces. All coding syntax errors and incompatible version dependencies are eliminated prior to execution. Thus considerable software revision is undertaken with confidence in the discipline of satisfying the interface specifications. More discussion of module interfaces appears below in the sections on automatic storage management and interface evolution.

Automatic Storage Management

A module interface can create objects without concern for who will own the newly allocated space. Neither the client nor the implementor of an interface own the space; it is always the storage management system. Only when no possible reference to an object exists will the object be reclaimed. Upon reclamation, the Cedar garbage collector provides *finalization* of storage, permitting an implementation to process the object prior to its being discarded. For example, we use finalization to implement font caches, where several client programs may be using the same font and when the cache is about to be reclaimed because the font is no longer in use, we get the chance to close the font file and free its buffers.

The Cedar garbage collection [29] is designed to minimize its impact on runtime efficiency. The garbage collector reclaims storage incrementally and operates as a background activity concurrently with other operations. Microcode support speeds up pointer update and reference counting operations. Programs are generally designed without regard to the intrusion of the garbage collector. However time-critical portions of interaction techniques and server programs are carefully crafted to avoid frequent allocation of storage; the use of preallocated caches is often sufficient. Performance monitoring tools [21] can measure the allocation activity of a running program to help in tuning it.

Object Oriented Programming Style

Objects in Cedar are record definitions with associated procedure operations included as procedure variables. Objects can include client-specified data to permit clients to customize object classes. When an object contains a collection of procedure variables for object operations, clients can replace all or a subset of those operations for alternative implementations. For example, the graphics package interface provides a display context object and a graphical operations object. Implementors of a new graphical device, such as a color display or a laser printer, provide alternative implementations of the display context routines. All the graphical transformation, clipping, and reduction algorithms use the device operations from the current context, extending the power of the graphics package in a device-independent way. Graphical display lists are implemented by providing an alternative graphical operations object in which each operation records its arguments on a list. We use these object-oriented programming schemes as an architecture for extending the graphics package to handle new devices or new functionality.

The window manager also provides an extensible interface for clients to design their own window semantics. Clients of the window package create their own window class, provide any client-specific window operations, and register it with the window class mechanism. Client-specific operations might include repainting the window contents when the window is opened or scrolled, or destroying private data structures when an instance of this window class is destroyed. As members of the window class, all generic window operations still apply such as opening, closing, moving windows, and adjusting the size of windows.

The typesetter knows how to format text files, but it has no direct knowledge for incorporating illustrations or other nontextual matter. However, the document is formatted by using a formatting object that has two operations: layout and paint. The layout operation receives a node from the document and returns its formatted dimensions. After the document is laid out, the paint operation receives the document node with positioning parameters and renders the information into an output object. Output objects are usually a printer file although other objects exist for previewing the formatted output on a display. A key feature of such a structure is its extensibility. New formatting objects for illustrations, tables, or photographs are easy to build and they can incorporate text within them by recursively invoking another instance of a text formatting object. We used such a scheme in the TiogaArtwork documentation graphics experiment [2]

Software Development Experience

Productivity

The text editor plays a major role in enhancing our productivity. Pieces of programs, identifiers, declarations, control structures are more easily copied error-free or inserted from abbreviations than typing everything. Programmers feel so confident of their abilities in this environment that they print very few listings. For example, the Cedar Nucleus (virtual memory, storage management, file system, initialization code) was just rewritten by a small group of programmers. Code was copied, retyped, compiled, edited, and shared without listings. Our facilities for browsing program modules and intermodule references surpass the value of paper listings. Frequently used facilities include find the next occurence of the selected word or phrase, find the definition of the selected identifier or procedure, open a window on the selected module interface or its implementation, position the window to the selected compiler error, display only the top level of the document tree, search for the surrounding matching brackets.

Summer Intern Projects

One measure of productivity is to observe the quality and quantity of software produced by people exposed to Cedar for the first time. Xerox hires summer research interns from graduate school with good programming skills but no Cedar experience. When they arrive, they are confronted with the very large Cedar programming environment. Even though the documentation is fragmented and uneven in quality, these students have acclimated well and produced substantial software projects during their visits.

Here is a sample of intern projects. The IconEditor is a simple bitmap editor for creating and modifying collections of small picture icons and is now part of the Cedar system release. The ShowPress package displays printer files on the screen by decomposing the printer file format and relying on the graphics package to display text, line art, and scanned images (photographs) as they would appear on the printed page. ShowPress is now part of the Cedar release and is integrated with the typesetting tool. A prototype MenuPackage investigated textual definition of a new menu layout and semantics for our window package. Parts of this design will be folded into a window package redesign. The ColorTool is an experimental user interface for several color models (red green blue, hue saturation lightness, hue saturation value, color naming system, CIE). A client-extensible slider mechanism was created and is now part of the window package. The PathTool is an interactive editor for trajectories composed of lines and curves. Incremental refresh techniques using concurrent processes permit the rubberbanding of complicated spline outlines. TiogaArtwork [2] incorporates illustrations into the typesetter and integrates the text editor, graphics package, typesetting software, graphics interpreter, and the formatting style machinery. SolidViews [3] provides both an interactive package for constructing three-dimensional solid objects and a rendering package for producing shaded colored views of solid objects.

These projects were implemented by six interns during one or two summer visits totaling approximately one and a half work-years of effort. Not all interns are as successful or productive. Motivation is rarely a problem, but lack of appreciation for the rich Cedar language and the concurrency available in Cedar can lead to frustration. Interns who do well have systems programming experience; generally they appreciate that previous difficulties they had in systems projects are eliminated with automatic storage management, exception handling, more general data typing, and object-style programming.

Certainly integrating editing with concurrent execution suits most interns. The ability to change your focus from debugging a program to editing several modules to reading mail to managing the file system to investigating documentation *without losing any state in the suspended activities* is a crucial feature cited by most interns.

Interface Evolution

Interfaces are the currency of shared development projects. They represent the contract between client and implementor. With interfaces in place, a client can design his application independent of the implementors. When errors are discovered or should performance problems dictate a different implementation, these can be corrected without changing the interface or affecting the client software, except to notify clients to reload the revised implementation of the interface.

However, the evolution of interfaces themselves does affect a client program. Upward compatible changes, such as adding new data types or procedures to an interface, require only recompilation of the client programs so that the expected versions will match with the new implementations. The compiler and binder enforce these version matches to prevent sloppy software management. Incompatible interface changes require editing of client programs, such as revising procedure call arguments, modifying data type declarations, or changing names. The checking by the compiler ensures that necessary changes have been completed.

Two examples of changes to Cedar demonstrate the value of interfaces and the checking by the compiler and binder. The Cedar graphics interface made paths and trajectories explicit objects, whereas previously they were defined implicitly by move, draw line, and draw curve operations. The changes to many graphics clients were straightforward, although some client programs required changes to the interfaces they exported, cascading the effects to other clients. The Cedar Nucleus was rewritten to provide a new virtual memory implementation, a new automatic storage manager, a new file system, a new stream input-output interface, and new initialization code. Obviously these changes presented a major upheaval to client programs. Considerable coordination and some delay was needed to collect many interface changes at once. Again, such a substantial conversion could be made with assurance through the checking by the compiler and binder. In fact, the release of Cedar with the new Nucleus was acknowledged to be one of the most reliable releases with such major changes.

Debugging graphics applications

Even with the checking assistance of the compiler and binder, design errors may not be discovered until the program executes incorrectly. Symbolic debugging based on source and object version maps of the Cedar release components is convenient and effective. The resident debugger catches errors not handled by an executing program, suspends the process and creates an action area window. Assuming that the suspended process is not one of the window manager processes, then debugging proceeds concurrently while other processes execute. A separate action area for each error provides a new context should an additional error occur either in another program or during attempts to debug the first error. Clients of the graphics package can debug without interfering with the graphics context of the running programs due to the virtual display abstraction of the window manger.

Should an error occur that prevents the resident debugger from creating an action area, such as when the error is within the window manager, then a *world-swap debugger* is available. A world-swap debugger operates in a totally separate address space and examines the saved address space of the system in error. It might execute locally on the same computer or remotely in a different computer which accesses the saved address space over the Ethernet. Most graphics applications are debugged without resorting to the world-swap debugger.

Runtime Binding and Interpretive Graphics Languages

Runtime binding is the dynamic linking of modules during execution. Programs are loaded into the Cedar virtual memory and export their interfaces for binding with other clients. This binding scheme encourages the integration and sharing of programs and profoundly affects our development strategies. Interpretive languages rely on runtime binding to provide an extensible command repertoire.

Application design and testing

There are two interpretive languages available for Cedar: a full interpreter for Cedar expressions, and a simple stack-based interpreter with graphics facilities. Both interpreters can invoke procedures written in the Cedar language and both are useful for experimenting without building a user interface.

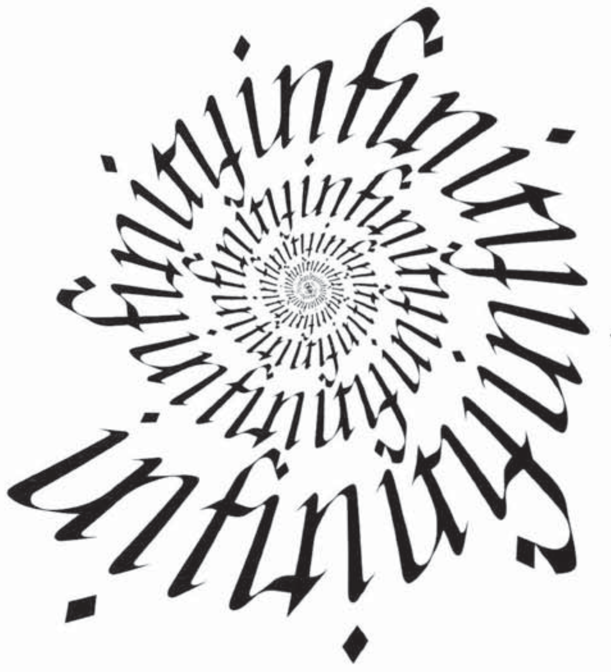
The stack-based graphics interpreter provides a display window and commands to invoke the graphics package. Compiled Cedar modules can be loaded with this interpreter that register new interpreter commands. Using these commands the modules can be exercised from the interpreter. We often collect such exercises into a command file for repeated use and testing. However, these command files are rarely executed by the interpreter top-to-bottom, but rather they provide a script from which sections can be copied to the interpreter using the editor.

Graphics experiments developed this way include the graphical style [2] and curve fitting [27] projects. For example, Cedar modules for the curve fitting algorithms and a suite of interpreted functions constituted the curve fitting prototype. Experience with these algorithms lead to revisions and development cycles from Cedar code to interpreted code and back. When the cycles converge on a stable implementation, the modules are packaged with a module interface suitable for use by illustrators or other clients.

Programmed illustrations

The interpretive graphics language provides another means of generating pictures. Some illustrations are more easily generated by a program than an artist interacting with a paint program or a line drawing program. For instance, Scott Kim's infinity spiral, Figure 3, was created by taking letters defined as spline-outlined areas in the interpreter and mapping them along a spiral using Mesa code loaded with the interpreter.

Figure 3. The infinity spiral created by Scott Kim and John Warnock. Kim designed the letters as spline outlines and Warnock developed the mapping from outlines onto a spiral. © 1981 Scott Kim.



Application Integration

Integrating applications was an important consideration in the design of the Cedar environment. The module interface concept provides a convenient way for client program designers to share abstractions. The object-oriented programming concept permits clients to extend and customize those abstractions. As described earlier, the window manager and the typesetting software both use object classes to extend the set of facilities they provide while at the same time to make it easier to integrate new applications. Illustrators may integrate with our printing software by using the printer file writing interface directly or by using a graphics device object that captures the displayed graphics and text in a printer file.

Unfortunately such integration reveals weaknesses in the abstractions. Our current graphics package for displays has a slightly different imaging model than our printer software. The treatment of sampled images (scanned photographs or shaded images) and the semantics of text are slightly different in the two models. Work is underway to design a new graphics package with a uniform imaging model that is suitable for static images as well as with the interaction semantics necessary for dynamic images.

Another example of integration within Cedar ranges well beyond graphics yet depends on it heavily. The Cedar Whiteboards database is a facility for posting objects on virtual whiteboards using the window manager. A whiteboard is like a blackboard except it is white: all the offices at PARC have porcelainized white surfaces that can be written with erasable marker pens. The postable objects contain text documents, graphical images, programs, printer files, or icons representing objects or other whiteboards. Relationships between objects are represented in the database and can be drawn graphically. The most widely used whiteboard database is the documentation whiteboard. It provides a new Cedar user with a browsing facility for searching documentation, experimenting with various tools, and reading the module interface code. Whiteboards integrates most of the Cedar features and applications to accomplish its task.

Computer Graphics Experiences

Surprisingly large numbers of applications in Cedar utilize graphics. Many applications use the visual graphics provided by the window package to draw boxes around buttons, to highlight interactions with color changes, and to display text with appropriate typography. Others use graphics directly, for example, the performance monitor tool displays animated bar graphs for processor and disk utilization; a clock icon displays an analogue watch dial; the typesetter tool displays animated pie charts to indicate the percentage of work remaining. These examples illustrate the pervasiveness of visual information that Cedar programmers choose to provide when the graphics tools are easily available.

The device independence of the graphics package has proven to be valuable. Devices have been implemented for the black and white display, the color display in all its configurations, the printer file format and the graphics lists facility. Experience with the first few implementations lead to adopting an object-oriented design for devices in the graphics package. Now we have a much better appreciation for the subtleties of imaging models across devices and between static and dynamic imagery.

One insight follows from the shared use of our color mapped display. Since the window package willingly supports concurrent programs on the color display, the single color map becomes a shared resource. Our original color display device lacks a virtual color map so chaotic color schemes are possible if two windows attempt to use the same portion of the color map. The problem will be addressed in the redesign of our graphics package.

Certainly the acceptance of the Cedar graphics package required some exceptional engineering triumphs. Using floating point coordinates within the graphics package prompted skepticism that the package would be fast enough for interaction. Yet through careful crafting of the graphics algorithms for the easy versus hard cases, microcode assistance for floating point calculations and careful use of allocated storage, interaction with the Cedar environment is acceptable even on our least powerful workstation.

User Interface

Although Cedar is very large, the user interface tends to be uniform across a wide variety of applications. Since most textual information is presented with the editor, most tools and programs that require information from the user use the same editing interface. The two most significant benefits are the consistency of editing actions and the universality of text on the screen. Consistency is achieved by brute force: since all editing is done by the same editor, all the editing actions will be the same. Fortunately, most people in the Cedar community prefer this editor over others they have experienced. Universality of text on the screen enables you to compose the information you need from anywhere text is presented by the editor (sadly, the small amount of text not displayed by the editor does not belong to this universe). Commands to execute can be copied from documentation; expressions to interpret can be copied from the program code; file names can be selected from mail messages or directory listings; and so on.

The central placement of the window package encourages a uniform user interface. Generally a new application will find it easier to use the interaction interfaces provided by the window manager and customize them for special needs. Hence a strong culture exists for using the mouse buttons and function keys on the keyboard. This culture has been refined over a decade of experimentation and provides the richest interaction environment the author has experienced.

Novice Cedar users have reported difficulty mastering the manual skills necessary to operate the user interface. More hand-eye coordination is necessary to point with a mouse than to hunt-and-peck on a keyboard. Left-handed people report frustration with the right-handed placement of the mouse, although the function keys on the keyboard are duplicated on the left and right to make it easier for lefties. Still after some period of adjustment most novices demonstrate adeptness with pointing and two-handedness.

A key ingredient to this learning process is a robust undo facility in the editor. Experimentation quickly increases when the user demonstrates proficiency in undoing editing changes, purposeful or accidental. More undo facilities are required before the remainder of the environment permits the same confidence or experimentation.

Real-time Support

Interacting with a pointing device and a highly formatted display requires an abundance of computing power. When a mouse button is depressed and the mouse quickly passed through menu buttons, text, or scroll bars, the screen reacts promptly even on our least powerful workstations. Such feedback of a causal effect appears crucial to the development of hand-eye coordination skills. Concurrent processes permit feedback to execute at a high priority while less time-critical computation proceeds when feedback is static or unneeded.

A significant contribution to the interactive responsiveness is due to the design and engineering of the software. Where performance is of great concern, measurement tools [21] provide insight into the critical areas of resource intensive code. These tools operate without modification to the code being monitored. The prolific rate of accomplishing change to software with the tools available enables greater experimentation with alternative designs.

Conclusions

Experimental programming was the focus of the Cedar project. A productive environment with appropriate tools to build moderate sized software systems is now in daily use at Xerox PARC. With this environment, considerable computer graphics research and development has been accomplished. Many software projects are into their second generation or beyond. The Cedar language, the module interface concept, the software development tools, and the distributed computing system design all collaborate to permit us to stand on one another shoulders, rather than standing on one anothers toes.

References

[1] Baudelaire, Patrick and Stone, Maureen, "Techniques for Interactive Raster Graphics," *Computer Graphics* **14**, *3*, July 1980, pp. 314.

[2] Beach, R.J. and Stone M. "Graphical Style – Towards High Quality Illustrations." *Computer Graphics* **17**, *3*, July, 1983, pp. 127-135.

[3] Bier, E. *SolidViews: An Interactive Three-Dimensional Illustrator.* MSc thesis, MIT, 1983.

[4] Birrell, A., Levin, R., Needham, R. and Schroeder, M. "Grapevine: An Exercise in Distributed Computing." *Comm. of the ACM* **25**, *4*, April 1982.

[5] Birrell, A. and Nelson, B., "Remote Procedure Call," *ACM Transactions on Computer Systems* **2**, *1*, February 1984.

[6] Boggs, D.R., Shoch, J., Taft, E. and Metcalfe, R., "Pup: an Internetwork Architecture," *IEEE Transactions on Communications*, **28**, *4*, 1980, p. 612.

[7] Brown, M., Kolling, K. and Taft, E. *The Alpine File System*, to appear as a Xerox Palo Alto Research Center Report, 1984.

[8] Cattell, R.G.G. *Design and implementation of a relationship-entity-datum data model.* Xerox Palo Alto Research Center Report CSL-83-4, May 1983.

[9] Crow, Frank, "Summed-Area Tables for Texture Mapping," to appear in *Computer Graphics* **18**, *3*, July 1984.

[10] Deutsch, P. *Requirements for an Experimental Programming Environment*, Xerox Palo Alto Research Center Report CSL-80-10, 1980.

[11] Geschke, Charles M., Morris, James H. Jr. and Satterthwaite, Edwin H. *Early Experience with Mesa*, Xerox Palo Alto Research Center Report CSL-76-6, 1976.

[12] Gonzolves, Tim, "Packet-Voice Communication of an Ethernet Local Computer Network: An Experimental Study," *Proceedings of Third International Conference of Distributed Computing Systems*, Miami, Florida, Oct. 82.

[13] Goldberg, A. "Introducing the Smalltalk-80 System," *BYTE*, **6**, *8*, August 1981.

[14] Goldberg, A. and Robson, D. *Smalltalk -80: The Language and its Implementation*, Addison-Wesley, Menlo Park, 1983.

[15] Guibas, L. and Stolfi, J. "A Language for Bitmap Manipulation," *ACM Trans. on Graphics* **1**, *3*, 1982, pp. 191-214.

[16] Ingalls, Dan. "The Smalltalk-76 Programming System: Design and Implementation," *Proceedings of Principles of Programming Language Systems*, ACM, Jan. 1978.

[17] Ingalls, Dan. "The Smalltalk Graphics Kernel," *BYTE*, **6**, *8*, August 1981.

[18] Lampson, B.W. and Pier, K.A.; Lampson, B.W., McDaniel, G.A. and Ornstein S.M.; Clark, D.W., Lampson, B.W. and Pier, K.A. *The Dorado: A High Performance Personal Computer, Three Papers*, Xerox Palo Alto Research Center Report CSL-81-1, January 1981.

[19] Lampson, B.W. and Redell, D.D. "Experience with Processes and Monitors in Mesa," *Seventh Symposium on Operating System Principles*, Asilomar, Dec 1979 and Comm. ACM **23**, 2, February 1980.

[20] Lyons, R.F. *The Optical Mouse, and an Architectural Methodology for Smart Digital Sensors*, Xerox Palo Alto Research Center Report VLSI-81-1, 1981.

[21] McDaniel, G. "The Mesa Spy: An Interactive Tool for Performance Debugging," *Conference on Measurement and Modeling of Computer Systems*, Seattle, Washington, Sept 1982.

[22] Metcalfe, R.M. and Boggs, D.R.; Crane, R.C. and Taft, E.A.; Shoch, J.F. and Hupp, J.A.; *The Ethernet Local Network: Three Reports*, Xerox Palo Alto Research Center Report CSL-80-2, February 1980.

[23] Mitchell, J. *Mesa Language Manual*, Xerox Palo Alto Research Center Report CSL-79-3, 1979.

[24] Myers, Brad A. "Incense: A System for Displaying Data Structures," *Computer Graphics*, **17**, *3*, July, 1983, pp. 115-125.

[25] Newman, W.M. and Sproull, R.F. *Principles of Interactive Computer Graphics*, 2nd ed., McGraw-Hill, New York, 1979.

[26] Pier K. *A Retrospective on the Dorado, A High-Performance Personal Computer*, Xerox Palo Alto Research Center Report ISL-83-1, August 1983.

[27] Plass, M. and Stone, M. Curve-Fitting with Piecewise Parameteric Cubics. *Computer Graphics*, **17**, *3*, July, 1983, pp. 229-239.

[28] Petit, Phil, "Chipmonk, an Interactive VLSI Layout Tool," *IEEE Computer Society International Conference*, San Francisco, Feb 1982.

[29] Rovner, P. *On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language*, to appear as a Xerox Palo Alto Research Center Report, 1984

[30] Schmidt, E. *Controlling Large Software Development in a Distributed Environment*, PhD Thesis, U.C. Berkeley EECS Dept. December 1982; also available as Xerox Palo Alto Research Center Report CSL-82-7, 1982.

[31] Schroeder, M. D., Birrell, A.D. and Needham, R.M., "Experience with Grapevine: the Growth of a Distributed System," *Proceedings of Ninth ACM Symposium on Operating Systems Principles*, Nov. 1983.

[32] Stewart, L.C., Swinehart, D., and Ornstein, S. "Adding Voice to an Office Computer Network," *Proceedings of GlobeCom 83, IEEE Communications Society Conference*. Nov. 28, 1983.

[33] Teitelman, W. and Masinter, L. "The Interlisp Programming Environment," Computer **14**, *4*, 1981, pp. 25–33.

[34] Warnock, J. and Wyatt, D.K. "A device independent graphics imaging model for use with raster devices," *Computer Graphics*, **16**, *3*, July 1982, pp. 313-319.