Metcard Digest

MetaCard is a multimedia authoring tool and GUI development environment for Microsoft Windows 3.1/95/98/Me/NT/2000/XP, Unix/X11, and Macintosh systems. Using MetaCard is the easiest way to build graphical applications, Computer Based Training (CBT), on-line documentation, and a wide variety of other products. Get the MetaCard Starter Kit and build something today!

The MetaCard engine technology has been acquired by Runtime Revolution. More details of this technology transfer and how it will affect MetaCard users can be found at http://www.runrev.com/metacardpr.html.

Overview

The 2.5 release of MetaCard Corporation's award-winning GUI application and hypermedia environment for UNIX/X11 workstations, Windows 3.1/95/98/Me/NT/2000/XP, MacOS, and Mac OS X now helps developers be even more productive. Not just another expensive and complex application development environment that requires extensive programming with a third generation language, MetaCard's Very High Level Language (VHLL) and Integrated Development Environment (IDE) make producing high quality, high functionality applications faster and easier than ever before.

MetaCard can also be used to prepare on-line help and computer-based training (CBT) packages. The multiple-card metaphor and hypertext-linking capability makes it natural for producing on-line reference manuals. As an example, the complete documentation for the MetaCard environment itself is available on-line in MetaCard stacks.

Press and end-user response to the product has been highly favorable. UNIXWorld magazine named MetaCard one of the 10 Best Products of 1992, and MetaCard was a finalist in SCO's Product of the Year contest in 1993. SunWorld magazine says "... once users have come up to speed with the package, it becomes a must-have UNIX utility, allowing fast prototyping, easy interactive presentations, and simple custom apps." MetaCard 2.0 was one of SCO World Magazines Hot Products for 1996, and MetaCard 2.1 was selected as one of the Top 10 Software Products of 1997 by IEEE Computer Graphics and Applications (Jan/Feb 1998 issue).

Benefits

Through the use of reusable software components that reside in MetaCard "stacks," this rapid application development environment allows even inexperienced computer users to develop applications in a fraction of the time required by other development tools. The interface layout tools are all WYSIWYG and will feel familiar to anyone who has used a drawing or painting program. Building an aesthetically pleasing and functional interface requires none of the esoteric knowledge of attribute and callback names required by other Motif based development systems.

When compared with application development tools that require the use of a third-generation language such as C/C++/Java, far fewer lines of code must be written to complete an application with the MetaTalk language. This results in higher developer productivity, higher product reliability, and reduced maintenance costs. The power and wealth of built-in capabilities of the language also means that you won't "hit the wall" and have to cut important product features late in development, a common occurrence when using "no scripting required" presentation tools and icon-based multimedia development environments.

Another benefit of the MetaCard environment is true application portability: applications developed on any platform can be run on any other supported platform without recompiling or other preprocessing.

Since MetaCard engines are freely distributable (the Home stack is the licensed unit), no special licensing arrangements or royalties are necessary to take advantage of this portability or distribute applications built with MetaCard. This flexibility will make multiplatform environments much more convenient for developers and users alike.

Key Features

- Easy to use yet powerful interactive GUI development environment with an integral

interpreted scripting language.
- Support for all button and field styles, pulldown and popup menus, tooltips, tabbed dialogs, and floating palettes.
- Supports Motif, Windows 95, and MacOS look and feel, switchable at runtime.
- Very High Level Language (VHLL) features such as associative arrays, regular expression pattern matching, and message passing support creation of high functionality applications with fewer lines of code.
- Custom (user-defined) properties for all objects with get/set callbacks support object-oriented development.
- Interactive GUI script debugger and profiler shorten development times required to produce high-quality applications.
- Build single-file, standalone, double-clickable applications for all platforms. No installers, external files, DLLs, browsers, or virtual machines are required to deploy MetaCard applications.
- Text editing tools that support multiple typefaces, sizes and colors, subscript and superscript, automatic scrolling, search, sort, and hypertext links. Import and export HTML format text.
- Object (vector) graphics with properties that can be set from the scripting language for dynamic data display.
- Import and play back sounds in .au (mulaw), WAV, and AIFF formats.
- Import and play back movies in QuickTime, AVI, MPEG, and FLC movies on UNIX systems, and in all format supported by QuickTime or Video for Windows on Windows and Mac systems.
- Import screen snapshots directly into MetaCard with the snapshot tool, or import images in GIF (including animated transparent GIFs), JPEG, PNG, BMP, XWD, and many other formats. A complete set of full 24 bit color image editing/painting tools support creation and editing of images, icons, cursors, brush shapes, and fill patterns on all platforms.
- Card transition visual effects and double-buffered object manipulation and animation are available on all platforms: no special hardware or software is required.
- Built-in support HTTP GET and POST make building Internet applications easy. The highest-level socket support of any scripting language allow you to use other popular Internet protocols like FTP/HTTP/SMTP/Telnet/etc. directly in your applications, or build your own custom protocol.
- Object architecture, scripting language, and external procedure call interface are source compatible with Apple Corporation's HyperCard. Import HyperCard 1.2 and 2.X stacks in raw binary, BinHex and MacBinary formats. SuperCard to MetaCard and OMO to MetaCard converters are also available.

The MetaTalk language has the power of languages like Perl and Smalltalk, but is much easier to learn and use. The MetaTalk virtual compiler gives performance comparable to the Perl and Java languages, which is on average 5 times the performance of HyperTalk and SuperTalk, and up to 30 times the performance of languages like the UNIX shells which have only conventional interpreters.

The MetaTalk language has all the features common to third-generation languages like C/C++/Java but has a much simpler syntax. The language supports all modern programming constructs such as if-then-else, repeat statements, switch-case, and local and global variables. MetaCard subroutines and function handlers support variable length argument lists, recursion, and a flexible message passing architecture.

MetaTalk also has extensive support for user-interface development. For an example of the power of the language and GUI toolkit, one has to look no further than the MetaCard development environment, which itself was developed in MetaCard. Not only does this demonstrate MetaCard's capabilities, it also provides a benefit unavailable in any other development system on any other platform: complete control over the development environment.

Any dialog, any menu, any error message, and all documentation can be enhanced, extended, or even replaced by any licensed MetaCard user. Why live within the static environments supported by other Motif and hypertext environments when you can have complete control with MetaCard?

A C based extension capability allows MetaCard to be used to develop computationally intensive or hardware dependent applications. You can call C or C++ functions directly from MetaCard with no relinking or other processing required. HyperCard-compatible XCMDs and XFCNs can be used directly on the MacOS version of MetaCard, and easily ported to the other

platforms using the HyperCard 1.2 compatible callback API.

An embedded version of MetaCard is also available so that you can attach the MetaCard scripting language and graphical tools directly to a C or C++ application. MetaCard also supports conventional interprocess communication using POSIX standard I/O functions on Windows and UNIX systems.

Architecture

MetaCard stacks consist of a collection of resources and event handlers, an architecture similar to most other GUI-based application development environments. The resources are the descriptions of the look of the interface including the sizes, positions, colors and other attributes of the buttons, text fields, and other controls.

Unlike many other development tools, however, MetaCard objects are not constructed in the application code. Instead, they are read directly out of the editable resource files, called stacks. So instead of hard-coding the application appearance or forcing users to deal with complex and easily corrupted text based resources for customization (such as .Xdefaults on UNIX systems), a direct-manipulation interface is always available.

Event handling is also greatly simplified in MetaCard. Instead of writing "callback" routines in a third generation language which require that the application be recompiled and relinked after every change, an interpreted scripting language is provided. Whenever events occur, messages are sent to handlers written in this MetaTalk language.

Errors that occur during script execution cause the MetaCard engine to stop processing, and to point out the exact line and position where the error occurred in a script editing window. After fixing the problem, the script can be rerun immediately: no recompiling or relinking is necessary. This design not only drastically reduces the time spent prototyping an interface, but also reduces the time spent testing and debugging it.

Pricing and Supported Platforms

Single user licenses of MetaCard are $995 (U.S. dollars) and can be used on any platform or combination of platforms. No need to buy a separate version for each platform as many other authoring tools require!

Multi-user (site) licenses are available from $3,600 for five users up to $30,000 for 250 users. Higher-ed educational institutions receive a 50% discount on any size license. Special pricing and bundled software is available to K12 schools through the MetaCard K12 Program.

MetaCard 2.5 is supported on 68K and PPC Macintosh systems running MacOS 7.1 through 9.X, with a separate Carbon/Mach-O engine for use with Mac OS X. The Win32 engine runs on Windows 95/98/Me/NT/2K/XP and Windows 3.1 systems with the Win32s library (see the README for installation instructions and a list of features not available under Win32s). Nine popular UNIX/X11 platforms are also supported: SunOS SPARC, Solaris SPARC, Solaris x86, Darwin (Mac OS X), SGI IRIS, HP-9000/700, IBM RS/6000, BSD UNIX, and Linux.

MetaCard makes very small demands on system resources. The engine sizes range from 1 to 2 MB, with most stacks ranging in size from 20K to 500K and a total disk space requirement of less than 4MB for the development system. No special display, networking, or other hardware is required.

Your initial MetaCard license includes unlimited technical support via email and free upgrades for a 1 year period. After that period, you'll need to renew your subscription and/or support contracts for each 1 year interval. See the FAQ for subscription renewal details. Non-toll-free telephone support is available with a separate phone support contract. See the licensing information in the MetaCard Starter Kit for details.

Conversion from HyperCard, SuperCard, or OMO

Whether it's to achieve cross-platform portability, to improve the performance or capabilities of a product, or to switch to a tool supported by a stable, long-term oriented company, it's becoming increasingly common for applications developed in HyperCard, SuperCard, Oracle Media Object (OMO), WinPlus/ObjectPlus, or HyperSense to be ported to MetaCard. Though some work will be required to convert most applications to MetaCard, in most cases it will only be a small fraction of the time and effort required to redevelop the product from scratch using some other language.

After conversion, you'll instantly benefit from the average **5** times better script execution performance MetaCard has over HyperCard and SuperCard (see the Benchmarks page for details), and the 32K script and field contents length limits will no longer constrain your development (MetaCard scripts can be 64K per object, and there is no limit on the amount of text you can put into fields).

Conversion from HyperCard

MetaCard can import HyperCard stacks directly from their native file format on all platforms. All objects and object properties (including scripts), bitmaps, icons, cursors, and sounds are converted. PICT images, colors (AddColor), and other information stored in the resource fork are not converted.

On Macs, MetaCard can import directly from HyperCard stacks. On Windows and UNIX systems, the stacks can be imported in raw binary, BinHex, and MacBinary formats, the latter two preserving resource-fork information including icons, cursors, and sounds. For best results, compact a stack before importing it.

After importing, conversion usually consists of addressing script-language compatibility issues, changing the size of objects and fonts to improve the look of the UI, and editing or replacing the bitmap images used in the stacks (many HyperCard stacks have white areas painted on them which are invisible with HyperCard's default white background, but become very obvious when viewed on MetaCard's default gray background).

MetaCard supports most HyperCard XCMDs and XFCNs, but these are not copied automatically from the resource fork of HyperCard stacks when they are imported. Simce XCMDs are not portable across platforms, you should first determine whether or not MetaCard has a feature built-in that is performed by an XCMD in the HyperCard stack. If an XCMD or XFCN is really required, you can move it into the resource fork of the MetaCard stack after you save it.

An excellent tutorial that shows how to convert a HyperCard application to MetaCard is available at http://www.hyperactivesw.com/mctutorial/.

Conversion from SuperCard and OMO

Conversion from SuperCard and OMO requires the use of a converter. These converts have two components. The exporter component runs in the source environment and exports a text file representation of the stack/project. The importer component runs in MetaCard and builds a MetaCard stack based on the information in the text file. The converters are available on this site as sctomc.sit and omotomc.sit.

Due to relatively poor performance of SuperCard and OMO, the exporter generally runs quite slowly and can take hours to export large stacks. We recommend breaking up large projects into small pieces (1MB or less each) prior to conversion for best results. After conversion, script compatibility and font and object size issues will need to be addressed, just as with HyperCard.

An enhanced version of the SuperCard to MetaCard converter that converts bitmap objects is available on the Cross Worlds WWW site.

Conversion from ObjectPlus/WinPlus and HyperSense

No pre-built automated conversion is possible from these tools to MetaCard. For small projects, reconstructing the project using the MetaCard development environment is the most efficient route. It may be possible to use copy/paste to move scripts from the source tool to MetaCard, greatly speeding development of the new application.

For larger projects, building an automated conversion tool may be worth considering. Since a new converter could be based on the existing SC or OMO converter, it may not be very difficult to build such a tool.

Script Conversion

The core of the various xTalk dialects are very compatible, and in general MetaCard is more compatible with any of the other tools than they are with each other (considerable effort has been made to improve compatibility). For example, MetaCard supports both the dynamic message path used in HyperCard and OMO, and the static path used in SuperCard. It supports

shared text and button hilites like HyperCard, and graphic objects like SuperCard. In addition, many of the commands and functions that are handled in HyperCard and SuperCard (like flushEvents and getResources) are built into the MetaTalk language.

There are a few key areas where compatibility problems frequently arise, however. The most common of these is in menu-handling scripts. Each of the xTalk-based products handle menus differently, partly because of the demands of the various development platforms (Windows and UNIX have no system menu bar like MacOS does), but also because menu support was the most recently added or enhanced in these products. As a general rule, figure that every script statement that refers to menus will need to be changed. Menu support in MetaCard is comparable to that available in the other tools, however, so extensive redesign of menu management routines will not be required in most cases.

Secondly, since the MetaCard development environment is very different from that used in any of the other tools, scripts that alter the development environment (by accessing the Mac menu bar or palettes or debugging features) won't run correctly in MetaCard. MetaCard's support of the HyperTalk doMenu command is rudimentary because of this limitation. Instead, MetaTalk has equivalent functions built into the scripting language (e.g., cut/copy/paste/create card/etc.), and these commands should be used instead of doMenu.

There are many instances where the other xTalk languages are more lenient than MetaTalk's. For example, the use of the word "the" in function and property names is more strictly enforced in MetaCard. Another potential trouble spot is that HyperCard allows predefined terms, such as function and property names, to be used as variable names, whereas MetaCard generally does not.

Since MetaCard has many more built-in commands, functions, and properties than HyperCard, it is much less tolerant of leaving quotes off of literals than HyperCard. The SuperCard-compatible feature "explicitVariables" allows you to set MetaCard's script compiler to flag unquoted literals as errors.

MetaCard is slightly more lenient than HyperCard when addressing objects. When a button or field is specified in MetaCard, both cards and backgrounds are searched, as a opposed to HyperCard where buttons are searched for on cards, and fields on backgrounds. You can override this behavior by using the word "card" or "background" in the object specification.

MetaCard supports custom (user-defined) properties, compatible with those used in SuperCard and Oracle Media Objects. There is no need to predefine property names with the "define" command like you must in OMO, though the "define" command is supported in MetaCard for compatibility purposes. MetaCard also supports getprop and setprop "triggers" which are called when custom properties are set or retrieved, a important feature missing in OMO and SuperCard.

MetaCard Corporation encourages users to report any incompatibilities they find to help achieve the goal of making MetaCard capable of importing and executing a greater percentage of HyperCard stacks unmodified in the future. When in doubt, consult the MetaTalk Reference stack to verify whether a feature is supported or not.

MetaCard 2.5 Reviewers Guide

This document is designed to help product reviewers understand what MetaCard is and what it can be used for, where it came from, how it is marketed and sold, and give some idea of what it will look like in the future.

MetaCard Specifications

Product name:	MetaCard
Product version:	2.5
Product category:	Cross-platform application development, multimedia authoring, CBT development, and programming instruction tool.
Availability:	April 2003

Pricing:	$995 for 1 year subscription including technical support for a single-user/all platform license. Significant discounts for academic users and multiple-user license packages.
Upgrade pricing:	Free for users with a current subscription. Subscription renewal is $300 per year, with discounts for academic and multi-user licenses. Unlimited technical support via email is $200 per year after the first year.
System requirements:	*Authoring, all platforms* 16MB RAM 800x600 display resolution 8, 16, or 24/32-bit color *Deployment of minimal app, all platforms* 2MB RAM + OS requirements no resolution requirements monochrome or better screen depth *Mac requirements* 68030 or better with System 7.1 or higher *Windows requirements* Windows 95/98/Me/NT/2000/XP or Windows 3.1 with Win32s
Technical Support:	Free unlimited technical support via email with a current technical support subscription. Phone support available at extra cost.
Training:	Available on-site or at MetaCard Corporation headquarters. Pricing information available upon request.
Distribution:	Available via electronic means directly from MetaCard Corporation world-wide. Local dealers available in Europe and Pacific Rim countries.

What is MetaCard?

MetaCard, like its cousins HyperCard, SuperCard, and ToolBook, is a difficult product to "pigeonhole". It has characteristics of high-level application development tools like Visual Basic and Delphi (a GUI IDT, scripting language, script editor, and script debugger). It has cross-platform support like Java and some 4GL-based products. It has data management and query features like databases such as Oracle, Sybase, Access, and Filemaker. It has multimedia capabilities comparable to dedicated multimedia authoring tools like Macromedia Director and the slide-oriented format found in presentation tools like PowerPoint. Finally, it has text management and hypermedia capabilities found in document preparation tools and Web browsers.

While the resulting combination of these characteristics could suffer from the "jack of all trades, master of none" syndrome, a remarkable synergy occurs instead. As users come to expect multimedia features and well-integrated context sensitive documentation and task-oriented wizards in all of their applications, tools that have been designed from the ground up to present these capabilities in a unified architecture have distinct advantages over tools that have them

grafted on.

The MetaTalk scripting language has been designed to be the easiest to learn and easiest to use general-purpose scripting language available. It can be learned as easily as so-called "visual programming" tools, yet is free from the profound limitations that are characteristic of those tools. The MetaCard development environment is also designed to be easy to learn and easy to use, and non-programmers can master the tool much more quickly than they could conventional programming tools like Visual Basic or Dephi.

The MetaTalk scripting language is compatible with the HyperTalk language used in HyperCard and the SuperTalk language used in SuperCard, but has many extensions. For example, HTTP support is built in, as is support for arrays and regular expression pattern matching. MetaCard supports the full range of features required for UI development including controls like combo-boxes and a tab control, and support for all types of dialogs (popups, modal and modeless dialogs, and floating palettes). Most of the popular extensions of SuperCard over HyperCard are also supported, including full color support, the "graphic" object, backdrop windows, and functions and properties like "within", "hilitedLines", "insert script", and "lockCursor". Most newer HyperCard features are also supported including shared/nonshared text and button hilites, and "start using".

Target Market

The MetaCard installed base is as broad as for most other horizontal-market products, with no one industry or type of application predominating. Scientists, engineers, and other technical professionals use MetaCard to build specialized data analysis, simulation, and presentation tools. Multimedia professionals use it to produce interactive product brochures and Computer Based Training packages. Educators use it to produce interactive courseware, demonstrations, and to teach programming and multimedia development. Technical writers use it to produce on-line help systems and documentation indexes and directories. Corporate and government software development departments use it to produce in-house and commercially distributed applications, rapid prototyping, and to build customized software development tools. Small independent software developers use MetaCard to build custom software applications for their clients. Hobbyists and other end users use it to learn programming and to build productivity enhancing tools for their own use.

The size of the installed base by platform ranks UNIX, Windows, Mac. Sales rates currently rank MacOS, Windows, UNIX, however, so the relative position of these platforms in the installed base will probably begin to equalize over the next year.

History of the Product and Company

MetaCard Corporation was formed in 1990 to develop easy to learn and easy to use multimedia and application development tools for UNIX workstations. The design goal was to produce a product that was compatible with HyperCard, but with full support for all the control and dialog types required to build any type of application.

The 1.0 version of MetaCard was released in June of 1992, the entire development environment of which was built in MetaCard. Since then, the development environment for every MetaCard release has been built exclusively with MetaCard itself, making it unique among high-level development tools.

In the 1.X series of releases (1.0-1.4), most of the extensions over HyperCard made by the SuperCard developers were added to MetaCard, including a compatible "graphic" object that supports vector-graphics, and many SuperCard-specific language features. Compatibility with HyperCard was also improved, and the HyperCard file format was reverse-engineered to allow loading HyperCard stacks directly into MetaCard without requiring a preprocessing step.

There were two major areas of development for the 2.0 release. The first was a complete rewrite of the language execution system to improve performance and add features not available in other xTalk languages. The resulting "virtual compiler" technology offers performance comparable to byte-code interpreted languages like Perl and Java, and which is at least 5 times faster than the partial compilation compilers used in comparable tools including HyperCard and SuperCard. It is up to 30 times faster than languages that rely on conventional interpreters like JavaScript and the UNIX shell languages.

A superset of HyperTalk, the MetaTalk language now has advanced language features not found in HyperTalk and OpenScript (the HyperCard and ToolBook scripting languages,

respectively). For example, like Perl, MetaTalk supports associative (string-indexed) arrays and regular-expression pattern matching, features that ease processing of complex files like HTML forms and mail folders. MetaTalk also has more specialized language extensions including call-by-reference arguments, delayed-event scheduling commands, and the ability to manipulate binary data (including NULL or 0 bytes).

Support custom (user-defined) properties was also added to all object types for MetaCard 2.0. This feature makes it even easier to use object-oriented techniques to develop GUI applications. MetaCard custom properties are also persistent and support triggers which call scripts when the properties are queried or set, supporting the most common applications of OO database techniques without requiring the use of complex (and expensive) C++ based OO database products.

The second major development for MetaCard 2.0 was the isolation of all OS-specific routines into separate modules. This greatly simplifies the job of porting MetaCard to other operating systems. The first port was to Windows 95/NT, which was released with MetaCard 2.1 in early 1997. Support for Windows 3.1 under Win32s was added for the 2.1.1 release in August 1997. The 2.2 release added support for the Macintosh.

Stacks are portable to different platforms without requiring recompiling or other preprocessing. In most cases, all that will be required when first moving a stack to a new platform is to make minor adjustments in object sizing and spacing to compensate for the fact that fonts are slightly different on each platform. After making these changes the stack can be used on all platforms without further modifications.

Applications developed with MetaCard can be distributed with the MetaCard engines (similar to a Java virtual machine or the Visual Basic DLL) without paying royalties. The only restriction on the operation of these stacks is that there will be a limit on the length of scripts that can be set on-the-fly in this environment. The "do" command is still available, however, so this restriction will have little or no practical impact (the "do" command is one of MetaTalk's most powerful because it allows you to build up a series of commands into a variable and then execute them, something that isn't possible with Java or any other 3rd generation language).

How MetaCard is marketed and sold

MetaCard is sold primarily direct from MetaCard Corporation. The MetaCard Starter Kit can be downloaded from the MetaCard WWW and FTP sites (http://www.metacard.com and ftp://ftp.metacard.com, respectively). It can also be sent out via email and on CDROM, floppy disks, or DAT or QIC format tapes. Resellers in Europe and the Pacific Rim can offer native-language pre-sales and technical support (30% of MetaCard sales are to customers outside the U.S.).

The MetaCard Starter Kit is fully functional, except that there is a limit on the number of statements you can put in the script of each object (currently 10 statements, which doesn't count the "on" and "end" handler definitions, if-then, repeat, switch-case, or comment lines). Installation of a software license key enables the full functionality of the product. The key can be acquired by filling out the licensing form and faxing, emailing, or HTTP POSTing it in to MetaCard Corporation along with a Purchase Order or credit card number (credit cards numbers are encrypted prior to transmission via email or HTTP).

Pricing is $995 U.S. Dollars for a single-user all-platform license (no need to buy a separate product to develop or deploy on a given platform). This includes free unlimited technical support via email.

The full range of marketing techniques are used to promote MetaCard, including advertising, PR, and participation on CD-based promotions. The single most important source of leads comes via electronic methods such as WWW searchs, links from other WWW sites, FAQ lists, newsgroups, and mailing lists.

Limitations

The MetaCard development interface is relatively generic in look and behavior because it must run on all platforms and because it receives lower development priority than improvements to the core engine. The highest priority for MetaCard development is to provide the ability for developers to create applications that pass muster as native applications, not necessarily to provide them with a development tool that itself is the best example of a native application on each platform.

Likewise the MetaCard development environment is relatively simple compared with some other IDEs, although it does include things like an debugger and an autoformatting/colorizing script editor that some other tools lack. Again, MetaCard's unparalleled adaptability and extendibility makes it easy for developers to mold the development environment to suit their needs.

MetaCard has relatively limited support for platform-specific component models like OLE/COM/ActiveX. These can be used from externals, but it is not a trivial matter to do this. Of course, it's seldom a trivial matter to acquire, pay for, and use these components in Windows-only products and of course all hope for portability vanishes once the decision to use these components is made.

Cut/Copy/Paste support is currently limited to text. The inability to use the system clipboard to transfer image data may seem to be an inconvenience but is not actually a problem in practice: bitmap data passed via the clipboard is uncompressed and so is in an inappropriate format for inclusion in a stack anyway (developers should import PNG, GIF, or JPEG format images instead so that they can take advantage of the superior compression controls available in specialized image editing tools). PICT data passed on the clipboard could not be viewed on UNIX or Windows systems because they do not provide support for PICT like MacOS does. PICT data is therefore also unsuitable for cross platform development and so the inability to cut/copy/paste it is not a significant limitation.

Features planned for future releases

Features planned for upcoming releases include improved table support, SSL, and extended access to QuickTime features like sprites and text tracks. New high-level language features, such as "columns" as a chunk type and "file" as a first level object type, are also planned.

A method for outputting Java byte-codes equivalent to MetaCard scripts has been designed, but implementation has been delayed until the serious performance, compatibility, and functionality limitations in Java have been rectified.

The MetaTalk language will also be extended to provide a more full featured object-oriented programming environment, which will allow development of larger-scale applications with MetaCard. The key tenants of object-oriented programming - encapsulation, polymorphism, and inheritance - are already available in the MetaTalk language, but must be extended before MetaCard will be as appropriate for large multi-developer projects as it is for the single-user projects that are presently its forte. These features and OODBMS features such as multiuser access, version control, and distributed data management are currently being designed.

MetaCard K-12 Program

When the thrill of creating pretty pictures and arranging clip art in HyperStudio wears off and students are are not willing or able to invest the time and effort required to learn the cryptic and out-dated Logo language, it's time to move up to MetaCard. MetaCard's easy-to-learn and easy-to-use MetaTalk scripting language is the choice of kids and professional developers alike. It's the easiest way to move up to the world of programming, yet provides a tool that they'll never outgrow.

The free MetaTalk Programmer package is a self-contained computer programming course for middle and high schools. A MetaCard license is not required to use the MetaTalk Programmer curriculum to teach computer programming concepts and techniques.

Why choose MetaTalk Programmer to teach computer programming?

It's fun!

One of the things that frustrates students about schoolwork is the lack of applicability to their lives and interests. It doesn't take a genius to figure out that students tend to pay more attention and try harder when a lesson is interesting to them. For this reason, MetaTalk Programmer stresses a high interest, hands-on approach to programming. Students create interesting programs from day one, and the more complicated the scripting gets, the more interesting the programs are.

The MetaTalk Programmer application is constructed with three goals in mind:

1. Each of the 20 lessons will be able to be accomplished within a 30 minute time period.

2. Each lesson will contain extension activities for students who can handle an accelerated pace.

3. Every 5 lessons will be a comprehensive review of all techniques covered to that point.

Middle school students can learn it.

Computer programming is something that has always been seen as a "high math" skill, out of reach for most "average" people, and certainly beyond the reach of children. And originally, this may have been the case... but it is not the case today. This unit is designed to teach basic concepts of computer programming to ANY middle school student, regardless of their mathematical background.

When you mention programming to most people, they imagine thousands of lines of obscure symbols and numbers, and some programming does look like that. But this unit uses a very advanced programming language called "MetaTalk", which closely resembles English. Look at the following line of MetaTalk code:

```
answer 2 + 2
```

This particular example of MetaTalk code would do just as you suspect it would... add 2 and 2 and answer 4. Not all code will be as easy to relate to, but the point is that real words and phrases are used to program instead of obscure symbols and numbers.

MetaTalk is the perfect language for beginning computer programmers or anyone with an interest in programming, because it so closely resembles English. In fact, MetaCard's programming language is one of several "Scripting" languages... titled scripting because it resembles English, and it controls a program much in the way that a script directs an actor or actress.

MetaTalk Programmer features some of the commands and structures that one will find in almost every other programming language. The logical thought process learned in MetaTalk Programmer can easily be applied when learning another programming language, or when working in a logical mathematics area such as Algebra.

It's valuable.

MetaTalk Programmer can be used by teachers with no programming experience or background. Teachers may discover, however, that they become interested in programming after using this unit! Programming is both challenging and rewarding, and can be very beneficial to both students and adults. It offers a unique combination of logical thinking and creativity not found in any other educational activity. Programming is an area experiencing tremendous growth in the job market, and it has a very promising future. The introduction to programming offered in MetaTalk Programmer might be a turning point for that one quiet student who sits in the back of your classroom, full of creativity, just waiting for an opportunity to demonstrate it.

MetaTalk Programmer runs on all platforms supported by the MetaCard Starter kit, which you must install before you can use MetaTalk Programmer. After installing the Starter Kit, run MTP by double-clicking on it, dragging it to the MetaCard engine, or starting up MetaCard and opening "mtp.mc" from the File/Open... menu. You may need to set the file type of the stack to "MCRDMSTK" on MacOS systems to open it with methods 1 and 3.

You can download MTP and the teachers guide by clicking on the links below. If you get a screen full of characters, shift or control-click on the link and choose to "save" from the popup menu.

mtp.mc
mtpguide.mc

To learn more about the goals and prerequisites for the course, be sure to click on the "Teachers Guide" link on the main menu page. After completing the MTP course, students will probably want to start using the full MetaCard development environment and the free Starter Kit version of MetaCard is a good place to start. When they want to build larger products than the Starter Kit allows, an upgrade to a full license is in order.

K-12 schools (elementary through high school) can purchase unlimited usage MetaCard licenses starting at $250 (U.S. Dollars) for a 10 user lab pack. No smaller-quantity licenses are available through this program. Contact MetaCard Corporation for information on larger license packages.

Embedded MetaCard

Embedded MetaCard is a library form of the MetaCard engine for UNIX/X11 platforms. It can be used to develop customized versions of the MetaCard engine for applications that make extensive use of externals developed in compiled languages like C, C++, and Fortran. It was designed for application developers who need a professional-grade GUI development environment without the long learning curve, large resource requirements, and high cost of conventional Xt/Motif development tools.

Although the standard MetaCard engines have always supported the ability to call C and C++ functions from the MetaCard scripting language, these calls are made using Remote Procedure Calls (RPCs) on UNIX systems. As is the case with any RPC architecture, calls to MetaCard external procedures on these systems involve considerable call overhead, and debugging is much more difficult because two processes must be monitored at the same time. By linking the MetaCard engine directly to a C or C++ application and libraries, call overhead virtually disappears, and debugging is greatly simplified.

Embedded MetaCard brings together the best features of non-graphical development environments such as Tcl/Tk and the COSE Desktop Korn Shell (an interpreted string-oriented language, easy extensibility, and simplified access to the GUI environment) and high-end layout tools such as Teleuse and UIM/X (an integrated development environment, direct manipulation layout tools, and support for compiled third-generation languages and libraries). This combination makes development with Embedded MetaCard faster and easier than with any of these other types of tools. Embedded MetaCard can be used to produce GUI applications in any domain, but it is especially suited to projects such as GUI clients for RDBMS servers and vertical market applications where short development cycles and easy customization are critical capabilities.

Embedded MetaCard is supported on 11 Unix/X11 platforms: SPARC SunOS 4.1.X, SPARC Solaris 2.X, Solaris x86, DEC Alpha, HP 9000/300 HP-UX 8.X, HP 9000/700 and 800 HP-UX 9.X, HP 9000/700 and 800 HP-UX 10.X, SCO Open Desktop, Silicon Graphics IRIS 5.X, Linux, and IBM RS/6000. Single platform pricing is $1495, with libraries for each additional platform selling for $500. As with the regular MetaCard engines, no runtime fees are required to distribute applications built with Embedded MetaCard.

FREQUENTLY ASKED QUESTIONS

For more questions and answers about MetaCard, see the old best.com MetaCard mailing list archives and old runrev.com MetaCard mailing list archives and new MetaCard mailing list archives, indexed records of all messages posted to the metacard mailing list. To search the new archives use the Google site search. For an updated FAQ and other information on MetaCard, see the Cross Worlds Computing site. Because the MetaTalk language used in MetaCard is a superset of the HyperTalk language used in HyperCard, the sites HyperCard Home Page, , and HyperCard Heaven may also be useful places to get more information.

1. What is MetaCard?

MetaCard is a graphical application development and multimedia authoring tool for Windows 3.1/95/98/Me/NT/2000/XP, UNIX/X11, and Macintosh systems that is similar to such PC/Mac products as HyperCard, SuperCard, and ToolBook. It can be used to develop GUI

applications, GUI front ends to shell scripts or other character based applications, Computer Based Training (CBT) systems, on-line help systems and other documentation, and interactive presentations.

The basic design is similar to commercial GUI builder systems such as Visual Basic on MS-Windows or UIM/X for X11/UNIX, but MetaCard is much easier to use, and also has multimedia features found in presentation packages and text management features found in document preparation packages.

The MetaCard application consists of an engine, which is specific to each computing platform, and a set of stacks, which are portable. There are 4 primary stacks distributed with the product: mchome.mc is the Home stack and controls the licensing of the product, mctools.mc is the development system stacks including the menus and dialog boxes, mchelp.mc is the on-line documentation, and mcdemo2.mc is a multimedia demonstration stack.

MetaCard is a product and registered Trademark of MetaCard Corporation, and has been in development since 1990 and in commrecial release since June 1992.

2. What kinds of things are people building with MetaCard?

A wide range of products have been built with MetaCard. See the Applicationspage for examples.

The primary use to date has been to develop GUI applications, application prototypes, and GUI front ends to command-line applications. In many cases these applications are in continuous use in production environments.

The second most common use is probably for on-line presentations and demonstrations. The royalty-free distribution policy combined with the interactive nature of the presentations make it a natural for producing demonstrations for use in kiosks, trade-show booths, multimedia CDROMs, and other free-running presentations.

The third most common application is for building on-line documentation and training packages. In most cases, MetaCard is used as a supplement to a large application, but it is also being used to document company policies and procedures.

These categories by no means exhaust the potential range of applications of MetaCard.

3. What platforms will MetaCard be ported to and when?

MetaCard 2.4 currently runs on 11 UNIX platforms: SPARC SunOS 4.1.X, Solaris SPARC, Solaris x86, SGI IRIS, HP-9000 700/800, IBM RS/6000, Linux Intel, LinuxPPC (and MkLinux), BSD UNIX, Darwin (the UNIX underlying Mac OS X), and SCO ODT/UnixWare. The Win32 engine runs on Windows 3.1 (with Win32s), Windows 95/98/Me, and Windows NT/2000/XP. The MacOS engine runs on 68K and PPC Macs running MacOS 7.1 to 9.X. The Carbon engine runs on Mac OS X.

Contact MetaCard Corporation if you want to see MetaCard ported to other OSes.

4. What other products are most similar to MetaCard?

There are several products use the same language and object architecture as MetaCard. These include:

```
HyperCard (Mac)
Apple Corporation (APDA)
1 Infinite Loop
Cupertino, California 95014-2084
(716) 871-6555
(716) 871-6511 (fax)
hypercard_feedback@apple.com
http://hypercard.apple.com/

SuperCard (Mac)
IncWell DMG, Ltd.
P.O. Box 6761
Chandler, AZ 85246-6761
(530) 647-8541
(530) 647-8157 (fax)
```

info@incwell.com
http://www.incwell.com/

Serf (Mac)
Dan Gelder
2557 Center Road
Novato, CA, USA 94947
serf@best.com
http://www.best.com/~serf/

HyperSense (NeXTStep)
Thoughful Software
616 East Locus
Fort Collins, CO 80524
303-221-4596
info@thoughtful.com
http://www.thoughtful.com/

The product WinPlus, formerly Spinnaker Plus, has been discontinued by ObjectPlus Corporation. Oracle Media Objects (OMO), which was also based on the Plus technology, has also been discontinued. An early HyperCard workalike for Windows, Echelon's WindowCraft, was discontinued long ago largely because of the superiority of the Plus technology. Plus has in turn been surpassed by MetaCard.

MetaCard is also often compared with products like ToolBook, Director, Tcl/Tk, Perl, and Visual Basic because it can be used to develop the same kinds of products these other tools are used to develop.

5. How difficult is it to learn to use MetaCard?

Although HyperCard and ToolBook are frequently classified as end-user programming tools, make no mistake: scripting *is* programming, and to get the most out of products in this genre a substantial time investment is required. On the other hand, because of its high-level features and relative freedom from syntactic constraints, scripting in MetaTalk is far easier than C/C++/Java or Visual Basic programming, or writing scripts for awk/Perl/Tcl, any of the UNIX shell languages, or AppleScript. It also requires less training to become proficient at MetaCard than is required to learn so-called "no programming" authoring tools based on iconic development environments.

The MetaCard integrated development environment (IDE) makes is easy to develop complete applications with a minimal scripting, something that is not possible with tools like Tcl/Tk or the COSE Desktop Korn Shell (dtksh) in which even the user interface must be constructed by writing scripts.

MetaCard is currently targeted at technical end users; a level of computer experience somewhere between the average word processor user and the average 'C' programmer. Experienced programmers should be able to become productive with MetaCard in just a few hours (going through the tutorials should be sufficient).

Less experienced users can use MetaCard, but will require more time to come up to speed (days to weeks). The one exception is that when used as part of a large project, it is usually possible to generate template MetaCard stacks that even inexperienced users can use to quickly implement dialogs and menus.

6. MetaCard vs. HyperCard, SuperCard, ToolBook, etc.

Of these other applications, MetaCard is most similar to SuperCard. Most of the extensions SuperCard has made over HyperCard have been included into MetaCard. These include: support for color, managing dialog boxes, object (vector) graphics, multiple selection list boxes, and a wide variety of additional commands, functions, and properties.

The MetaTalk language has many features that make it superior to all of these other products, such as support for associative (string or number) arrays, call by reference parameters, extensive pattern matching and string formatting features, and the capability to handle much larger amounts of data (e.g., you can put multiple megabytes of text into fields). MetaCard's script compiler also provides much greater performance than is available with these other tools.

See the Benchmarks page for details.

MetaTalk supports custom (user-defined) properties, similar to those supported in ToolBook and Oracle Media Objects. It also has getprop and setprop "triggers" which are called when custom properties are set or retrieved, a important feature missing in OMO and SuperCard, and inconvenient to use in ToolBook.

MetaCard can import and run HyperCard 2.3 stacks as well as stacks produced with earlier versions of HyperCard. Due to minor differences between HyperCard and MetaCard, and between the Mac Toolbox and X11 or Windows (e.g., the use of different fonts), most stacks will require minor changes to make them function correctly after importing. Sounds, icons, and cursor resources are converted on input, but all other resources are discarded. The Mac version of MetaCard supports HyperCard format XCMDs/XFCNs, but use of these is discouraged because of their relatively poor performance and lack of portability. The MetaCard XCMD/XFCN interface is compatible with HyperCard's, so if you have the source code to the externals used in a stack, you should be able to use them with MetaCard with minimal changes.

Since the MetaCard development environment is very different from HyperCard's, scripts that alter the HyperCard development environment (by accessing the Mac menu bar or palettes or debugging features) won't run correctly in MetaCard. MetaCard's support of the HyperTalk doMenu command is rudimentary because of this limitation. Instead, MetaTalk has most of these features built in, and these commands/functions/properties should be used instead of doMenu.

There are many instances where the HyperTalk parser is more lenient than MetaTalk's. For example, the use of the word "the" in function and property names is more strictly enforced in MetaCard. Another potential trouble spot is that HyperCard allows predefined terms, such as function and property names, to be used as variable names, whereas MetaCard generally does not.

Since MetaCard has many more built-in commands, functions, and properties than HyperCard, it is much less tolerant of leaving quotes off of literals than HyperCard. The SuperCard-compatible feature "explicitVariables" allows you to set MetaCard's script compiler to flag unquoted literals as errors.

MetaCard is slightly *more* lenient than HyperCard when addressing objects. When a button or field is specified in MetaCard, both cards and backgrounds are searched, as a opposed to HyperCard where buttons are searched for on cards, and fields on backgrounds. You can override this behavior by using the word "card" or "background" in the object specification.

MetaCard Corporation encourages users to report any incompatibilities they find to help achieve the goal of making MetaCard capable of importing and executing a greater percentage of HyperCard stacks unmodified in the future. When in doubt, consult the MetaTalk Reference stack to verify whether a feature is supported or not.

A utility that converts SuperCard projects to MetaCard stacks is available in the file "sctomc.sit" on the MetaCard anonymous FTP sites. As with HyperCard stacks, 100% conversion is not possible, but significant problems with this converter should be reported to MetaCard Corporation as bugs.

7. MetaCard vs. Tcl/Tk and the Desktop Korn Shell (dtksh)

All three packages have the same goal: to make developing GUI applications faster and easier than is possible with conventional approaches based on C or C++ and the Xt/Motif toolkit. All three are based on an interpreted language with powerful string manipulation capabilities and have a simplified way of interaction with a GUI toolkit.

The packages differ in the way they are distributed, the complexity of their languages, the completeness of their development tools, their performance, and in their support for multimedia and object persistence.

Tcl/Tk is a free source-ware package that can be acquired via anonymous FTP from several Internet sites, although the commercial (and expensive) Tcl Pro development environment is required to do serious development with it. A C development system is generally required to compile and install it. The Desktop Korn Shell is distributed with the COSE desktop system by most computer vendors, though it is also available as a commercial product from additional companies. MetaCard is a commercial product available via anonymous FTP and CDROM from MetaCard Corporation and distributors worldwide.

MetaCard has the simplest scripting language to learn and use, and is also the most complete language since it has a larger number of built-in commands and functions. Tcl and ksh are syntactically much more complex, and require complicated quoting, parenthesizing, and other syntactic hurdles for most common operations. They also lack support for audio, video, animation, hypertext, and an object oriented message passing architecture.

MetaTalk scripts run up to 30 times faster than equivalent ksh scripts and significantly faster than equivalent Tcl scripts in most cases. This is due to the fact that MetaCard uses a virtual compilation architecture similar to that used by Perl and Java, whereas ksh use a conventional interpreter and the Tcl language design makes it impossible to compile many scripts even when a byte-code compiler is available. See the Performance Benchmarks page for details.

MetaCard and dtksh both have the Motif look and feel. Tcl/Tk looks similar to Motif in some ways, but is very different in others. dtksh, since it is based on Xt/Motif, is the largest and slowest of the toolkits. It also lacks the support for object graphics found in the other two packages.

MetaCard is the only product with a full featured IDE (Interactive Development Environment). With the others, design and layout must be done by writing scripts, which takes far longer. The free versions of these tools also lack such essentials as a script debugger and packaging tool for building standalone applications, whereas MetaCard includes all of the key features found in the relatively more expensive Tcl Pro product including a syntax checker, a script debugger, and an application packager. MetaCard also has the best cross platform support (dtksh is not portable to non-UNIX systems, and Tcl/Tk support is relatively limited on Windows and very weak on MacOS).

MetaCard is also the only tool to support object persistence. With the other tools, all information such as the text in fields and the state of buttons must be saved and loaded using the scripting language. This is a tedious and time consuming process for all but the most trivial applications.

A more extensive comparison of MetaCard with Tcl/Tk and the Desktop Korn Shell (dtksh) appeared in The X Resource Issue Eleven (Summer 1994), an O'Reilly & Associates publication (707-829-0515) (this article is also available as Interactive GUI Development Environments.

8. MetaCard vs. Gain Momentum and Icon Author

These three packages are used to develop interactive multimedia presentations and computer-based training. The products differ widely in price, but also in scale.

MetaCard and Gain Momentum both are scripting language-based, whereas Icon Author is based on a visual programming metaphor. Which of the two approaches is better depends a lot on the particular task. Categorizing the task is not easy, but is essential if the proper tool is to be chosen.

If the presentation or training package is relatively simple, or if the information to be presented is well structured, the icon-based approach taken by Icon Author and Authorware and similar products can be a very efficient way to build a package. There are many potential obstacles in this approach, however, and some of them are insurmountable. See the Testimonials page for one example.

If the information to be presented must be linked by hypertext links, if the path through the presentation must be changed dynamically, or if the user interface of the presentation must simulate an actual application, the scripting language / GUI toolkit approach is usually better.

The primary differences between MetaCard and Gain Momentum are price and scope. Gain Momentum costs much more than MetaCard and run time fees must also be paid. A Gain Momentum presentation also requires several times the RAM and disk space a comparable MetaCard presentation requires. Gain Momentum does include a more extensive range of file format conversion utilities and a full object-oriented database, and so may be a better choice for the development of very large systems.

9. MetaCard vs. FrameViewer and HyperHelp

All three products are used to deliver on-line documentation, in some cases as context-sensitive help for another application. All three products support hypertext links, and can display text using multiple fonts, sizes, and colors, and can display images and other graphics.

The primary advantages of MetaCard are its lower cost and greater flexibility since it has a scripting language, which the others lack. The primary advantage of the other products are that they can use the same source as is used to print hardcopy documentation.

10. MetaCard vs. UIM/X, TeleUse, XDesigner, etc.

All of these products are used to design and lay out GUI applications. The primary differences is that MetaCard is a complete, integrated environment, whereas the others are merely layout tools (UIM/X and TeleUse have limited interactive capabilities, but nothing near the scale provided by MetaCard (or Tcl/Tk or dtksh, for that matter)). The other tools require that complicated and bug-prone C or C++ code be written to manage the user interface. You also can't use the tool itself to edit the tool. For example, such MetaCard dialogs as the "Utilities" stack couldn't be developed using these other tools.

In addition, these other tools cost a lot more than MetaCard, and since they use the poorly designed Xt/Motif toolkit, they are much slower, require considerably more computing power, and are harder to learn and to use than MetaCard. Applications developed with Xt/Motif are also not portable to other platforms.

11. Where can I get MetaCard?

If you have anonymous FTP access to the Internet, you can download the free MetaCard Starter Kit (which includes complete on-line documentation and is fully functional except that it places a limit on the length of scripts you can create for each object) from ftp://ftp.metacard.com/MetaCard/ or ftp://ftp.uu.net/vendor/MetaCard. You can also get it from the Get MetaCard page.

The distribution can also be emailed, or sent out on CDROMs or DAT tapes (a nominal charges applies for the latter two). Contact MetaCard Corporation at info@metacard.com or call 303-494-6763 if you want to get the distribution either of these ways. Be sure to specify which platform you'll be using MetaCard on if you want MetaCard emailed to you.

After installing the software, you can license it by following the directions in the Licensing stack (available from the MetaCard Help menu). Your initial MetaCard license includes 1 year of unlimited technical support via email and free upgrades for 1 year. After that, you'll need to renew a subscription as specified in Question 14.

12. How do I get a license for MetaCard (PO, payment terms, etc.)?

See the Licensing MetaCard stack (from the Help menu) for pricing and terms. For most companies, the process is simple: fax a PO to MetaCard Corporation (303-499-9855) and email in the License Form with the PO number in the appropriate field. You'll get the key via return email, usually within a few minutes.

13. What kind of license should I (we) get?

The single-user license requires that the name of a user be supplied. Though there is no license manager, node locking, or user-name locking, only the specified user may legally use the single user license. The licensed user can use the software on any machine or combination of machines. If the license must be transferred at any point, a new user must be specified and a nominal fee paid.

With 5 and larger packs, individuals names do not have to be supplied. A single key is supplied, and copies are made of the license Home (mchome.mc) stack. It is the responsibility of

the site-support contact to ensure that no more than the licensed number of Home stacks exist on the sites machines. If more than 1 user will be using MetaCard, or if the licensed user will change frequently, a 5 or larger pack is therefore strongly recommended.

Any license can be upgraded to a larger license at any time by paying the difference between the prices of the smaller and larger licenses. Licensees granted the 50% educational discount who later wish to sell applications that they have developed with MetaCard can upgrade to a commercial license by paying the difference between the educational and commercial price.

14. How do I upgrade to a new version of MetaCard?

Your initial MetaCard license includes support and upgrades for a 1 year period. After that period, you'll need to renew your subscription and/or support contracts for each 1 year interval. A subscription includes free upgrades to all versions that are released or go to beta test within that 1 year period. At least 2 upgrades are guaranteed to be released within the duration of each subscription.

Subscriptions are priced yearly according to the table below, and educational institutions will receive a 50% discount from these prices. Unlimited technical support via email is available separately priced according to the table below, and the 50% discount *does not* apply to this service. All pricing is in U.S. Dollars.

	Subscription	Technical Support
Single user	300	200
5-user	1000	300
10-user	1500	300

To upgrade, you should relicense the new Home stack rather than continue to use your old one. It's technically possible to use your old Home stack, but the color and custom properties set in that stack may conflict with the new development environment rendering it ugly at best and inoperable at worst. See the directions in the file README.install for details.

15. Are there international versions of MetaCard?

MetaCard supports both Latin-1 (8 bit) and 16 bit character sets on all platforms, as well as the X11R5 Xmb calls for mixed-size text on UNIX systems, but the current development and documentation stacks (mctools.mc and mchelp.mc) are only available in English. Creating versions for other languages would be relatively straightforward, no such translations are expected to become available in the near future.

16. Are any examples of applications built in MetaCard available?

Some applications and scripts that have been contributed are available for anonymous FTP from ftp.metacard.com, directory MetaCard/contrib. Additional contributions are actively sought, and should be addressed to support@metacard.com.

17. Is there a MetaCard mailing list? An archive site?

The unmoderated MetaCard mailing list metacard@lists.runrev.com is maintained by Runtime Revolution Ltd. All MetaCard users are strongly encouraged to join the list, to seek advice when needed and to share their knowledge when appropriate.

To subscribe, go to http://lists.runrev.com/mailman/listinfo/

The messages posted to the various incarnations of the metacard list are archived at http://lists.runrev.com/pipermail/metacard/, http://www.mail-archive.com/metacard%40lists.runrev.com/ and http://www.mail-archive.com/metacard%40lists.best.com/.

If you have comments or questions about the list, please address them to metacard-admin@lists.runrev.com.

18. Where can I read about MetaCard?

MetaCard press releases have been covered in most major computer magazines since it was first released in 1992. MetaCard was named one of the Hot Products for 1996 in SCO World Magazine (January 1996), 10 Best Products of 1992 by UNIXWorld Magazine (January 1993), and has been reviewed in Sun World (January 1993), UNIXWorld (May 1993), InfoWorld (Sept

14, 1998, available on the InfoWorld WWW site), Inside Solaris (December 1999, available on the ZD Journals WWW site), and The Internet Eye (January 2000).

An article on a MetaCard application for Linux appeared in the November 1997 issue of the Linux Journal. An extensive comparison of MetaCard with Tcl/Tk and the Desktop Korn Shell (dtksh) appeared in The X Resource Issue Eleven (Summer 1994), an O'Reilly & Associates publication (707-829-0515). This document, titled Interactive GUI Development Environments, is available as Interactive GUI Development Environments.

19. Where can I get documentation on MetaCard?

All documentation for MetaCard is available on-line. A book form of the on-line documentation is also available. See the "Order Forms" stack from the "Licensing MetaCard" stack in the MetaCard release for details.

Many of the books on HyperCard can be used to supplement the MetaCard documentation, since the scripting languages are compatible. Some examples include the 3rd Edition of Danny Goodman's best-selling "The Complete HyperCard 2.0 Handbook", Dan Winkler's "HyperTalk 2.2: The Book" and "Cooking with HyperTalk 2.0", and "HyperProgramming - Building Interactive Programs in HyperCard" by George Coulouris and Harold Thimbleby.

A contract was signed for third party book by a major publisher a while back, but unfortunately the author missed several deadlines and the contract was canceled.

20. Can I redistribute my stacks? Are royalties required?

The MetaCard engines can be distributed without licenses, so stacks that you produce can be distributed without any special arrangements with MetaCard Corporation, whether you are using the Starter Kit or a licensed Home stack. When stacks other than Home stacks open, no license screen is shown. Stacks can be run directly from the command line on UNIX systems since they have a shell script tacked on to the front of them, and can be built into single-file applications on all platforms which renders them indistinguishable from stand-alone applications.

21. How do I redistribute my stacks?

First, clean out the stackFiles property, which is what MetaCard uses to locate other stacks by name. Choose "Stack Files" from the Edit menu and press the "Reset" button. Then, save your stacks into separate files using "Save As...".

You'll need to distribute an engine together with your stacks. You'll also need to distribute the mctools.mc stack if you use any of the standard dialogs or bitmaps (icons or cursors). You can also use the built-in "Resource Mover" to move these objects into your stack instead.

If you prefer a single-file distribution, you can use the "Standalone Builder" to bind your stack to the engine and make a single executable file.

22. Can I call 'C' functions from MetaCard?

On UNIX systems, the file external.tar.Z in the standard distribution has templates for doing this. These same templates are placed in the Externals folder in the Win32 and MacOS distributions. MetaCard can call commands and functions in the external, and these external 'C' routines can call back to the MetaCard engine to get and set variables and to execute commands.

See the MetaCard Tutorials stacks for a demonstration of the external capability and instructions on how to create and modify externals.

23. What is embedded MetaCard (libmc.a)?

The MetaCard distribution comes with a template for calling C functions and a tutorial that explains how to use it. On UNIX systems, this standard system spawns the external C functions as a separate process and uses X properties to pass information between this external and the MetaCard engine. It is suitable for most applications that need to call C functions to access databases or specialized libraries. Due to the communication overhead, it probably isn't suitable for applications that need to call C functions repeatedly in loops. It can also be difficult to debug externals since they're run as separate processes, and so developing large externals can be very time consuming.

To remove these limitations the MetaCard engine is also available in an embedded version on UNIX systems. This library (libmc.a) can be linked directly to applications which can call MetaCard functions with only normal function call overhead (which can be up to 40 times faster than communicating via properties), and which can be debugged using ordinary debugging tools.

On Win32 systems, externals are built as DLLs. Since debugging DLLs is relatively straightforward, and since performance isn't a problem with this architecture, Embedded MetaCard is not available for Win32 systems. For MacOS, externals are stored as code resources, which also are relatively easy to debug and don't have major performance penalties associated with them.

24. How do I communicate with MetaCard from my Motif application?

The XT program in the external.tar.Z file is an example of this, and can be used as a template for using MetaCard as a help system for your Motif application.

25. What printing support is provided?

MetaCard 2.0 supports production of PostScript files of cards images on UNIX systems. These files can be printed on most PostScript printers. The standard system printing features available on Windows 95/NT and MacOS are accessible from the MetaCard engine for those platforms.

You must use the "Print Setup" dialog to set the printing parameters before printing cards. See the help for that dialog and the card describing the print command in the MetaTalk Reference for more information on printing.

26. Does MetaCard support 2D or 3D graphics?

2-d vector (object) graphic capabilities similar to what is available in SuperCard and ToolBook are included in MetaCard. The property names are the same as those used in SuperCard where possible. For example, you can set the points property of a polygon style graphic to dynamically change the way it looks. See the demo stacks for examples using the graphic object.

You can also import EPS graphics on systems that support the DPS X server extension (RS/6000, SPARCSVR4, Alpha).

No support for 3D graphics is provided, but can easily be added using externals. See the external tutorial in the MetaCard Tutorials stack for and example of how to draw into MetaCard windows from external C functions.

27. Does MetaCard have a script debugger?

As of MetaCard 2.0, a complete GUI debugger and profiler is included with the MetaCard development environment.

Other common debugging techniques include using the put command to put things into the Message Box (equivalent to using printf with 'C' programs), using "write <expression> to stdout" to trace scripts by writing to the terminal window that MetaCard was started from, and using the "watch" command to watch variables. For more complex programming tasks, the external command interface can be used to produce C or C++ back ends to MetaCard user-interfaces.

28. Can I use MetaCard as a database?

The multiple card metaphor combined with full text search makes MetaCard an ideal platform for building small databases that contain unstructured information. Large databases (more than 1000 records) or those that require multiuser protection would best be implemented via external commands and a standard database package.

29. What are the system requirements for running MetaCard?

MetaCard takes less memory than just about any other UNIX GUI development system, especially anything based on the UNIX/X11 Xt/Motif toolkit (MetaCard is a C++ application built on a C++ graphical toolkit). It performs adequately on 8 MB CISC systems. On RISC systems, 12-16 MB RAM is recommended. Disk space requirements are also small: the entire distribution requires less than 5MB, the largest piece of which is the demo stack (which is not required for development).

30. How do I play audio/video in MetaCard?

MetaCard supports audio on systems with built-in support and on some systems with optional support. The recommended format for cross-platform use is 8-bit 8 kHz mulaw (the Sun/NeXT native format, also known as ".au") as this is the only format supported on all platforms. Higher quality formats such as AIFF and WAV are supported on Windows and Mac systems, but not on most UNIX platforms. The "play" command plays audio from a file or from an audioClip that has been imported into a stack.

The Windows 95/NT engine can play AVI files using Video For Windows as well as any format supported by QuickTime 3.0 (and later) if QuickTime is installed on the target system. The MacOS engine requires QuickTime 3.0 or later to play movies. The UNIX engines can play QuickTime, AVI, MPEG, and FLI/FLC animations, but only a limited set of QuickTime codecs is supported (specifically, Sorenson is not supported, nor is QTVR). The files can either be played directly from disk, or imported into MetaCard stacks before playing to reduce startup times on systems with sufficient memory. MetaCard has properties for scaling the movies, playing them back at particular locations within a window, changing their frame rate, and leaving the last frame of a movie on the screen (which can be used to make transitions between movies smoother). See the documentation for the "play" command and "player" object for more information.

There are several commercial video/movie players available, all of which can be made to work with MetaCard. MetaCard has properties and commands that allow starting and stopping these players, playing the movie into a MetaCard window, and sharing colormaps between the players and the MetaCard engine. The Win32 engine has an mciSendString command that can be used to send any command to any MCI-compatible device.

31. What are substacks and how should I use them?

During development, it's usually best to keep each stack in its own file as this eases making backups of stacks and opening the backups after having loaded a newer version. When you are nearing the release the application you're developing, choose one stack to be your main stack and use "Main Stack" dialog (from the "Stack Properties" dialog) to make some or all of the other stacks substacks of that stack. Which stack to choose as the main stack depends on the project, but it's often convenient to make it a license splash screen that comes up when the application is run and then closes itself and opens some other stack.

Remember that all messages sent to any of the substacks pass through the main stack. This means you have to be careful to check the target if you have any handlers for any of the

standard MetaCard messages. This feature does make main stack scripts a good place to put handlers that are used in several of the other stacks in your application.

32. Which stack is "this stack"?

Many MetaCard dialogs (notably the Message Box and the menus) set the defaultStack property to the topStack before executing their scripts. The defaultStack property determines which stack is used when the chunk expression "this stack" is used (or the stack expression is omitted entirely) in a script.

The topStack is the top-most editable stack. It will usually have a page number in parentheses and may also have a "*" (denoting that the cantModify property is set to false) in its title bar. To avoid confusion, it is recommended that only a single stack be edited at a time. Close or iconize/minimize/collapse the others.

33. How do I get back to the stack I was editing?

To reopen a stack you have closed (which normally will be neither saved nor deleted when you close it), you can type the following into the Message Box where "my stack" is the name of your stack:

```
topLevel "my stack"
```

If your stack is saved as a substack of another stack, you can also open it from the Components dialog from the Stack Properties dialog of your mainStack.

If this stack will be used frequently, add a button to the Home stack with the "topLevel" command in a mouseUp handler.

34. How do I save a stack into its own file?

Use the "Save As..." item in the File menu and choose "Yes" if a dialog asking you if you want to make the stack a main stack appears.

35. How do I open a stack in the same place as another stack?

You can't, at least not reliably. The major problem is that window managers such as mwm aren't required to honor an applications request for window placement. Even if it does, when you specify a position for a window, it may be offset by the width of the window borders, which vary in size depending on which window manager is being used: use a different window manager, get a different position.

It is a much better design to use multiple cards for your application rather than trying to outsmart the window manager by positioning a stack by setting its loc or topLeft properties in a script. Also see the "in the window" option to the go command.

36. How do I run other programs from a script?

Use the shell() function or the "open process" command. See the MetaTalk Reference and Concepts & Techniques stacks for details.

37. How do I use MetaCard to build non-graphical programs?

Any UNIX MetaCard engine can be used execute scripts stored in text files instead of in stacks (this feature is not available on Win32 or MacOS engines). By convention, these script files should have a ".mt" extension. The script is executed by making it the first command line parameter (e.g., "mc sample.mt").

The scripts should contain an "on startUp" handler, which will be called when the script is executed. The script can read from its standard input with the statement "read from stdin until empty", and can write to its standard output with "write x to stdout". The "put" command without a destination container will also write the expression to stdout (normally it goes into the Message Box in this case).

38. How do I import images?

Use the "Importer" dialog box (from the File menu). MetaCard can import GIF, JPEG, BMP,

XPM, XBM, XWD, and all PBMplus image formats. See the "import" and "export" commands in the MetaTalk Reference stack for details.

The easiest way to import images in MetaCard is to use the snapshot feature: open the image using the application used to create it or an image viewer application, and then import a snapshot of it with the "import snapshot" command or the "Importer" dialog.

The PBMplus image conversion package is a freeware package that will convert most common image formats on UNIX systems. It can be acquired from many Internet archive sites. It is also available on many CDs including the UNIX Power Tools book/CD (O'Reilly & Associates/Bantam Book ISBN 0-553-35402-7) and most Linux CDs.

If you want or need a commercial image conversion product, Image Alchemy from Handmade Software may do the trick. If you need to convert images produced on a Macintosh, DeBabelizer from Equilibrium may be a better solution.

To avoid running out of colors, in which case MetaCard does a simple nearest-color remapping of the image, consider quantizing the images to reduce the number of colors they use. Remapping all of the images to use the same colormap is also a good way to reduce the number of colors required. A good rule of thumb is to use no more than a total of 160 colors for all of the objects in a stack.

39. How do I put an image into a button (icon)?

Any image can be used as an icon, though transparent GIF is the format most appropriate for them . The most common technique is to import all of the images that will be used as icons and fill patterns in that stack onto a single card of your stack. This card won't ever be seen by the users of the stack and so its appearance isn't critical.

On UNIX systems you can import non-transparent images and edit them to make them transparent. To do this, choose the bucket tool from the Paint Tools palette and click mouse button 2 in the background. This will will erase all connected areas of the same color.

Next, double click on the image with the pointer tool and write down the id of the image. Go to the card where you want the button to be and create a button. Make sure the button is selected and then set the icon property to the id of the image (1234 in this example) using the Message Box:

set the icon of the selectedObject to 1234

The same technique can be used to set the backPattern property of a button or any other type of object.

40. How do I create text that uses 16 bit or ISO 8859 characters?

To access high bit characters (upper 128 characters in an 8 bit font) on UNIX/X11 systems, you can use the mod3 key to toggle MetaCard fields to enter characters with the high bit set. Use the program "xmodmap" to determine which keyboard key is bound to to mod3 and to bind one if there isn't one already. Unfortunately, learning the key binding can be difficult.

If you only have a few characters to enter, the easiest way is probably using the "Character Chooser" or the numToChar() function (on UNIX/X11 systems, you can get the number of the character you need by displaying the font with the "xfd" program).

To access the fonts for other languages (e.g. those that use 16 bit characters), you'll need to set the textFont property to the font name in the "Object Font" dialog, or by typing it into the Message Box. To specify a font set on UNIX systems (for 16-bit fonts), put all of the fonts required (separated by commas) in the textFont property.

For example, to make the first line of a field be displayed in the Kanji24 font, you'd type: set the textFont of line 1 of field "some field" to "kanji24". Use the X program "xlsfonts" to find out which fonts have been installed on your system. Note that isn't possible to edit 16 bit text, you can only import it from a file produced in another editor, or by using "quick paste" (select the text in editor window and click mouse button 2 in the MetaCard field where you want to paste the text).

Support for Unicode and other 16 bit character sets is not available in the Win32 or MacOS engines yet.

41. Why is there more than 1 background?

To provide maximum flexibility in control and script placement, a card can have zero or more backgrounds. All messages sent to the card (and controls on the card) are passed through each of the groups on that card.

Without a background, everything goes on the card layer. Multiple backgrounds can be created and the "Edit Backgrounds" dialog box can be used to place them on or remove them from cards.

Note that "groups" and "backgrounds" are the same thing. The only difference in the use of the terms is that when addressing objects, "groups" are accessed relative to the card and "backgrounds" are accessed relative to the stack, so group 2 is the second group on the current card, whereas background 2 is the second group in the stack, and may not even be used on the current card.

42. How do I edit the standard dialogs and menus?

Any of the supplied dialogs can be edited by bringing up the dialog in topLevel mode (type 'topLevel "Stack Name"' into the Message Box). You may also have to unset the cantModify property (in the Stack Properties dialog).

To save any changes you make, be sure to save the tools stack, since these dialogs are not part of the Home stack (type 'save stack "MetaCard Menu Bar"' into the Message Box to save the tools stack. Note that the file mctools.mc must be writable for this to work).

43. How do I build menus? How about combo boxes?

There are four ways to implement menus, combo boxes and other "multiple choice" controls. The easiest way is to set the style of the button to the appropriate value and to enter the strings that will be displayed in the menu into the "Contents" field in the "Properties" palette. When the user chooses one of the items in the menu, a menuPick message will be sent to the button with the name of the menu choice as an argument.

The second way is to set the "menuName" property of the button to a stack name (any stack can be used as a menu panel). See the MetaTalk Reference stack for details on this command and property. The "Menu Properties", accessible from the "Button Properties" palette supports setting many menu-related properties. Note that the easiest way to build these menu stacks is to clone the existing menus. See the Concepts & Techniques stacks for tips on how to do this.

The third way is to simulate a menu panel using a field (or group) that you hide and show. Finally, you can use the "popup" command to open a menu panel over any control.

44. Where is the menu separator control?

There is no menu separator control, separators are really 4 pixel tall buttons with their autoArm properties set to false. The "Lay Out Menu" button in the Utilities stack will size buttons into separators if their initial size is below a certain threshold.

If the first character in a line in the button contents is a - character, the menu created by that button will have a dividing line at that point.

45. "Save as" doesn't seem to work. Why not?

When you save a substack, it is saved *with* its mainStack into a file. If you then reopen the stack, what you will see is the mainStack, *not* the stack you were editing. The general rule: unless you are preparing to distribute your stack, don't use "save as..." to make a backup of

it. Instead, use the normal OS file copying feature to make a copy of it, or use the "clone" command and save the clone into a new file.

46. Why can't start editing groups or edit button scripts?

Strange behavior can occur if there are multiple stacks with the same name loaded. Try renaming one of the stacks and then try use 'topLevel "oldname"' to see if a copy of the stack exists.

47. Cut/copy/paste doesn't seem to work. Why not?

Cut/copy/paste of text selections doesn't work if you have your window manager set up to use pointer focus. See the README.install file for instructions on how to change it back to explicit/click-to-type focus.

Strange behavior can occur if there are multiple stacks with the same name loaded. Try renaming one of the stacks and then try type the following into the Message Box to see if a copy of the stack exists:

topLevel "oldname"

48. Why does my text selection keep disappearing?

Most likely your window manager is set to use pointer (implicit) focus. MetaCard is an active-focus application, which means that a stack only takes the keyboard focus when a text field within the stack needs it. Window managers that operate in focus-follows-pointer (implicit) focus mode give the focus to MetaCard windows that have no use for it, removing the focus from windows that do need it.

Pointer focus is an obsolete technology inherited from the old Sun and Apollo workstations where graphical displays were used primarily as a way to open multiple terminal windows. It does not work well with true GUI applications which open dialog boxes and support keyboard traversal among controls within the dialog boxes. Therefore, using explicit (click-to-type) focus policy is strongly recommend with MetaCard. Explicit focus is the default for mwm, olwm, NeXTstep, the Macintosh, OS/2 Presentation Manager, and MS-Windows. Instructions for reconfiguring olwm and mwm to use it can be found in the file README.install.

49. Why does the selection change size every time I click?

Most likely the Caps-Lock key is down.

50. Why can't I type text or use the backspace key in fields?

Either the Caps-Lock key is down, the Mod 3 key (sometimes bound to Scroll-Lock or Num-Lock, run the "xmodmap" command to determine how these keys are bound), or the window is not accepting the focus. This last problem occurs with broken (or obsolete) window managers such as olwm and twm running in pointer focus mode. Please use explicit focus if possible (see the file README.mc for details), or use the -pointer option to the mc command.

51. Why is editing text sometimes sluggish?

If you have an image or EPS object that overlaps a field, part of it will be redrawn after each character is typed. Set the layer of the image (or the field, or the group it is in) such that the image is under the field.

To improve performance and reduce memory usage on both the client and server, you should also crop your images down to the minimum size necessary. To crop an image, click on a painted on area with the pointer tool, and resize it using the resize handles.

52. Why do my images look washed out or miscolored?

MetaCard uses the system colormap unless the global property "privateColors" is set to true. If you're running another application that uses lots of colors (WWW browsers are notorious color hogs) Note that if you set the privateColors property to true, it may cause all the other applications on the display to "go technicolor" when MetaCard gets the keyboard focus.

The best solution to this problem is to use the same colors for all of your images, and to

select colors for all the other objects from this palette. You may also want to color quantize any images in your stacks to reduce the number of colors they use. Most image conversion packages have filters for performing these operations.

53. Why does the display flicker when I select text or move controls?

Starting with the 2.1 release, stack windows are not automatically buffered with off-screen bitmaps/pixmaps. The buffers are automatically created when objects are selected, or when the "move" or "visual effect" commands are executed.

You can force a stack to be double-buffered at all times by setting the "alwaysBuffer" property of the stack to true.

METACARD TESTIMONIALS

Here is what some MetaCard users have to say about the product. More unsolicited testimonials can be found in the MetaCard-List Archives. Search for "CBT" in the 4Q98 archive, for example.

MetaCard is an awesome authoring environment; it is truly a superset of both HyperCard and SuperCard, and provides a load of great features for your stacks. If you haven't DL'ed the demo version, I would highly recommend doing so.

Ken Ray
Sons of Thunder Software

After six months of using MetaCard, I still marvel at the amount of power it delivers. So far have have used it to develop a cross platform multimedia application, implement local network functionalities and now add internet support. Given that I was never formally trained and have never written a line in C, this is simply amazing. I can't thing of any other development tool that would allow me to do that much with that much ease.

Pierre Arnaud
Ecole Supirieure de Commerce de Paris

Man... You people have a great product!

Mike Gehl
TOTO Multimedia

A general thing I'd like to say: MetaCard is wonderful, and complex. We have found at least as many bugs or weak points in OMO, but there was nobody that would answer our requests there. We found some bugs in HyperCard, but lots of weak points, where we had to work a long way around. MetaCard is the most powerful and complete xTalk environment I have seen.

Ruediger zu Dohna
GINIT GmbH

We just finished up our latest CBT course using MetaCard and are having great reviews. Thanks for providing a product that makes development under Win32 with deployment under UNIX/X an enjoyment and practicality.

Steven Seidl
National Cryptologic School

MetaCard is the multimedia cbt authoring tool of choice for SGI's Education, Design, & Technology (ED&T) group. Using MetaCard has increased our productivity and creativity.

My group had originally tried to develop our product using IconAuthor (IA). We had hoped that IA would have a low learning curve and that IA programs would be easier to maintain. In the final analysis, however, I would say that if you have experience programming or experience authoring in HyperCard, then learning MetaCard will not be any huge task. ...and as far as maintainability of your MC application that entirely depends on how you design the MC stack. I have found my MC application to be *easier* to modify than the equivalent IA program.

MC's service is not to be beat. MC would respond to my questions in about 2-8 hours (including weekends). 98% of these responses resulted in complete resolution of my problems. I have found MC to be knowledgeable, prompt, and reliable. New MC updates come often and even betas are reliable working versions. MC's understanding of UNIX is exceptional.

Lori Pfeifer
Silicon Graphics

I have used a wide range of Unix development tools on many types of Unix platforms. MetaCard is the best cross platform tool I have ever used. I have been developing software for over 20 years. MetaCard has exceeded all my expectations and has made it possible for us to provide a easy to use GUI front end for our systems management product which distinguishes us from our competitors.

Rich Morgan
Tactix, Inc.

Thanks very much for your help. I must say that MetaCard has the fastest and most efficient support (compared to many large software vendors that I deal with everyday). You have a good customer service which is what it takes to make a product work.

Eric Rosevear
Sgs-Thomson

For your info, I have just posted off an e-mail order form to upgrade from our single MetaCard license to 5 licenses.

This reflects confidence here that we will be able to sort out our timing/synchronization issues, and that MC will become the standard product in our department for academics (and particularly graduate students) to program many of their experiments.

Mark Randell
Psychology Department
University of Western Australia

Honestly, I find it difficult to express how much I appreciate MC 2.1 (especially the Win32 engine!). ISI has some interesting projects coming up in the next six months - a couple of internet-enabled multimedia CDs. I hope to use MetaCard to make them happen. ToolBook isn't even in our vocabulary :) but we *are* looking at Quest and/or mTropolis for parts of the projects. While each one has strengths, neither has the full range of functionality that MC has. (I'm lobbying for MC all the way. Could you guess?)

Thanks for giving us such a great tool to work with.

Phil Davis
Information Synthesis

A big advantage of using MetaCard to develop software is that MetaCard makes it easy for me, the application programmer, to write my own programming tools. One reason this is a big advantage is I need not be limited to the tools that come with MetaCard. Another is that the tools I write for myself work exactly the way I want them to.

I state this from experience. I've just finished writing a stack explorer. This tool can walk any toplevel stack and ferret out what I want to know about the stack's object tree--in particular,

it detects when an object has a script and extracts the script's text. I can view the information the explorer collects in a scrolling field and I can save it to disk.

Of course, having the capability to add tools to MetaCard would be of little use if it were difficult or time-consuming to do so. Well, it isn't. Because of MetaCard's open architecture and powerful debug tools (interpreters are always the best debuggers), I went from concept to debugged code in less than three days. When you consider that I'd only been using MetaCard for three months at the time, with no previous HyperCard or SuperCard experience to draw on, it says a lot about how easy it is to become productive with MetaCard.

Morton Goldberg
ERIM International, Inc.

Just a short note to tell you how appreciative I am of the speed with which you answer questions and try to help people with MetaCard. I have been a long-time SuperCard user but I am more and more impressed with MetaCard and with the cross-platform capability, I plan to use it almost exclusively in the future.

Good job!

Philip Chumbley
Manna

I thought I'd tell you what I think after my first look at the MetaCard Starter Kit so far. You may quote this anywhere you like as long as you don't quote out of context. First, I have to apologize. Whatever I might have said about MetaCard's stack and substack metaphor -- forget it. Once you have used it, it's great! The product is quite powerful and extremely versatile (and how fast it is!). I went to trying out some stuff and whoof! there it is. And come to think that the editing environment runs so smoothly even though it's all done in MetaTalk!

M. Uli Kusterer

Thank you for providing a way for me to use HyperTalk on both platforms, and in such a powerful implementation.

It's been a long wait, and well worth it.

Richard Gaskin
Fourth World

I just unleashed 36 students on the project I finished converting from HyperCard to MetaCard. I posted .sea and .zip files of the Mac and Windows versions (standalone plus a couple dozen other stacks the standalone uses or opens) on the class web page and in a Mac lab folder. Students downloaded/copied and ran the project on a variety of computers, usually in a campus Mac lab or campus Windows NT lab, or on their home computers (mostly Windows).

In general, I was extremely impressed (and relieved) that everything worked so well with a large number of students operating it on several types of PowerMacs and Win32 machines at school and home! MC is fantastic!

Richard Herz
University of California, San Diego
(for more on this project, see http://www-mae.ucsd.edu/research/herz/reactorlab/.)

METACARD YEAR 2000 STATEMENT

MetaCard 2.1 is year 2000 compliant, but only if the developer using it specifies 4 digit date fields for all dates. MetaCard 2.1.3 has been adapted to use "35" as a century cutoff date, and so will more likely work properly even with the 2-digit date fields in the default short form of the date() function. In MetaCard 2.2 and later, the century cutoff date is user selectable.

Guide to High-Level Development Tools

Higher developer productivity is one of the primary benefits of a high-level development tool. Higher productivity results in lower development costs, shorter development schedules, lower maintenance costs, and more highly-featured and usable end products. It also means that less time is required to learn and use the tool, since there are fewer low-level details that need to be mastered before the actual work on a project can begin.

It can be difficult to determine whether or not a given language and development environment will support high productivity, however. Here is a list of questions you should ask before choosing a development tool. A tool with all of these features will support high developer productivity, and using such a tool for your next project will cut a considerable fraction (perhaps even a majority) of the time it would take to develop the product using a lower-level tool:

1) Can you execute a string containing language statements?

The ability to build up a list of commands as a string and then execute that string using the language interpreter is probably the single most important feature of a high-level language. Although it's not used very often, this one feature can save writing dozens or even hundreds of lines of language statements. In MetaCard, the "do" command can be used to execute strings, and the "value" function can be used to evaluate strings as expressions. Perl, Tcl, and ksh have similar features. The various BASIC dialects do not. Neither does Java, and by it's very design it is impossible to implement a feature like this in Java.

2) Can you refer to a variable as either a string or a number?

Since most applications require the manipulation of both strings and numbers, an important high-level feature is the ability to operate on a variable as either a string or a variable without having to declare its type or perform a separate conversion operation on it. MetaCard, Perl, ksh, and Tcl all support this feature, but the latter two languages have very low performance in numeric-intensive operations. Visual Basic supports a "variant" type that has this feature but there are many restrictions on its use. Java does not have this feature.

3) Can you refer to words and items in a string with single expressions?

There are many times when being able to get the words or items (elements separated by commas) or lines in an input stream one at a time is useful. With most low level languages (like C/C++/Java/BASIC), you have to write a loop to scan the string a character at a time looking for delimiters and then copy the substring. With MetaCard, you just write something like "word 2 of line 3 of somevariable".

4) Can you change the layout of the controls in and application and change scripts without having to restart the application?

With most low-level development tools, and even high-level tools like dtksh and Tcl/Tk that lack Integrated Development Environments (IDEs), every time you have to change the user interface or scripts of an application, you have to stop running the application, make the change, and then restart it from the beginning to see the changes. Even worse is that with low-level languages (like C/C++/Java/BASIC), you have to wait for a compilation phase to complete before you can restart the application. Since most changes are made to areas in an application that require several actions to get to, the developer must execute those actions over and over again while developing and debugging the application. This process is an enormous waste of time, and takes most of the fun out of developing applications.

MetaCard has the very unusual ability to save all of this tedious restarting and navigation through the interface to get to the part under development. Instead, the developer can make changes to the layout or scripts in an application while that application is running. This saves enormous amounts of development time, and allows productivity unmatched by any other language or development environment.

5) Can you crash the program or system with a programming error?

The absolute worst use of a developer's time is waiting for a system to reboot. While this "feature" of development in third generation languages is most common on MacOS, there are many ways to render even Windows and UNIX systems unusable with them even if you can't

actually crash them. And sometimes programming errors in MacOS applications written in third generation languages cause a problem even worse than crashing: data corruption can occur that can cause the loss of some or all data on a hard disk. Even if it's only the program that aborts (called a GPF on Windows or a core dump on UNIX), you're still left with the poor productivity associated with having to restart the application to debug it. With MetaCard, programming errors simply cause an error dialog to appear that can be dismissed with minimal disruption of the task at hand.

A simple MetaCard calculator application

To give you an idea of what's involved in developing a GUI application using an interactive development environment, this article will describe the steps needed to build a simple application in MetaCard. The application is a simple calculator. To make the example even simpler, this calculator with actually just build up an expression that will be evaluated with the MetaCard "value" function when the "=" or return key is pressed. This saves the trouble of handling things like operator precedence (the "value" function handles this automatically) and allows us to implement a unique feature: a backspace key. Due to its simplicity, this sample application was developed in about 15 minutes from concept to complete implementation.

The complete application can be acquired with anonymous FTP from as calc.mc from ftp.metacard.com:/MetaCard/contrib/ or can be run directly from your UNIX WWW browser if you've got MetaCard configured to be a helper application. See the Get MetaCard page for instructions on setting up MetaCard as a helper application. After you've got MetaCard configured, just click on this Calculator link to run it.

The application interface is created using MetaCard's built-in Interactive Design Tool (IDT). Similar to other UNIX IDTs such as X Designer and Builder Xcessory and Windows tools like PowerBuilder and Visual Basic, the MetaCard layout tools allow you to create controls, position them, and change properties such as the colors and fonts used to draw the controls.

The first step in creating an application is to create a new window and name it. Next you populate the application window with the necessary buttons, text fields, scrollbars and other controls.

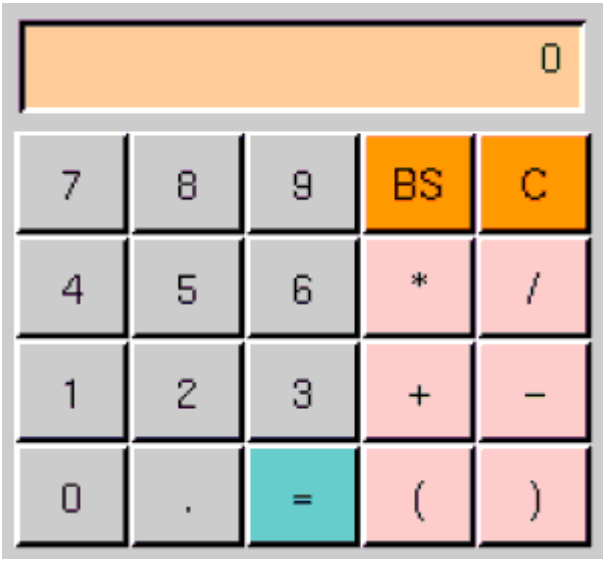

Figure 1: Screen Shot of the Finished Calculator

After the interface looks about right (Figure 1), it's time to add the scripts that make it functional. To add a script to an object, you open the built in script editor by clicking on the "script" button in the "properties" dialog for a selected object. These scripts are written in a language called MetaTalk, which is a superset of the HyperTalk language used in HyperCard.

A MetaTalk script is broken into functions called "handlers", each of which handles a particular "message". Messages are sent as the result of user action, or as the result of a call from another handler. For example, the calculator application script contains the following handler, which handles the messages generated when the user clicks on one of the buttons:

```
on mouseUp
  if word 1 of the target is "button"
  then handle_key the short name of the target
end mouseUp
```

Since in this application the buttons don't have individual scripts, the messages sent to the buttons are passed up to the buttons' parent, which in this example is the "card" (which is roughly equivalent to the main XmForm widget in a UNIX Xt/Motif application). This being the case, the card's handlers get called each time the user types a key or clicks in the application's window. While this architecture can be very convenient, there is one complication: the script must distinguish clicks on buttons from clicks on the "Display" field and from clicks on the card itself. Using the "target" function, which returns the type and name of the object the message was originally sent to, is one way to do this.

The complete script for the calculator application is shown below. Most of the work is done in the "handle_key" handler. It removes and restores the "0" that should be displayed when the display has no other value, and uses the MetaTalk "value" function to evaluate an expression that the user has entered. Note that all number-to-string (and back) conversions are done automatically, and there is no need to specify the type any of the variables. The next 4 handlers handle the messages sent when the user presses special keyboard keys, each of which generates a specific message. Each of these handlers in turn calls the "handle_key" handler with a corresponding key name.

```
# handle keystrokes and button presses
on handle_key which
#suppress leading 0
  if field "Display" is "0
  then put empty into field "Display"
  if which is "="
  then put value(field "Display")\
       into field "Display"
  else if which is "C"
    then put 0 into field "Display"
    else if which is "BS"
      then delete last char of field "Display"
      else put which after field "Display"
#put 0 back into display if needed
  if field "Display" is empty
  then put 0 into field "Display"
end handle_key

on returnKey
  handle_key "="
end returnKey

on enterKey
  handle_key "="
end enterKey

on deleteKey
  handle_key "BS"
end deleteKey

on backSpaceKey
  handle_key "BS"
end backSpaceKey

on keyDown whichkey
#make sure user pressed a valid key
  if whichkey is in ".0123456789+-*/=C"
  then handle_key whichkey
  else beep 1
end keyDown

# since the buttons don't have scripts, all the
# mouseUp messages end up here.  Check to make
```

```
# sure that user clicked on a button and not
# the card or "Display" field, then handle the
# key specified by that button (the short name
# is the text displayed as the button's label).

on mouseUp
  if word 1 of the target is "button"
  then handle_key the short name of the target
end mouseUp
```

A generic "keyDown" message generated when the user presses one of the alpha-numeric keys on the keyboard. A single parameter, the key the user pressed, is passed. In this keyDown handler, the parameter is put into the variable "whichkey". If the key pressed is valid (it appears in the validation string checked with the "is in" operator), it's passed on to the "handle_key" handler. If not, the application beeps and throws away the key.

To keep the application simple, no other input validation is performed. This isn't as serious a deficiency as it would be with a C based program, however. If the user tries to specify an illegal operation (e.g., divide by zero, or entering two operators in a row), the standard MetaCard error dialog appears when the "value" function is executed. While this isn't all that user-friendly, it's certainly superior to the core dump you'd get with a C program that does too little input validation.

If a more polished interface is required, it would be trivial to add checking for invalid operations. It is even possible to override the standard error dialog if you want to rely on the MetaCard run-time system to catch the errors, yet bring them to the users attention with a more informative message than is given in the standard error dialog.

Note that in this script the object specifications in MetaTalk (e.g., field "Display") is done by description rather than by specifying a variable as is done in Motif (and in most other scripting languages for that matter). The description can either be the name of the object, its id (a unique number assigned to the object when it is created) or by number. For example, "field 1" is the first text field that was created in this application with the IDT.

It is also possible to store an object description in a variable and use the variable to refer the object as is done in other languages:

```
put "field Display" into somevar
set the backgroundColor of somevar to "red"
```

The disadvantage of this technique is that the resulting code is much less readable code that uses descriptions, especially when the variable initialization is done in a different place than the use of the variable.

Testing and debugging

After typing the script into the script editor and clicking on the "Ok" button, the script is ready to test immediately. No need to run a compile or link, or to restart the application and run it to the point needed to test the new code. This makes it possible to develop the application iteratively. If a dialog box doesn't look right or behave intuitively, you can change it immediately. This is in sharp contrast to conventional tools where a lot of code will frequently have to be discarded if any substantial changes have to be made in the controls used to implement an application.

If there are serious errors in a script, the interpreter will stop and open a dialog box that points out the nature of the problem and ask if you want to fix it. If you click on the "script" button in the error dialog, the script editor with the offending script in it and will position the cursor on the statement that caused the error. You can then fix the error and continue immediately. This is in contrast to a typical Windows or Xt/Motif development environment where errors usually result in an application crash (or maybe even a system crash) which means you have to restart the application in a debugger and repeat the exact sequence of steps that caused the error. Worse still is the case where you have to recompile a module or library with symbols if it was originally compiled without them.

Deploying the application

As with most Interactive Development Environments, to deploy a MetaCard application, just two files are required: the stack (application) and the MetaCard engine (equivalent to a dynamic Xm library or DLL, but much smaller). Both the stack and the engine can be any place on the PATH. No .Xdefaults files are required on UNIX systems, so the installation procedure doesn't require the root permissions typically required to install application resource files in /usr/lib/X11/app-defaults. If your application uses MetaCard resources such as icons, cursors or dialog boxes that haven't been moved into your application stack, the file "mctools.mc" must also be included in your distribution, or the "Resource Mover" dialog used to move the necessary components from the MetaCard tools stack into your application. Like the engines, MetaCard tools stack components can be distributed free of royalties.

Another advantage of not requiring a compiler to build your application is that you can even deploy your application on other platforms without having to build a separate binary for each. In fact, you don't even have to *have* a machine of the architecture you're deploying on. Just provide instructions about where to acquire the engine for the target platforms (all MetaCard binaries are available from the MetaCard FTP site ftp.metacard.com/MetaCard), or include the engine appropriate to the target platform in your distribution.

A MetaCard "top" program for Linux

A "top" program is a very useful thing to have on a multi-tasking operating system like Linux. You can use it to keep track of the CPU and memory usage of all programs running on the system, and to detect and kill runaway processes. But the character-based "top" program that comes with Linux systems could be improved upon. Because it isn't aware of the windowing system, it doesn't go to sleep when the window it is running in is iconified. It would also be nice if you could select a process to kill by clicking on it instead of having top type in its process id.

Fortunately, Linux has a very simple and elegant way to get process information, and so it's easy to develop a graphical application to display this information using a tool like MetaCard. This article will show you how it's done using MetaCard's scripting language MetaTalk. MetaTalk is a Very High Level Language (VHLL) that supports building complete applications with very little effort. The MetaTalk language has a very English-like syntax, but is also very concise. This makes it easy to learn, yet doesn't sacrifice the power and compactness found in other HVLLs like Perl. It is also very readable, and so it's easy for non-expert users to figure out what a script does (which can be much more difficult with languages like Perl).

The /proc pseudo-filesystem

Each process that is running on a Linux system has a directory in /proc. In that directory there are several files that contain information on the process. To implement a graphical "top" program, we're most interested in the information in the file "stat", which contains process run time and memory usage information. We'll also use the file named "cmdline" which contains the command line used to start up that process.

Each "stat" file contains a single line of information, with the fields separated by spaces. Details on the format can be found in the "proc" man page. It says that on a Linux 1.2 system, words 14 and 15 are the processes user and system running times, respectively. Word 23 is the process size (in bytes) and word 24 is the number of 4K pages currently in RAM for that process.

There are also files in the /proc system that contain information about the whole system. We'll use the "/proc/stat" file which contains overall system resource usage. We'll need that information to compute the percentage CPU usage for each process.

The procedure

Each time the display is to be updated, a program must do the following:

- 1) read the "stat" file in the /proc directory
- 2) find all of the subdirectories in the /proc directory
- 3) read the "stat" and "cmdline" files in each of these directories
- 4) compute the CPU time for the process by subtracting from the last time
- 5) save the current CPU usage
- 6) convert the CPU usage into a percentage of total usage

- 7) build the list of processes and display it
- 8) schedule a time to redo the update

The MetaTalk handler that does all of these things is called `updatelist` and is shown in Listing 1. This handler, like all MetaTalk message handlers, starts with the word `on` and the name of the message to be handled. The first few lines of this handler declare all the local variables used in this handler. While not strictly necessary, it's a good idea to declare variables to avoid bugs caused by misspelling variable names. To check your scripts, you can set the MetaCard property `explicitVariables`, which will flag as an error any variable used before it was declared.

The handler then gets the global system time statistics from the file "/proc/stat" and subtracts the values from the last time the handler was called. The time statistics are then stored in a local variable declared outside the handler, which works like a "static" variable in C: it retains its value like a global, but can only be referred to within the script (i.e., it doesn't pollute the global name space). This variable, like all MetaTalk variables, can be used as an associative array without special declarations. All you have to do is put a string (which can be a number) between the [] characters.

Note the expressions of the form `word x of y`. These are called "chunk" expressions and are a very powerful feature of MetaTalk. With them you can access elements of a string individually without having to split up the whole string into an array first. Also note that you can add words together without having to explicitly convert them into numbers first. This saves development time, and makes scripts smaller than they would be in lower-level languages.

The readfile function used in the updatelist handler is shown in Listing 2. Like all function handlers, it starts with the keyword `function` and is used in expressions just like MetaCard built-in functions.

There is one unusual thing about this function, though. Normally one would use `read from file x until eof` in MetaCard. This is because when you specify eof as the terminating condition, MetaCard speeds things up by getting the file size and reading the whole file with a single system call. But this doesn't work for /proc files since the stat() system call returns that they're all 0 bytes long, even though they contain data! So, we must force MetaCard to read the file a byte at a time by specifying `empty` as the terminating condition.

The readfile function also has to handle the case where the file specified isn't there (which can happen if the process exits after the "ls" command that gets the directory listing is executed). It also needs to convert nulls (ASCII 0 characters) in the strings to spaces, since some of the files use this character as a delimiter. Note that this requires a scripting language that can handle binary data. MetaTalk has no problem with this, but many other scripting languages would.

The rest of the updatelist handler does steps 2 through 7 in the above recipe. The resulting listing is sorted twice to remedy one of the more annoying characteristics of the character-based top program: the individual processes bounce around the listing unpredictably if they're not using any CPU time. Instead, we'll sort them by process size and then by CPU time which is a more useful way to do it. This is only possible because MetaCard's sort is stable, which means order is preserved on sequential sorts if elements have the same key value.

Finally the `updatelist` handler uses the `send` command to schedule a call back to itself in a few seconds and stores the id of the timer `send` creates in a variable. This local variable can be used to cancel the timer using the `cancel` function. For example, when the application is iconified, we want to stop the processing (See Listing 3).

I snuck in a little bit of object-oriented programming in the `send` command. The message is sent to `me`, which is the object whose script is currently executing. The time interval is specified as `the updateinterval of me`. This object has a custom property named `updateinterval` that is persistent, which means it's saved whenever the object is saved. That's why you don't see any initialization of this property in these listings: it's data stored with the stack, not code.

Building a stack

Backtracking a little, before you can write a script, you need to create an object to attach the script to. MetaCard applications are composed of one or more stacks, each of which has one or more cards. The metaphor (inherited from HyperCard) is of a stack of index cards, but you can also think of it as being pages in a book, slides in a presentation, or the frames in a movie or

video. The stacks are stored as binary files similar to the resource files used on the Mac and Windows. But in addition to object descriptions, data such as the state of buttons and the text in fields is also stored in these files.

The objects are created using MetaCard's IDE (Integrated Development Environment), which you use to create objects and to size and position them. To start building a new application window, choose "New Stack" from the File menu (the first card is created automatically with the stack). Then choose tools from the tool bar and draw the fields and buttons you need. See Figure 1 for a screen snapshot of the MetaCard "top" application.

After creating the stack and the controls in it, you add scripts to the stack, cards, and/or controls. Normally each object has its own script, but I've put all of the handlers for the "top" program in the card script. This means that the mouseUp handler (See Footnote) is a little unusual since it gets called through the message-passing hierarchy: messages not handled by an object can be handled by objects-higher up in the hierarchy. In this case, a handler in the card script handles messages sent to any of the controls on the card.

Writing one big handler instead of a bunch of small ones usually means that you have to figure out which object the message was originally sent to. The `target` function supplies this information. Like `me`, the `target` function returns an object that you can get the properties of or send messages to. This mouseUp handler also shows off the MetaCard `switch` control structure:

- The "Kill Process" case is executed when the user clicks on the "Kill Process" button. This section gets the PID of the process from the selected line in field and uses the MetaTalk `kill` command to kill it.
- The "Set Update Interval.." section prompts the user for the new updateinterval value using the `ask` dialog, verifies that the value is a number and tells the user that it must be if it isn't. The `ask` and `answer` dialogs are built-into MetaCard and are quick and easy ways to get simple responses from the user. If the new data checks out OK, the handler cancels the current timer and then calls the "updatelist" handler to restart it.
- The "toplist" case will be executed when the user clicks on a line in the main field. This case enables the "Kill Process" button and suppresses updates for 5 seconds, allowing the user time to kill the process before the selection is cleared when the field is next updated.

Listing 4 shows the handlers for the stack-oriented messages, including those send when the stack opens and closes, and is iconified and uniconfied (remember the goal of having the application go to sleep when it is iconified).

MetaCard doesn't have a constraint-based geometry system, so you must write scripts to handle resizing and repositioning controls when a stack window is resized. The resizeStack message handler that does this geometry management is shown in Listing 2. Using the IDE, I set the stack properties such that the stack is not resizable in width, only in height (since you need to be able to see all fields). So this handler only has to resize the main field vertically and reposition the buttons along the bottom edge of the display. This simple 4-statement handler reliably handles the task without triggering the time-consuming trial-and-error phase of development required to get a constraint-based system working correctly.

The complete stack is available for download as top.mc (note that you may have to shift-click or control-click to force the download to be saved into a file). It can be run using the free MetaCard Starter Kit available on the Get MetaCard page.

Performance characteristics

Running the MetaCard version of top takes about twice as much CPU time as the character-based version (6% vs. 3% on a Pentium 90 running Linux 1.2.13). This is a typical result, since a well written MetaCard script generally runs from 2 to 4 times slower than a comparable C program. Of course, the MetaCard version only took a fraction of time to develop. And because it is so much smaller, it will take far less time and effort to maintain and customize (the character-based top program is written in C and is about 10 times as long). Memory usage for the MetaCard version of "top" is considerably less than the total of the character-based "top" program and the "xterm" and extra "bash" process required to run it.

Conclusion

The real power of this graphical top is that it's easy to modify to suit your needs. You could easily add columns to display some of the other information shown in the character-based top, or

change the signal used to kill a process. Since MetaCard has built-in object graphics capabilities, you could even add a graphical display of the resources used by individual processes over time.

Interactive GUI Development Environments

A comparison of Tcl/Tk, the Desktop KornShell, and MetaCard

This series of articles compares and contrasts three popular interactive development environments for UNIX/X11 workstations: Tcl/Tk, the COSE Desktop KornShell, and MetaCard. The Perl language is also included in some comparisons.

Portions of these articles originally appeared in *The X Journal* which is published by SIGS Publications. Others were published as *Interactive GUI Development Environments* which originally appeared in the in *The X Resource*, Issue 11, published by O'Reilly & Associates, Inc.

Abstract

Developing graphical applications in C is a laborious process, and it is very difficult to generate high-quality applications when you have to spend much of your development time tracking down memory leaks and segmentation faults and waiting for compile/link cycles to finish. These three tools offer a way out of this predicament.

Each of these tools has an interpreted scripting language with powerful string-manipulation features and all make access to the components in the user interface simple and efficient. They promise to free developers from the drudgery of GUI application development, and to greatly increase their productivity.

Table of Contents

For more information on the strengths and weaknesses of Tcl with respect to other languages, see A comparison of Tcl with other systems.

For a special note to those considering Java for application development, see Java is taking us in the wrong direction.

Part 1: Overview of interactive GUI development environments

Once upon a time (1984), professional software engineers were the only people capable of developing graphical applications. The toolkits available at that time - the Macintosh toolbox, Digital Research's GEM, the interminably-in-beta-test Microsoft Windows 1.0, or the fledgling VAX-based X window system - were primitive and required mastery of a very complex GUI API. Worse still, applications usually had to be written in C, a language infamous for being difficult to learn and very unforgiving of programmer mistakes.

A few years later, some bright engineers at Apple Corporation decided that what "the computer for the rest of us" needed was a GUI development environment "for the rest of us". The result of this project was HyperCard, the first full-featured development environment designed for non-programmers (first released in 1989). Since then, many other end user programming tools have been developed including such HyperCard-like tools as SuperCard and ToolBook, and

independently evolved tools such as Visual Basic (VB) and Visual Basic for Applications (VBA).

Though tools like these were a long time coming for the Unix/X11 market, they're finally here. Products such as Tcl/Tk, the COSE Desktop KornShell (dtksh) and MetaCard make it possible for non-programmers to develop GUI applications without requiring the massive effort required to master C/C++ and the Xt/Motif toolkit.

But end users are not the only ones to benefit from the development of these tools. By offering a higher-level language for GUI development, these tools are much more productive development tools for *all* developers. In fact these tools are so much more productive than conventional tools, they will eventually replace conventional GUI development environments for nearly all application development.

Just as C compilers replaced assemblers as the standard software development tools in the early 80s, and languages like Perl and the shell languages have replaced C programming for most system administration tasks, it won't be long before developing an application in C or C++ with a low-level GUI API like Xt/Motif will be about as popular as programming in assembly language is today. This transition is already largely complete on the Microsoft Windows platform where the vast majority of applications are developed with tools like VB, PowerBuilder, Borland's Delphi and a wide variety of 4GLs and other high-level languages.

Why an interactive GUI development environment?

The primary advantage of a VHLL (also commonly called a scripting language) over a third generation language like C or C++ is that fewer lines of code must be written to complete a given task. Fewer statements take less time to write, and can be understood and modified more easily. The bottom line is increased developer productivity and better quality, higher functionality applications.

A second benefit arises from the fact that VHLLs are usually interpreted, which means you don't have to wait for compile-link-run cycles. This also greatly improves developer productivity since more time is spent developing and less time waiting. The productive time gained is actually much more than just the compile-link-run cycle time, since the developer doesn't repeatedly have to refocus attention on the task at hand, a process that can take as long as the compile-link-run delay itself.

While one might guess that choosing a high-level language might require sacrificing performance, benchmarking shows that in many cases screen update performance of applications developed with these tools is even *faster* than comparable applications developed in C with Xt/Motif due to the greater overhead of that toolkit (details to follow in the last article in this series). Furthermore, most VHLLs can be extended by adding commands written in C to improve performance where needed or to take advantage of libraries that are only available for C or C++. Certainly performance concerns aren't a reason to choose a third-generation environment over one based on a VHLL.

Scripting Language Overview

Though there is no universally accepted definition of a what a scripting language is, there are certain features that they usually have:

- Interpreted execution, so compile-link cycle is required. Execution errors cause the environment to point out the error to the developer allowing them to fix it and continue.
- A simple syntax. For example, statements are usually terminated by returns (new lines) rather than by semicolons or other punctuation. In most cases statements have a common "verb-arguments" format rather than the mixed order used in third generation languages.
- Untyped variables which act as strings or numbers depending on what operation is being performed on them.
- Variables are created when referenced, rather than through explicit declarations.
- High-level string manipulation features such as concatenation, substring operations and searching/sorting primitives are built in to the language.
- No pointers or memory allocation. "Garbage collection" of unused memory is done automatically, freeing the developer from having to deal with the most time-consuming and bug-prone aspect of programming in C.

Note that many of these features are also common to the fourth-generation languages (4GLs) used with many database front-end toolkits. The various shell languages and UNIX pattern-matching languages like awk and perl also have many of these characteristics. Even

some older languages like Lisp could be classified as scripting languages based on this criteria, though perhaps the simple syntax requirement can't be met. New-generation 3GLs like Java have some of these characteristics, but the complex syntax, requirement to compile as a separate step, and the need to type all variables means that Java should not be grouped with true scripting languages. See also Java is taking us in the wrong direction

Another useful distinction is that scripting languages are not the same as macro languages. Macro languages typically lack the control flow (repeat, if-then-else) and the procedure building capabilities that are required in a general purpose programming language.

Part 2: Origins and general description of the environments

Tcl/Tk, the Desktop KornShell (dtksh), and MetaCard resemble each other more than they resemble conventional programming environments based on C and a toolkit like Xt/Motif or the OI toolkit. Each tool has an interpreted scripting language, the ability to dynamically lay out a user interface, and the ability to extend the environment using C externals when required.

Due primarily to the greater power of the scripting languages, programs written with these tools are typically much shorter than equivalent programs writing in C or Pascal. This means that they take much less time to write and are easier to modify. Though class libraries and "component-ware" can be used to shrink the amount of code that needs to be written in a language like C++, programs written with these scripting languages will still be smaller in any non-trivial application.

Since the languages are interpreted, there is no compile-link-execute cycle and so development and debugging go much faster. Script execution errors are pointed out and can be fixed without exiting the environment. In most cases new code can be executed immediately. Though C-based environments can use tricks like precompiled headers and incremental linkers to reduce the compile-link cycle time, the fact that you have to restart your application from the beginning to test your new code remains a major disadvantage of developing graphical applications with compiled languages.

Another advantage of these tools is simplified access to the graphical controls. Instead of using complicated sequences of operations, you can get and set object properties with simple, single-line statements. The need to look up property names and the structures used to set properties is also greatly reduced, since the naming and argument passing conventions are simpler and more consistent between object types.

Their Rapid Application Development (RAD) capability make these tools especially useful for prototyping and developing applications that undergo frequent modifications. Combined with an Interactive Design Tool (IDT) for building the user interface, it becomes possible to have nonprogrammers prototype and customize applications, whereas they can only do the most rudimentary screen layout tasks with conventional Xt/Motif based GUI builders.

In many cases, even end-users can make simple modifications to applications, such as changing the layout or even adding buttons or fields, without requiring the assistance of a programmer. This is in striking contrast to most applications developed in Xt/Motif, where most end-users cannot even change the colors used in the application due to the overwhelming complexity of the X resource files used and the fact that the object-hierarchy is usually undocumented.

Though they are frequently dismissed as being "toy" development environments, Tcl/Tk, dtksh, and MetaCard are professional strength tools and are suitable for most, if not all, types of graphical application development. There are many commercial applications that are built in MetaCard and Tcl/Tk, and the number continues to grow as the tools become more sophisticated and as developers begin to accept them as viable replacements for conventional development environments. One big advantage of using these tools for commercial application development is that the pointer and memory leak problems that plague C and C++ programs cannot occur with these tools. As a result, applications built with them are more likely to be robust from their first release, whereas C applications are likely to need several patch releases to achieve the same level of reliability.

There are tradeoffs, however. Since the languages are interpreted, performance is always a concern. The languages also lack some of the features that make structuring large programs easier. These tools are also relatively new, and are still evolving to meet the needs of their users. This means that there are very few "gurus" who can teach new users how to use the tools and there is little third-party documentation and training available. Finally, at least for languages

other than MetaTalk, debuggers, profilers, and other aids that are standard in more conventional environments are missing or still at an early stage of development.

Although tools in this genre are relatively new to the X Window System, they are already well established on the Macintosh and PC platforms. For example, HyperCard was released on the Macintosh in 1987. Since then, hundreds of thousands of people have used it to build custom applications. There are dozens of commercial applications that use HyperCard as a database environment, user-interface toolkit or as a help system, including one of the most popular computer games of all time: Myst. Visual Basic for Microsoft Windows also has an installed base in the millions and continues to grow in popularity. Microsoft has is now including a subset of Visual Basic in all of its productivity applications, replacing their application-specific macro languages. The result will be that millions of people will be using a general-purpose application development system to customize the applications they use every day.

While it might be tempting to equate these interactive GUI tools with C and Xt/Motif based User Interface Management Systems (UIMSs) such as UIM/X or TeleUse, there are several important distinctions that should be made. First, the interpreted languages used in these systems (C in the case of UIM/X and the proprietary scripting language 'D' in TeleUse) are designed to supplement the third-generation language code used to build the bulk of the application code, not to replace it. These tools are also not complete environments since must use compilers and other tools to build your applications. This increases development costs and results in applications that are not as easily portable as those built in dtksh, MetaCard, or Tcl/Tk. Finally, these tools are targeted at professional software developers, so they are much more expensive and usually much more difficult to learn to use than the interactive GUI development environments described here.

Tcl/Tk

The Tool Command Language (Tcl, pronounced "tickle") was developed to be used as an embedded language. Application developers can use the language to provide customization and macro capabilities in their applications without having to develop a language themselves. Besides saving developers' time, it can also save end-users' time since they don't have to learn a new language to use a new application.

As a language targeted at non-professional programmers, Tcl was designed with a simple syntax. It follows the "verb-arguments" design shared by MetaCard and many other scripting languages, though it does have some shell-like syntax requirements. For example, literals must be enclosed in quotes and variable references must be preceded by a dollar sign ($).

Tcl's most rapid growth has come as a result of the release of Tk (tee-kay), a toolkit that supports construction of graphical user interfaces using the Tcl language. Unfortunately, many shortcuts were taken in the design of Tcl and Tk due to the limited resources of the original development team:

The basic idea for Tk arose in response to Apple's announcement of HyperCard in the fall of 1987. HyperCard generated tremendous excitement because of the power of the system and the way in which it allowed many different interactive elements to be scripted and work together. However, I was discouraged. The HyperCard system had obviously taken a large development effort, and it seemed unlikely to me that a small group such as a university research project could ever mount such a massive effort. This suggested that we would not be able to particpate in the development of new forms of interactive software in the future. -- John Ousterhout (architect of Tcl and Tk)

Built on a custom toolkit written in C, Tk is gradually evolving toward full platform style-guide compliance. Even so, it is already close enough to Motif look and feel in the current version (4.0) that many users won't notice the differences. The look and feel on Windows deviates from the standard style to a greater degree.

Tcl/Tk does not come with an IDT. Although there is a freely available IDT for Tcl/Tk called XF, it is not part of the Tcl/Tk distribution and it is not used by most developers, the majority of whom still construct their user interfaces using hand-coded scripting language statements. In addition, XF has very limited direct manipulation abilities; it lacks the ability to graphically lay out controls, for example. It is better described as an "indirect-manipulation" tool since most object selection and manipulation is done with list boxes that contain the names of the objects, rather than with the objects themselves.

Sun has released a GUI builder for Tcl/Tk called SpecTcl that is of much higher quality than XF, but it is a commercial product and hasn't achieved widespread adoption by Tcl/Tk development community. Like XF, SpecTcl is not a full IDE as it lacks things like script editing and debugging tools.

Tcl/Tk is a sourceware package written in the C language. Potential users must get the source code (the anonymous FTP site ftp.cs.berkeley.edu is one source) and compile it on their systems before they can begin using Tcl/Tk. The advantage of this distribution method is that Tcl/Tk is very easy to extend, both by adding new commands to the language and by adding new display widgets. The latter is not usually required however, since Tk includes support for bitmap and object graphics, and has a powerful text widget that supports multiple fonts and hypertext links.

dtksh

The Desktop KornShell (dtksh) is an evolutionary improvement over the Windowing Korn Shell (wksh) which was developed by UNIX System Laboratories. Though wksh was never actively marketed, it was distributed with some System V Release 4 systems, such as UnixWare. Unlike Tcl/Tk and MetaCard which have a Motif interface, wksh originally used an Open Look toolkit.

When the COSE group was defining its standard desktop environment, it was recognized that an easy way to create graphical applications would be an important component of the environment. Therefore, the CDE specification included a "Dialog and Scripting Language" component. Though it would have been possible to use an existing Motif GUI tool like MetaCard or Tcl/Tk, the developers given responsibility for this aspect of the project decided to evolve wksh to meet the requirements.

Rather than starting with a clean slate, as is the case with MetaCard and Tcl/Tk, wksh was developed using two existing technologies: the Korn shell and the Xt/Motif toolkit. The advantage of this approach is that the underlying toolkits are robust and well-known. They are also quite broad since they have been extended over the years to solve a wide variety of problems. The disadvantage is that neither was designed to be particularly easy to use, nor easy to learn. In fact, due to backward-compatibility requirements, the basic structure of these components have changed only slightly over the years.

Like Tcl/Tk, dtksh interfaces are constructed by executing scripts. In in most cases these scripts are (will be) built by hand, rather than with an IDT since an IDT is not built in. Though some of the IDT/UIMS vendors have hinted that their tools will be able to output dtksh scripts instead of the usual C and UIL code, the current complexity and high cost of these tools ($3000-12,000) puts them out of the reach of most potential users of dtksh.

Since it is based on Xt/Motif, dtksh lacks built-in support for object graphics, and has only rudimentary text formatting features. To some extent these limitations can be overcome by using dtksh's ability to draw directly to the display using the lowest level Xlib calls, but the complexity of implementing drawing or painting functionality at this level is beyond the ability of all but the most experienced developers.

Adoption of dtksh has been slow in coming, probably due to the slow rate of adoption of the COSE desktop by the various workstation vendors. But presumably as the COSE desktop becomes more widely established, so will dtksh. At present, the only way to acquire dtksh is to acquire the COSE desktop from your workstation vendor (usually as part of an OS upgrade), or a third party such as TriTeal.

MetaCard

MetaCard is a commercial tool designed to be compatible with Apple's HyperCard. It uses the same English-like scripting language and object property names used in HyperCard. It can even import and run HyperCard stacks. The MetaCard scripting language (MetaTalk) has many features designed to make it easy to learn and easy to use, such as the ability to deal with unquoted strings as literals and a lenient parser that allows noise words such as "the" to be freely used in commands without flagging them as syntax errors. It also has no unnecessary syntactical hurdles such as requiring variable references to be proceeded with a dollar sign ($) as is required with the other two languages.

The MetaCard user interface looks and works like OSF/Motif on X11 system and like Windows 95 on Windows 95 and NT systems, but it is actually built with a custom toolkit

written in C++. The user interface for MetaCard is itself written in MetaCard, so it can be customized by users to improve their productivity, or by developers to make it easier for end-users to customize their applications.

Unlike the other two environments, MetaCard has an IDT built in, making it by far the fastest tool with which to create user interfaces. The IDT was designed to be easy to use by non-programmers. In fact, it more closely resembles drawing program than it does most other IDTs. MetaCard's development environment also includes a graphical script editor and debugger, tools not included in the basic Tcl/Tk and dtksh packages.

Also unlike the other two environments, MetaCard applications are saved as binary files. As these binary files (called stacks) are read, they are converted directly into MetaCard objects. In contrast, with Tcl/Tk and dtksh an application's user interface stored as scripts. When an application is run, these scripts are interpreted to construct the interface. One disadvantage of this approach is that these script become quite long when construction large applications, making them more difficult to modify and maintain. It also makes producing GUI builders much more difficult, since it is necessary to intersperse calls to user functions with the code that generates the interface, making it difficult to regenerate the interface without overwriting the user functions.

Another disadvantage is that binary data such as bitmap graphics and audio clips must be stored in separate files from the scripts, whereas they can be stored directly in MetaCard stacks. Distribution of MetaCard applications is therefore much easier than for the other systems since in most cases only a single file needs to be moved.

The MetaCard binary format architecture also means that MetaCard objects are persistent. All of their properties, including the state of buttons and the text in fields is saved with the application. With the other two tools, if it is necessary to preserve the application state across uses of the application, scripts must be written to save this type of information to files and to read and restore the state from these files when the application is restarted.

MetaCard stacks can contain multiple cards, organized in a doubly-linked list. From a given card you can go to the previous card or the next card; stacks wrap around between the first and last cards. Since only the text in fields changes between the cards, the layout of the cards only has to be done once. This architecture can be used to easily construct address book-type applications, but can also be used to build documentation stacks where each card has a topic on it and the cards can be flipped like the pages in a book. For example, MetaCard's on-line documentation is constructed of MetaCard stacks. The user can use next and previous buttons to navigate through the topics covered in the stack. In addition, the MetaCard "find" command can be used to search all of the cards in the stack for a particular string. A third form of navigation is to use hypertext links. The user can click on a word and the card that has more information on that term will be opened.

MetaCard has built-in support for multimedia features such as video and audio playback. It also supports bitmap and vector graphics including the ability to edit both, making it a useful authoring and presentation tool in addition to a user interface development environment. MetaCard's text field control supports multiple fonts and sizes, alignment, subscript, searching and sorting, and hypertext links, making it easy to construct useful and aesthetically pleasing documents.

Part 3: Architecture of the languages

Variables

In all three languages, variables can be used without being declared. In Tcl and MetaTalk variables are local unless explicitly declared to be global, while the opposite is true for ksh. ksh is also the only dynamically scoped language: variables declared in a function have scope that includes all functions called by that function. This characteristic means that it is very important to declare all variables used in a function. For example, consider a function that uses the variable i (a common name for a looping variable) that calls another function that uses i as a looping variable. If i is not declared to be local in the called function, it will alter the calling functions i and a very difficult to find bug will be the result.

In MetaTalk variables are automatically interpreted as numeric values when arithmetic operations are performed on them. In ksh and Tcl, this conversion must be forced by using a special syntax (enclosing the expression in double parentheses in ksh and using the `expr`

command in Tcl). Tcl and ksh also require the variable name be preceded by the $ character when the variable is referenced, but *not* when it's set. Ksh also doesn't require the $ when a variable is referenced inside a double-parenthesis arithmetic evaluation.

MetaTalk (like Perl) uses double-precision floating point values for all internal calculations, whereas in ksh variables are treated as integers unless they are explicitly declared to be floating point (only single precision floating point variables are supported). Tcl takes a middle ground, treating values as 32-bit integers unless they are initialized with a number containing a decimal point, or an operation is performed on them that requires conversion to a double-precision real. This requires extra care in programming because integer overflow is not one of the conditions that cause conversion, and because there are some built-in functions that only accept integers. Ksh has a similar limitation when passing floating point values to functions: pass too large an integer, and the value will be truncated without warning.

All three languages have the ability to use a variable as an array. Arrays can be numerically-indexed as in third-generation languages, but strings can also be used as array indices. These "associative" arrays are a powerful feature that can in many cases make up for a language's lack of support for structures and classes. For example, suppose you needed a sorted list of the words used in a document, and the number of times each was used. Assuming the variable "somevariable" contains the text to be counted, this short MetaTalk script is all you need:

```
  repeat for each word w in somevariable
# automatically creates an array element
# and put a number into it
    add 1 to count[w]
  end repeat
# get a list of the array elements
  put keys(count) into sorted
  sort lines of sorted
  repeat for each word w in sorted
    write format("%20s %d\n", w, count[w]) to stdout
  end repeat
```

A C/C++/Java program to do this would be many, many times longer, and would take much, much longer to write. Even worse, it's quite likely that it would run slower than the MetaTalk script because in most cases it wouldn't be practical to optimize the associative array and sorting operations to the degree done to the MetaTalk engine.

As is required for numeric variables, arrays must by declared in ksh whereas any variable can be used as an array in Tcl and MetaTalk. In MetaTalk and ksh, accessing an array element that doesn't exist returns an empty string. In Tcl this causes an error, which means you must include extra commands to check for this case when using arrays as dictionaries.

All three languages have constructs that allows access to variables declared outside the current function. This feature can be used to implement call-by-reference. For example, in Tcl the name of an array can be passed in to a function and the `upvar` command executed to allow access to the elements of that array. The ksh language supports hierarchical variables for which a root name can be passed to a function and subvariables within it set individually. ksh also supports a `nameref` keyword that can make one variable a reference to another which is used like the Tcl `upvar` command. The MetaTalk language uses a more natural parameter list syntax to denote variables passed by reference (it works just like the C++ "reference" construct).

MetaTalk is the only language that supports the creation of static variables that have scope only within the script in which they are declared. Like global variables, these static variables retain their value across function calls, but unlike globals, the risk of name-space collisions with variables in other scripts is eliminated.

Tcl and ksh both allow variables to be unset, removing them from the environment. MetaTalk lacks this feature, and globals must be set to empty if their memory is to be reclaimed. Memory used by local variables, of course, is reclaimed as they go out of scope in all three languages.

All three support the full range of regular expression pattern matches available in languages like awk and Perl, although the syntax used by ksh is incompatible with the standard regular expression syntax used by vi and grep. All three also support the "globbing" type regular expressions used in the shell languages. MetaTalk also has validation operators that make it

easier to determine things like whether or not a string is an integer, real number, or x,y coordinate.

Control Statements, Commands, and Procedures

All three languages support the common control structures such as if-then-else, switch-case, and repeat loops. All three languages have special statements for catching errors and for sending messages to other objects.

All three tools can send events (call functions) after a certain period of time has elapsed, although with dtksh this is done through the GUI toolkit instead of with a built-in language element. This capability makes managing some time-sensitive UI techniques (e.g, leaving an alert up for a certain period of time) much easier.

Unlike Tk and Motif, all MetaCard objects can have custom properties, and special handlers can be written that are called when these properties are retrieved or set. Furthermore, like all MetaCard object properties, these custom properties are persistent across invocations of the application. Like member functions in object-oriented languages combined with the power of an object-oriented database, this is a powerful feature that makes building large applications easier.

All three languages have the most commonly used mathematical functions built in. MetaTalk also has advanced features such as a random number generator, and functions that find the min, max, and average of a list of numbers.

Ksh has a very minimal set of built-in commands, functions, and constants (42), many of which are only useful when ksh is used as an interactive shell. This requires that it rely heavily on running Unix commands such as `sort` as subprocesses for high-level data management, an architecture that seriously degrades performance. Ksh scripts use operators and other syntactic elements to a much greater degree than scripts in the other languages. For example, ksh supports several different types of pattern matching as operators, and these can be used in a wider variety of situations than is possible with the other languages. For GUI management, dtksh supports most of the huge number of calls in the Xt and Xm (Motif) libraries, and also many COSE Desktop specific calls.

Tcl has slightly more commands built-in (44) than ksh, including high-level commands for searching and sorting. Most of the GUI management is done using a separate set of approximately 130 Tk library calls. Many of these calls serve several different purposes which are specified by the type and number of arguments passed.

MetaTalk has by far the largest range of commands (96) and functions (154), including support for high-level data management, multimedia and animation, and support for the most common GUI techniques (e.g. password, alert/message, and file selection dialogs can all be displayed with built-in commands). The MetaCard GUI objects are managed primarily by setting the predefined object properties (354).

All three languages have the ability to read and write files, and to run and manage processes. Ksh is the strongest language in this area, since process management is such an import part of interactive shell use.

Tcl and MetaTalk have built-in support for accessing individual elements (words) in strings. MetaTalk's "chunk" expressions can be used to access individual characters, words, items, or lines in a text string. Tcl "lists" only support word-level boundaries by default, but lists can be nested to an unlimited depth, something that is not possible in MetaTalk. The list element delimiters that have to be inserted into the text string to implement this ability makes it impossible to use lists for general purpose text processing, however. For example, the "word count" procedure given in MetaTalk above can't be implemented in Tcl using lists, since any " or { characters in the input stream will cause the list operations to fail.

Tcl, like Perl, also requires that text strings that need to be accessed using something other than word boundaries (e.g., line boundaries or the components in a directory path) be "split" into chunks with an explicit operation, whereas MetaTalk can access the individual items in a string directly. Ksh has an "IFS" variable that can be used to change the word delimiter used when reading from a file, but the only way to to access words, items, or lines from a text string stored in a GUI input field or variable is to use complex set of substring and pattern matching operations, or to loop over all of the elements using the `for` construct, extracting the required element when it comes around.

All three languages support user-defined procedures. These procedures can have a variable number of arguments, which are easily referenced from within the procedure. While building procedures with variable numbers of arguments is also possible in most third generation languages, it is always much more difficult. Tcl and MetaTalk name the variables using an argument list in the function definition, whereas ksh functions only get positional parameters ($1, $2, etc.) which must be renamed in the function body if readability is a concern.

MetaTalk is the only language that makes a distinction between user-defined functions (which return values) and procedures (which don't). The other two languages have built-in functions for mathematical operations, but treat all user-defined functions as commands (procedures).

Tcl and ksh procedures can return values, but they must be called using a special syntax when used in argument expressions to other commands. Ksh functions can only return integers between 0 and 255, which means that most functions will have to use variable references (which changes the calling arguments) or command substitution (an alternate function call syntax which causes values printed to stdout to be redirected to a variable in the calling function). In MetaTalk, the same syntax is used to call both built-in functions and user-defined functions, and no syntactic gymnastics are required to return a value from a function.

MetaTalk is case-insensitive, although string comparisons can be made case sensitive by setting a property. The other two languages are case-sensitive. This can be problematic given the "camel capitalization" scheme used for property/resource naming used in all three GUI toolkits (the first letter of each word in a string is capitalized). This is especially likely to happen with dtksh since it uses the complex Xt/Motif naming scheme. For example, to add a callback for double clicks on a button the resource multiClick must be set. Write it as as the more natural multiclick and the interpreter will warn you that no such resource exists. To get around this, the examples in the standard reference book for dtksh, "Desktop KornShell Graphic Programming" by J. Stephen Pendergrast, Jr., all use the ksh `alias` command to rename all the Xt/Motif functions to all lower-case names.

All three languages use the '#' character to start a comment that extends to the end of the line, although ksh only recognizes the '#' character if it is preceded by a space and Tcl only accepts it if occurs where the interpreter expects to find a command. MetaTalk also accepts the HyperCard-standard sequence "--" to start a comment.

Debugging

Of the three, only MetaTalk comes with a GUI script debugger. Command-line script debuggers are available for Tcl and ksh, but these are not part of the standard development environments. Since all three are interpreted languages, however, most errors can be located easily by examining the line reported when the interpreter reports an execution error (although Tcl does not report the line number of an error, complicating the location process somewhat). In addition, there are many ways to output status information while the scripts are executing. This is similar to using printf in a C program, but more powerful since the messages can be sent to several different destinations. Tcl and ksh make this process even easier since they have commands that cause the environment to print out statements as they are executed.

MetaCard's script debugger and profiling tool can be used to set breakpoints and trace through a script statement by statement. In addition, variables can be examined and set while script execution is stopped. The MetaCard profiler can be used to find performance bottlenecks in scripts and also to provide script coverage measurements when doing QA on an application. Tcl has a "time" command which can be used to measure the performance of scripts, though no graphic display of this data is available in the standard environment.

Calling C Functions

All three tools can be extended by writing external functions in C or other languages. The architectures are very different, however. As a sourceware tool, extending Tcl/Tk is the most straight forward since you merely link your new functions into the Tcl interpreter. The full range of Tcl utility functions are available from your functions, as opposed to the limited access available in the other tools. The disadvantage of this approach is that the application must be exited and restarted every time the user-written C code is changed. Tcl also has a "send" command that can be used to send messages to another Tcl process through X events, and the ability to dynamically load libraries, both of which can be used to reduce the need to restart the application when developing external functions in C.

External C functions are added to dtksh by making them into dynamic libraries and loading these libraries with dtksh scripts; each external library and each function within each must be explicitly loaded by a separate statement. The external functions can call back into the dtksh interpreter to execute scripts and to get and set global variables. The disadvantage to this approach is that it is not portable to other operating systems, or even to all versions of UNIX.

Unlike the other two tools, standard MetaCard externals are run as separate processes. MetaCard's external procedure call mechanism relies on the X property and event mechanism for communication. Although calls using this path are much slower than the mechanism used in the other two tools, there are a couple of advantages. First, the MetaCard process does not need to be exited and restarted during the testing phase, which improves productivity. Secondly, the API is portable to platforms that don't support shared libraries such as DESQview/X and many older version of UNIX. External procedures can also call back into the MetaCard interpreter to get and set global variables and to execute MetaTalk scripts. An embedded version of MetaCard is also available, to which C functions are linked directly, similar to Tcl architecture.

Part 4: Architecture of the language/GUI interface

Managing the GUI

All three languages have built-in support for managing the GUI. In Tcl/tk you must specify the full path to a control. The Tcl syntax is fortunately very terse, however, and you can also put the path into a variable and use that to access the control if you'll be referring to it frequently. For example, to set the foreground color of a button:

```
.sample.manager.button configure -foreground blue
```

Unfortunately, not all options of the controls can be set with this mechanism. For example, to set the state of a Tk radio button widget, or to put text into an entry widget, you set a global variable that was specified when the widget was created. The advantage of this approach is that it saves typing since the full widget path doesn't need to be specified each time the value is changed. It also makes maintenance easier since when you rearrange the controls in an application you don't have to change every reference to that control in the scripts. The disadvantages are that it is an inconsistent architecture which makes it more difficult to learn and that it may require that large numbers of global variables be created which increases the chances of name-space collisions.

Some settings in Tcl/Tk do not have option names predefined, so you must use a built-in function to set them. For example, it is necessary to call the function tk_list-boxSingleSelect to make a list box object ensure that only a single line is selected at a time.

In dtksh, a variable called a widget handle is returned from each command that creates a widget. If not declared to be a local, this global variable can then be used to access the widget from elsewhere in the application. The widget can also be specified using a mechanism similar to that used in the X Resource system where the full path to the object can be specified. This addressing supports using wild cards, which isn't possible with the other two tools.

The command XtSetValues is used to set the resources of a dtksh widget:

```
XtSetValues $somewidget set:true
```

In MetaTalk, access to the controls from scripts is usually by forming a chunk expression to describe the object. For example, to set the highlight of a button (to show that it's been selected) in a window named sample, the following script would be executed:

```
set the hilite of button "some button"\
     of stack "sample" to true
```

If the current script is associated with a control in stack sample, the stack specification can be omitted, saving typing and making maintenance easier since the controls can be rearranged without having to change the scripts that refer to them. MetaCard objects can also be accessed by name or by ID, since each object is given a unique and permanent ID when it is created, something that is not possible with the other tools, since widget settings are not persistent across executions of the application. MetaCard, like Tcl, will also automatically recurse if a variable is used to refer to an object. For example, if the variable "it" contained the string "button 3", the following script will move the third button 8 pixels upwards:

```
set the top of it to the top of it - 8
```

Adding User-Interface Components

Adding additional display widgets to Tcl/Tk is relatively easy because the full source code of the other widgets is included, and because the new widgets can be linked directly into the Tcl/Tk executable. Tk doesn't use an object-oriented architecture however, so new widgets must be self contained.

It is also possible to add widgets to dtksh. Since it is built on the Xt toolkit, any of the public domain and commercial widgets developed for use with that toolkit can be used with dtksh. Writing your own widgets is not as straightforward, however. Since source code is not included you must first license the Motif source from OSF (at considerable expense). Though it is theoretically possible to create widgets without the source, it is generally regarded as being impractical to do this due to the complexity and inadequate documentation of the underlying structures. Xt is ostensibly an object-oriented architecture, but there are so many deviations from the tenants of that paradigm that it is actually more difficult to subclass Xt/Motif widgets than to write complete widgets for Tcl/Tk.

There is no way to add user-defined controls to MetaCard, though in many cases the ability to dynamically modify the graphic and bitmap objects from the scripting language can be used to achieve the same results. There is also an "imagePixmap" property that can be used to draw any type of graphics into an image in real time from a C/C++ function, a feature that can also be used to develop custom controls.

Part 5: Sample Application (font chooser)

Application Design

In this section we describe the implementation of a simple, yet useful, application in each of the environments. Considerable effort has been made to avoid choosing an application that overly favored one environment over another. This is especially difficult in the case of MetaCard, since the IDT makes construction of layout-intensive applications much easier. MetaCard also has a database built-in in the form of the multiple card architecture, so applications such as calendars, note-pads and address books are trivial to develop. These types of applications would be much more time consuming to produce in the other environments.

Due to development time restrictions, large scale applications and those that require extensive C coding could not be considered. Since this is the strong point of Tcl/Tk, the sample application doesn't demonstrate one of the best aspects of this tool.

Since dtksh lacks support for color bitmap and object graphics, and since the Xt/Motif text widget lacks support for many of the formatting and hypermedia features present in MetaCard and Tcl/Tk, we had to rule out all applications that require graphical displays of data and interactive multimedia presentations.

One sample application that meets all of these restrictions is a font chooser dialog, similar to that found in a word processing application such as Microsoft Word. The dialog allows the user to select a font, a point size, and a style. A preview area shows the font defined so far. When the OK button is pressed, the X Logical Font Description (XLFD) name used to load the font is output to the dialog's standard output. The MetaCard version of this dialog is shown in Figure 1. The applications developed in the other two tools look very similar and so aren't shown.

Figure 1: MetaCard Version of the Font Chooser Dialog

The scripts used for the different implementations of the font chooser dialog are shown in the Font Chooser Example Scripts: Tcl/Tk, dtksh, and MetaTalk.

A similar comparison of a directory browser application written Tcl/Tk with the same application development in MetaCard is available as MetaCard and Tcl Directory Browers

With all three tools, the first step in building an application is to construct the user interface. In most cases, several different possible layouts of the controls are tried and the one with the best look and usability is selected. Next, scripts are added to handle each of the

controls. This is also a dynamic process, where short sections of the scripts are written and then tested immediately. Finally the interactions between the controls are tested to ensure that they function properly.

User-Interface Components

All three tools support the full range of user interface components: push buttons, radio buttons and check boxes, labels, editable text fields, list boxes, menus and dialog boxes.

Tk has a wide range of special-purpose controls and has a simple and consistent naming scheme for resources and callbacks. Tk lacks built-in support for option menus, and lacks a "default" style for buttons (the double border around the button that is activated when the user presses the return key). Keyboard traversal to buttons and scrollbars also isn't supported in the current version, though this is scheduled be added to the future Motif-compliant release of Tk.

dtksh supports the widest range of controls and of optional behaviors for the controls of the three tools. Although this makes it the most flexible tool, it also makes it the most difficult to master. The problem is exacerbated by the common use of different terms for similar features in Xt/Motif. For example, the text in a label (non-editable) widget is called the labelString, while the text in an editable text widget is called the value. The callback for selecting a push button is called the activateCallback, while for toggle buttons it's called the valueChanged callback.

Rather than the wide range of widgets available in the Motif toolkit, MetaCard is built on a few control types, each of which can be configured to look and behave in a wide range of ways. For example, the button control can be a push button, a check box or radio button, or a menu button depending on how its properties are set. The field control can be either editable or non-editable, or even a list box. Like Tk, MetaCard supports object graphics as a control type, which makes developing applications that require odd-shaped buttons easier than with the other tools.

Layout

The starting point for building the font chooser application in MetaCard is to lay out the dialog using the IDT. Controls are constructed by choosing the appropriate tool and dragging out the size and shape required. Double clicking on an object brings up a properties dialog that allows you to set the name and other attributes of the object. Alignment of the controls is done by selecting multiple controls and using one of the resizing/alignment routines available in the layout dialog box.

In contrast, layout of the same dialog in dtksh or Tcl/Tk is a much more tedious process. Each time you make a modification, the dialog must be destroyed and the script that constructs it edited and rerun. Since you can't tell what the interface will look like from examining the script that creates it, it often takes several tries to get a control positioned and sized correctly. The vendors of IDTs for the Xt/Motif toolkit claim improvements in productivity from 2 to 10 times over doing layout without an IDT. Since development of the user interface typically takes from 25 to 50% of the total time required to develop an application, high productivity in this area is an important advantage.

A related concern is the ultimate quality of the interface produced. With an IDT, it is easier for the developer to take the time to make sure that the interface is logically laid out and aesthetically pleasing. Without one, the layout process is often terminated as soon as the minimum functionality is achieved because it is too time consuming to tweak the interface in an effort to optimize ease-of-use or aesthetics.

Even without the need to try different arrangements of controls, which I had already done in MetaCard, layout of the font chooser dialog with the other two environments took experienced developers at least 4 times longer than the layout time in MetaCard.

Geometry Management

Both dtksh and Tcl/Tk have manager widgets that resize and reposition their children when they are resized. The layout rules are specified as attachments and constraints. For example, a button might be attached to the left edge of the manager, and set to be 20% of the width of the manager. The advantage of this approach is that the layout and resize process is automatic; the developer does not need to write any code to manage the process.

The major drawback to this approach is that in many cases it is difficult or even impossible to specify the constraints properly. Those this is also a problem with the Tcl/Tk geometry

managers, it is more serious with the Xt/Motif-based dtksh:

The creation of a visually appealing and usable Motif user interface will often require an application designer to lay out control widgets in a way that is not easily achieved using the standard Motif manager widgets.

In such cases the ideal solution should be to implement a custom widget that provides the appropriate control layout. Unfortunately, the complexity of writing such a widget often outweighs the benefits from doing so, and the developer will end up trying to trick the existing Motif widgets into giving an acceptable layout.

-- Alastair Gourlay, "The One-Minute Manager" X Resource, Issue 10, p 63.

Using manager widgets to enforce a desired resize behavior is not easy for new Motif programmers. Actually, it's not very easy for experienced Motif programmers either.

-- J. Stephan Pendergrast, Jr., "Desktop KornShell Graphical Programming", p 170.

Dialogs that degenerate into useless (and ugly) configurations are all too common in Xt/Motif applications. The constraint managers used in this architecture can also override the user's preferences. For example, the Xt/Motif file selection dialog has the annoying habit of resizing itself to a fixed size every time a directory is changed, despite a user's best efforts to keep it large so that more files can be seen at one time.

Since most dialogs don't need to be resizable, many application developers just spare themselves the trouble of fighting with the geometry management issues and make the dialogs non-resizable. This not much of a sacrifice: resizable dialog boxes are extremely rare in Macintosh and Microsoft Windows applications. Another example is that most of the dialogs used in HP's Vue desktop environment (the basis of the COSE desktop) aren't resizable..

MetaCard lacks a constraint-based geometry management system. Instead, the developer must write a script to resize and reposition the controls when the resizeStack message is sent. Like the scripts used to construct the interfaces in dtksh and Tcl/Tk, these resize scripts can be tedious to write since every control must be specified separately. However, unlike the with constraint-based geometry managers, with this algorithmic method it is always straightforward to get the look you want.

Dialogs such as menus are very easy to build with constraint-based geometry managers since all of the buttons are the same size, and are arranged such that they are touching each other. The font chooser dialog, on the other hand, was particularly difficult to lay out with geometry managers since it is very irregular. Since the dialog was initially laid out so that everything could be seen, it was not necessary to make this dialog resizable. In order to ensure that the interfaces looked the same for all three applications and to reduce the length of the dtksh and Tcl/Tk scripts (since manager widgets did not need to be created and offsets and attachments specified), a short MetaCard script was used to dump out the coordinates of the controls I had laid out in a form usable by dtksh. A similar script was used for layout of the Tcl/Tk version of the application

Resources/Options/Properties

In dtksh, as with most applications based on Xt/Motif, resources are used to specify things like the fonts and colors used in an application. With Tk the term "option" is used instead of the term resource. In both of these environments, resources can be set either directly in the scripting language or through the use of the X resource system (e.g, with the .Xdefaults file).

Unfortunately, X resource files can be very confusing to non-programmers. It is very easy to accidentally alter a resource file such that an application looks bad and is difficult or impossible to use. For this reason, most commercial applications allow only limited (if any) access to this method of changing an application's look-and-feel. And since new resource values do not take effect until the application is restarted, they are not a very useful tool when prototyping an application with dtksh or Tcl/Tk. Indeed, though Tcl/Tk supports X resources, the documentation explicitly recommends against using them.

In MetaCard, the term "properties" is used to refer to these settings, which can only be set from the scripting language. During construction of the interface in MetaCard, properties are set

using properties dialog boxes. These properties dialogs and menus are just MetaCard stacks and have no special relationship with the objects whose properties are being set. They just execute MetaCard scripts to change the properties. For example, to set the text displayed in the Font Name button, the following script is used:

```
set the label of button "Font Name" to "Helvetica"
```

Event Handling

Nearly all GUI toolkits have an event driven architecture. Each time the user of the application does something, whether it is typing on the keyboard, moving the mouse, or clicking with one of the mouse buttons, events are sent to the application.

In Tcl/Tk and dtksh, the events are handled by callbacks. The callbacks are functions in the scripting language that are attached to

the controls by executing a statement that binds the function to a particular event type. For example, the following dtksh statement binds the function selectCB to the event that occurs when a push button is pressed:

```
XtAddCallback $widget activateCallback "selectCB $widget"
```

The dtksh function that brings up a message box telling users what they clicked on can be declared as:

```
function selectCB {
   XtGetValues $1 labelString:name
   XtSetValues $MB messageString:"You clicked on $name"
   XtManageChildren $MB
}
```

In this example, $1 is the widget handle which passed in as the first argument due to the way the callback was added, and "Click Button Example" is the string put into the title bar of the dialog box. Note that you must get the name of the button in a separate statement, since the XtGetValues call puts the value into a variable ($name) rather than returning it. The MessageBox $MB must be created in the application initialization code.

In Tcl/Tk, two different methods of binding functions to events must be used. The most commonly used events have predefined options. For example, in the sample application the command option is used to specify a function that should be called when the style check boxes are clicked on. Less commonly used events must be bound to functions using the "bind" command. For example, there is no predefined option for catching a single click on a list box and so in the sample application the raw ButtonRelease-1 event on the list box must be bound to the resetFont function.

In MetaTalk, the MetaCard scripting language, events are called messages. When a message is sent, the script of the target object is searched for a handler for that message. For example, the following handler would be executed if it was in the script of a button and that button was clicked on. It opens a dialog box that displays the name of the object that was clicked on:

```
on mouseUp
  answer "You clicked on" && the name of the target
end mouseUp
```

When a message isn't handled, it is passed up the object hierarchy. For example, a message might be passed from a button to a group, which is MetaCard's container object. If it's not handled by the group, the message is passed on to the card and then on to the stack. The stack is often used to handle events that can be sent to any of the controls or cards in the stack. In the sample font chooser application, a single mouseUp message handler in the stack script handles the messages sent to all of the controls in the application.

Part 6: Performance Benchmarks

Some simple measures of performance were taken for each of the languages. The scripts perform the same operations, but were written using the most common idioms for each language. See and compare the scripts used for each language: Tcl, Ksh, MetaTalk, and Perl.

Note that these scripts have been converted to HTML to be readable in WWW browsers. If you need the scripts in ready-to-run form, get ftp://ftp.metacard.com/MetaCard/contrib/bench.tar.gz.

Test	# of loops	Tcl	Ksh	MetaTalk	Perl5
Simple loop	10000000	1111	103	4	5
Iterative factorial	10000	60	28	9	8
Iterative factorial with 'if'	10000	45	17	4	10	1
Recursive factorial	10000	03	921	19	53	6
String operations (stems)	10	02	21	22	26	1
File write and read	2000	3	604	10	52	2
Run Subprocess	1000	91	23	24	2
Set widget state	5000	02	32	56	10	3
Set with locked screen	5000	1	5	2	1
Start up to simple window	1	1	1	1	2
Start up to complex window	1	3	5	2	6

The test machine was a 90 MHz Pentium with 32 MB RAM running UnixWare 2.0. Versions used were Tcl 7.4 pl3 with Tk 4.0 pl3, dtksh for UnixWare 2.0 (from ftp.novell.de, ksh version M-12/28/93, dtksh version 1 August 1995), MetaCard 2.0, and Perl 5.002 with Tk-b11.02 (based on Tk 4.0 pl3). All times are in seconds. Script execution times were taken using each language's built-in time functions, but were verified using the UNIX time command. Times were averaged over at least two runs.

The first test measured the performance of a simple loop operation, adding 1 to a numeric variable within the loop. The second test adds a multiply for computing factorials. The third added an if-then statement to the factorial loop. The fourth called a factorial subroutine recursively instead of using a repeat loop.

Two things stand out in these numeric-operation benchmarks: Tcl has relatively poor performance in numeric operations, and dtksh has huge overhead when calling subroutines that return values (due to the fact that ksh procedures can only return values between 0 and 255 without forcing a special redirection-of-output operation).

The string operation benchmark extracts words from the Linux dictionary file /usr/dict/words (350K/38470 words) that meet a certain criteria: they must be 5 letter words that are not plurals, don't contain apostrophes, and at least one other 5 letter non-plural must share the same first 3 letters. While this may seem like an unusual task, it is a real-world example of a type of task that all of these languages were designed to solve easily. In this case the words were used as stimuli in a psychological experiment that measured performance in a word-completion task (the subject saw three letters followed by two underscores and asked to report the first 5-letter word they thought of).

The ksh implementation of this benchmark is not directly comparable to the other two since ksh lacks the important "sort" operator needed to sort the associative array keys. The scripts should also have converted the words to lower-case, but this was also omitted from this

benchmark: Tcl and MetaTalk have operators for performing this conversion built in, but ksh does not and the steps required to implement it would drastically reduce its already poor performance.

As in the factorial benchmarks, there is a striking difference between performance of the two languages that use conventional interpreters (Tcl and Ksh) and the two that use "virtual compiler" technology (MetaTalk and Perl), the latter being at least 9 times as fast as the former.

The next two benchmarks show that there isn't as much difference between the languages for file and process intensive operations, presumably because the OS and hardware are the limiting factors here.

The last four tests measure the GUI performance of the tools. To improve measurement of the start-up tests, the GUI measurements were made running the applications across an unloaded network to an HP 9000/700 display. This additional overhead slowed down start-up times allowing more precise stop-watch measurements.

The update test measures performance getting a value out of a widget, incrementing it, and then putting it back. All three tools allow suppression of updates, and so performance was also measured when suppressing updates (the comments in the scripts show how this was done). Even without this using feature it is clear that the tools have plenty of display updating performance. Given the performance bottlenecks of malloc and the client-server communication path, writing an application in C probably offers little performance advantage in display update intensive applications. Indeed, the much shorter times for Tcl/Tk and MetaCard relative to dtksh on the update test compared with the first two tests is an indication that update-intensive applications written in C with Xt/Motif are likely to be even slower than if they had been written in Tcl/Tk or MetaCard.

The "start up to simple window" benchmark shows that starting up an application with only a small number of controls is very fast with all three tools. On the other hand, adding 256 buttons to the window as was done for the "startup to complex window" benchmark shows that there is considerably more overhead for managing widgets in the Xt/Motif-based Desktop KornShell than in the other two tools. The Perl/Tk benchmark shows that there is considerable overhead in the "glue" routines required to connect Perl to the Tk library.

Benchmark conclusion

The general conclusion that can be drawn is that any of the languages should be plenty fast enough to keep up with a human being. MetaCard's considerable performance advantage over the other two make it the best tool to use for developing larger applications, especially for applications that do a lot of numeric processing.

Another characteristic of the languages that isn't reflected in the benchmarks was the relative stability of the tools. While many programming errors were made developing these tests in all three languages, MetaCard and Tcl never crashed during development. dtksh, on the other hand dumped core dozens of times during development of these simple tests. Strange as it may sound, the most "mature" of these tools was also the least reliable.

The performance differences between Perl5 and MetaCard are significant for some benchmarks, but not for others. The raw-statement execution speeds of the two languages are almost identical, but Perl has several "shortcuts" that allow faster execution of some operations. For example, requiring different operators for comparing numbers and strings gives Perl a considerable boost in the factorial with 'if' and string operations benchmarks. MetaTalk lacks this requirement and so has to check to see if the operands are, or can be converted to, numbers in each comparison. This has some performance penalty, but simplifies the process of writing scripts.

In general, the MetaTalk language design philosophy has been to simplify syntax whenever possible, even if it means minor costs in performance. For example, of the 17 "Common Goofs for Novices" listed in "Programming Perl" (Larry Wall and Randal L. Shwartz, 1990, O'Reilly & Associates, p. 361-364), 16 can't occur with MetaTalk because the syntax is so much simpler. Particularly noteworthy is goof #3 "Using == instead of eq and != instead of ne" for comparisons, which is the aforementioned performance-enhancing Perl feature. Even so, unlike the difference between the performance of MetaTalk vs ksh or Tcl, MetaTalk performance is not qualitatively different from that of Perl5.

Some of benchmarks for Tcl have improved with the release of the new byte-code compiler for Tcl 8, but preliminary results indicate that the speedup is far short of what is needed to bring

Tcl up to parity with MetaTalk and Perl. The primary reason for this lack of improvement is that the design of the Tcl language is such that it is often impossible to preprocess arguments to function calls as is required to precompile them. Consider the following example:

```
set x 7; set y 9
puts stdout "$x + $y = [expr $x + $y]"
=> 7 + 9 = 16
```

The problem here is that the [] around the expr function is inside quotes, which means it cannot be evaluated until *after* the `puts` command is executed, which means it can't be precompiled into byte code. This technique is commonly used by Tcl developers, and there is no built-in support in the language for distinguishing what can be compiled and what can't be. Developers will have to spend considerable time developing the skill to determine where execution occurs, and therefore whether or not optimization can be performed. In this sense, MetaTalk represents a difference in capability over Tcl that is comparable to that made by structured languages like Pascal over GOTO-dependent languages like BASIC: by separating argument processing from string handling, MetaTalk provides language-level support for ensuring high-performance applications.

Part 7: Application deployment, documentation, and support

All three tools have one tremendous advantage over conventional development environments: applications are truly portable. No compilation or other preprocessing is necessary. You just move the source code (or in the case of MetaCard, the binary stack) over to the target system and run it. Of course, the interpreter must have been built and installed before this will work.

With dtksh, presumably this engine will be installed as part of the normal operating system installation. Unfortunately, this portability only extends to systems that will be upgraded to OS releases that include the COSE desktop environment. Many systems, probably including the majority of SPARC systems running SunOS 4.1.X, won't be upgraded to run CDE and so won't be able to use dtksh. And since the dtksh engine can't be redistributed free of charge as is the case with Tcl and MetaCard engines, it is not possible to include the dtksh engines in distributed products just in case the customer does not already have it. This means that it is impractical to build many kinds of commercial products with dtksh, since the potential market is so severely limited.

The MetaCard engines are prebuilt for all supported platforms. After this executable is placed in the user's PATH, MetaCard stacks can be run directly from the command line or by double clicking on them with a file manager. Deployment of Tcl/Tk applications is similar, though in some cases the binaries will need to be compiled for each target platform.

Of the three environments only MetaCard has encryption built in to password protect stacks. This protection makes it possible to distribute MetaCard applications without the concern that an end user can modify a script or copy some or all of it for their own use.

A final potential advantage of Tcl/Tk and MetaCard over dtksh is portability to non-UNIX platforms. Tcl/Tk is available on the Macintosh and MS-Windows, though it doesn't have native look and feel on these systems. MetaCard is available on Windows 95/NT and has native look and feel on all platforms. Products that are source-compatible with MetaCard have been available on the Mac and Windows for years, including HyperCard, SuperCard, Oracle Media Objects, and Object Plus.

It is very unlikely that dtksh will ever be available on non-UNIX platforms. Although the Tcl/Tk and MetaCard executables are larger (900K and 1.3M respectively), they are self-contained. dtksh, on the other hand, is only 800K, but requires many additional dynamic libraries totaling over 2MB. Porting the required Xt and Motif libraries to other OSs would be very difficult, and their large size would make dtksh non-competitive in these environments.

Documentation and support

Tcl/Tk comes with a man page describing basic functionality, but most developers will want to purchase the book published by Addison Wesley "Tcl and the Tk Toolkit" (ISBN 0-201-63337-X). The book is comprehensive and indexed, but lacks important reference information such as tables listing all of the widget properties and the built-in Tcl commands. Instead, this information is available in an on-line form, presumably to reduce the degree to which the book

becomes obsolete as the toolkit evolves. Technical support for resolving problems with Tcl/Tk is generally acquired through the comp.lang.tcl newsgroup or from an experienced friend or colleague.

The standard of dtksh documentation is J. Stephen Pendergrast's "Desktop KornShell Graphical Programming". Technical support for dtksh will come from the vendors, which will vary greatly depending on the vendor and on the support plan purchased with the OS.

MetaCard comes with extensive on-line documentation in the form of MetaCard stacks. Though no index is provided, the documentation can be search with a full-text search, and extensive hypertext links are included. Hardcopy documentation is available at extra cost. In addition, since MetaTalk is compatible with HyperTalk, any of the dozen or so books on HyperCard can be used as scripting language reference material. MetaCard technical support is available free of charge via email from MetaCard Corporation.

Part 8: Best use of the tools and conclusion

Tcl/Tk shines in longer term projects, especially when it's embedding capability can be used. Projects that require specialized commands and widgets would be easiest to produce in Tcl/Tk as will those that require manipulating complex data structures.

The fact that Tcl/Tk is free also makes it an attractive package for universities and other budget conscious operations. The extensive range of extensions and specialized tools, especially those that allow connection to relational database systems, may make producing some applications quicker than is possible in dtksh or MetaCard where the database connectivity layer may need to be written in C from scratch.

The best application of dtksh is for one-time projects, especially if the developer already knows the ksh language and the Xt toolkit. It also is a viable option when a Motif widget that meets the current needs of the developer is already available either free of charge or from one of the vendors of add-on widgets. It should also be free with most UNIX operating system upgrades, which will be an advantage for budget conscious organizations and for users that only occasionally use the tool.

On the other hand, dtksh is by far the most difficult tool to learn and use. It is also the least efficient both in terms of performance and lines of code that must be written to solve a given problem. The weak text widget and lack of support for for vector and bitmap graphic objects also means that in most cases it will take longer to develop an application in dtksh than with the other tools. Due to the design of ksh (no true functions or local variables, and great dependency on global variables), large dtksh applications will be much more difficult to maintain than large applications written with either of the other tools. Finally, the bleak prospect for portability means that dtksh is not the tool to use to develop applications that need to run on non-UNIX operating systems.

MetaCard's IDT puts the other two tools at a severe disadvantage for all but the most trivial layout tasks. For most small applications, the ability to use an IDT far outweighs any of the other features the other tools provide. MetaCard is also the easist tool for non-programmers to use to aid programmers in the design of an application, since they don't have to learn the scripting language to begin prototyping. The persistence of MetaCard applications and the built-in database also make it easier and faster to develop applications that require maintaining information across invocations of the application.

In addition to its overwhelming performance advantage over the other two, the MetaTalk scripting language is also the easiest to learn and to use, since it is syntactically much simpler than the other two languages. It is also relatively free of the dependence on global variables for basic operations that plague the other two languages. The GUI script debugger is also always available to handle those rare cases a scripts behavior can't be understood by simply reading it.

MetaTalk also supports a much wider range of built-in commands and functions, such as those required for multimedia applications, than the other two tools. Since MetaCard doesn't require construction of the interface in the scripting languages, the scripts for MetaCard applications are typically much shorter than for the other two tools, and so are easier to maintain.

Conclusion

Tcl/Tk, dtksh and MetaCard all offer a vast improvement in GUI development productivity

over toolkits based on C or C++. They should be considered as a first choice for most application development. Shorter scripts, which take less time to write and less effort to maintain than code written in a third generation language is only one of the benefits.

Perhaps even more important are the improvements in the quality of applications developed with these tools. First of all, fewer statements necessarily means fewer bugs. In addition, the pointer and memory leak bugs that are so common in applications developed with C and C++ environments are eliminated by design with these tools. Finally, the fact that applications can be developed faster and modified more easily makes it more likely that an application will be tuned to suit the needs of end users.

As these tools are improved to make application development even easier, it seems likely that they will gradually replace C and C++ based development environments for almost all graphical application development.

Part 9: Acknowledgements and references

I would like to thank the reviewers of my early drafts of this article for their comments and corrections: Sven Delmas, Phil Gardner, Earl Lowe, Tomas Lund, John Ousterhout, Steve Pendergrast, and Brian Raney.

References

COSE Desktop KornShell

1. Pendergrast, J. Stephen, Jr. , Graphical Programming with the CDE Desktop KornShell, The X Journal, March-April 1994, pages 39-52.

2. Pendergrast, J. Stephen, Jr. , Desktop KornShell Graphical Programming 1995. Addison Wesley.

3. Yager, Tom, Hidden Treasure, PC-Unix Connection, Open Computing March 1994 pages 141-145. (actually on wksh).

4. Rosenberg, Barry, Korn Shell Programming Tutorial, Addison Wesley

5. Young, Douglas, OSF/Motif Reference Guide, Prentice Hall

Tcl/Tk

6. Morin, Richard, Tcl, Tk, and friends, UNIX REVIEW, V11 N4, April 1993, pages 93-94.

7. Ousterhout, John K. and Rowe, Lawrence A., HYPERTOOLS A revolution in GUI applications" (listed in the TOC as "Hypertools: A GUI revolution"), The X Journal, March-April 1993, pages 74-81, "

8. Ousterhout, John K., Hypergraphics and Hypertext in Tk, The X Resource, Issue 5, Winter 1993, pages 113-128.

9. Ousterhout, John K., Tcl and the Tk Toolkit, Addison Wesley, 1994.

10. Richard, Kevin and Johnson, Eric F. Tickled Pink (Cross Thoughts), UNIX Review, March 1994, V12 N3, p87-90.

MetaCard

11. The Best Products of 1992, UNIXWorld Magazine, January 1993 pages 46-55.

12. Fickes, David, MetaCard: Useful but tough, Sun World, January 1993, pages 55-57

13. Burgard, Mike, Programming for the Masses, UNIXWorld, May 1993, pages 109-112.

14. Southerton, Alan, Beyond the Desktop, Modern Unix, Wiley, 1993 pages 204-208.

15. Deignan, Michael, Multifaceted MetaCard, SCO World, January 1996.

16. Raney, Scott, The Scripting Revolution, The X Journal, November-December 1995.

17. Goodman, Danny, The Complete HyperCard 2.0 Handbook, Bantam, 1990

18. Winkler, Dan and Kamins, Scot, HyperTalk 2.0: The Book, Bantam, 1990.

Perl

1. Wall, Larry and Schwartz, Randal L, Programming Perl, O'Reilly & Associates, 1990.

The Case Against Strained Carrots

Suppose one day you go to the store to buy some carrots. But when you get there, you discover that the only carrots your grocery store has are the strained carrots in the baby-food section. Looks like you won't be making the cole slaw, beef stew, or moo-shu pork you had planned.

A similar scenario is unfolding on the World Wide Web: the utility of the information available is being severely limited because organizations are over processing it. Whether they're using custom Java applets, JavaScript, HTML forms and CGI scripts, or content that requires plug-ins to view, the net result is that *they* are controlling how information is queried, transmitted and presented, not the information consumer. This is analogous to grocery stores only carrying strained carrots.

The analogy here is deep: not only does use of these technologies put severe limits on how you can use the information and on how easy it is to find, but it also unnecessarily discriminates against smaller organizations because only the largest organizations can afford to publish their information this way. In the real baby food industry, only the largest companies can afford the technology required to produce, package, and distribute strained carrots. Likewise, no Internet equivalent of a farmers market or roadside stand can afford the custom development required to make "strained carrots" of their information to make it accessible to Internet users.

Here are a few examples of problems that the Internet would be an excellent vehicle for solving, most of which are solved now by spending lots of time on the telephone. Unfortunately, it's beginning to look like the Internet is not going to be able to support any better solutions to these problems than using phone any time soon because the rapid adoption of HTML forms, Java, and other processing technologies are going to delay the implementation of the necessary infrastructure.

1) You need to ship a crate across the country. Who can do it inexpensively in a reasonable amount of time?

Most of the large shipping companies are hard at work publishing their rates in HTML format and building Java applets and HTML forms with CGI back ends to present the information as they want you to see it. This means that to do comparisons you have to visit each of their sites in turn, wait for all the pretty graphics and Java applets to download, type in the address information manually (again), and then copy down the information the system spits back out so that you can compare it with the information from all the other companies.

Suppose instead you could run an application on your local system that would ask for the source and destination addresses, the weight of the crate, and when you need it picked up and delivered. The application could then query *all* of the shipping companies that cover your area. The results would come back, and you might discover that Joe's Moving, based three blocks away, happens to be moving a family to your destination city tomorrow, and will move your crate for one quarter the cost of any of the other companies.

The same type of form (minus the "weight" field, of course ;-) could be used to search for the best way to send *you* via airline to some other city, or to get the best cab, rental car, or mass-transit route information for a local trip.

2) You need a particular kind of spark-plug for your lawn mower. Who has it in stock, and how much will it cost?

Many companies are putting their parts catalogs on the Internet, but only using proprietary data formats and in some cases, proprietary Java front-ends. This makes doing searches like this impossible.

The local hardware might have the part, but what are the odds that they have a WWW site,

given the expense that seems to be required to fluff it up with pretty graphics and to build and maintain a custom CGI application to allow querying their inventory database?

3) You want to purchase a new computer. You have the specifications, and now you want to compare pricing and availability from different companies.

There are several computer vendors that have HTML forms for getting quotes on custom computer configurations, but each uses their own proprietary format. Again, searching across companies is impossible, as is automating searches if you do this sort of thing often.

The same type of query application could be used to specify the configuration for a new car, or even a grocery list, which would allow you to compare pricing and availability of the items on your list from all of the local supermarkets with one operation. Why should everyone have to waste time filling out forms with the same information over and over, and learning the layout of each companies WWW site, wading through graphic and Java-applet downloads on each of them?

4) Your native language is Spanish, and you want the prompts in the query forms you use to solve problems #1-3 above to be in Spanish. Or your eyesight is poor, and you want your form arranged a certain way or using a text font of a certain size.

Certainly it's not practical for each company provide a language-specific Java applet or HTML form. But if companies provided access to their information in standard formats, you could customize the application used to query them as you see fit.

How it will work

Four components are needed for these activities to become practical. The first and most important is published standards for information formats for the information providers. Though standards for this type of thing already exist in individual industries and for Electronic Data Interchange (EDI, see http://www.premenos.com/standards/), we need to greatly expand the scope and use of these standards.

While defining a process for designing these standards may seem to be an intractable problem in itself, similar interoperability problems for data interchange between calendar and scheduling applications from different vendors, and an Internet Engineering Task Force has already been formed to solve them (see http://www.imc.org/ietf-calendar/).

The second component is wide availability of translators to convert data from the proprietary data formats used within organizations to these standard formats. There are many such translation tools for EDI, some produced by ISVs and some by the database vendors themselves. Once these off-the-shelf tools become available, even small businesses will be able to make their information available on the Internet without having to invest in the custom programming required to make "strained carrots" first.

The third component is a way of locating the organizations that offer the type of service or product you're looking for. This kind of search can already be done on most of the Internet search services, but some facility for doing the searches automatically from within other applications will be required.

The final component is the applications that information consumers will use to select and narrow down the list of sites to query, display the query form, handle the connections to the query sites, and collate and display the returned information. These kinds of applications should be easy to develop and easy to customize, and are ideal candidates for implementation using MetaCard, Tcl/Tk, Visual Basic for Applications, or other high-level GUI development tools.

These applications must be flexible and reusable, and people will quickly become adept at using particular applications. Advanced users will tweak the interface to present the information exactly how they want it, and will automate those queries they do on a regular basis. If history is any guide, after a short break-in period users will be extremely reluctant to switch to another application or even upgrade unless substantial new features become available.

Contrast this with the architectures such as Java applets and/or HTML forms, where upgrades are frequent and automatic, and the user has no control over how or when the interface is changed. Most users really hate having upgrades crammed down their throats like this, because it frequently requires them to invest time learning a new interface and working

around compatibility problems when they can least afford the time it takes to do these things.

What can you do?

If you're developing an corporate Internet policy, or even just a Web site, the place to start is to realize that if you base your system on plug-ins, Java applets, or HTML forms and CGI, that you're building a disposable system that most people won't want to use when standards for information interchange become common. Learn what you can about EDI and IETF standards for data interchange, and support them whenever possible, and press for new standards if no existing standards are applicable.

When (not if) standard information formats become widely used, we will all be able to automate repetitive tasks and manage information on the Internet as easily as we manage information locally using PIMs (Personal Information Managers) and custom applications built with scripting language based development tools.

GET METACARD

MetaCard Engine

The MetaCard technology was acquired by Runtime Revolution Ltd. and is now available at the company's site, rebranded under the Revolution product name:

http://www.runrev.com

MetaCard Development Environment

The MetaCard integrated develpment environment (MC IDE) which was built using the MetaCard engine is now available as open source. The MC IDE includes the files mctools.mc, mchelp.mc, and mchome.mc.

While the MC IDE is open source, it requires the Revolution engine which is governed by the terms and conditions outlined in the Revolution License Agreement included in the Revolution package at the URL shown above.

A working group has been formed to maintain and enhance the MC IDE, and the most recent version can be downloaded there for free:

http://groups.yahoo.com/group/MC_IDE/