VUIMS:
A Visual User Interface Management System

Jon H. Pittman, Director of System Architecture
Christopher J. Kitrick, Senior Graphics Programmer

Wavefront Technologies, Inc.
530 E. Montecito St.
Santa Barbara, CA 93103
(805) 962-8117

wavefro!jon@hub.ucsb.edu
wavefro!chrisk@hub.ucsb.edu

Abstract

VUIMS is an object-oriented user interface management system that was designed to support reconfigurable components. VUIMS consists of a collection of objects and a semantically rich token language. The objects implement primitive presentation and interaction functions. The token language controls interaction and visual style. High level objects can be created from primitive objects using token templates. The user interface and application are controlled by token streams that are emitted in response to user actions.

VUIMS supports a variety of presentation and interaction styles through simple, robust manipulation of a hierarchy of visual panels with a rich set of relationships and constraints.

VUIMS has been used to implement two commercial high-performance computer graphics applications and an on-line help system. It has evolved over a three-year period and has proven to be an effective tool in commercial use.

Introduction

Rising expectations of usability and the increasing complexity of software make the design, development, and support of interactive graphics software difficult. In particular, creating usable human interfaces remains an intractable problem. User Interface Management Systems (UIMS) and user interface toolkits address this problem by providing standard user interface components and a system for their composition.

Effective user interface technology must support the interface designer and the design process. It must encourage prototyping, alternative generation, and progressive refinement of design solutions. It must also allow the designer to work easily at both the overall and detail levels. In effect, user interface technology must allow the designer to fluidly manipulate and shape the interface over time to converge on a design solution.

This paper describes the Visualizer User Interface Management System (VUIMS) - a UIMS which supports the iterative design and construction of high-performance interactive graphics applications. VUIMS allows the construction of standard user interface components from primitive objects. These components form a visual vocabulary that can be combined in a number of ways to form user interfaces for graphic applications. VUIMS differs from other UIMS and toolkits in that it is independent of a particular presentation and interaction style. Components are configured through an interpreted token language to implement a variety of user interface styles and behaviors. This token language implements application actions in addition to interface actions.

This paper describes the internal structure, use, and implementation of VUIMS. It includes the following information: an overview of the system, a discussion of related work, the VUIMS architecture, an example object configuration, a description of the VUIMS kernel, a description of VUIMS objects, a description of the token language, a discussion on implementation and evolution of VUIMS, and conclusions and future plans.

© 1990 ACM 089791-410-4/90/0010/0036 $1.50

Overview

VUIMS achieves the following goals:

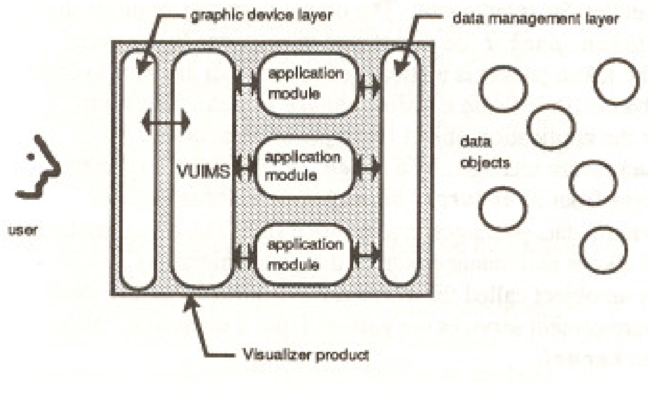
- **separation of application from user interface** -- The application modules do not know anything about the user interface driving them. Interface development and prototyping can take place separately from the application development.

- **user control** -- The user interface monitors and responds to user behavior and asks the application modules to perform user-requested tasks. Application modules are subservient to the user interface which is, in turn, subservient to the user.

- **extensibility** -- The user interface can be modified or extended without major disruption to the user interface software or the application modules.

- **external syntax and semantics** -- The visual style of the user interface, as well as the behavior invoked by user actions, are not embodied in the user interface software. They are embodied in tokens that are stored in a data file external to the executable. This allows syntax and semantics to be changed without recompiling the software.

Product architecture. VUIMS is part of an overall architecture for a family of interactive computer graphics, animation, and visualization software products. This architecture is shown in the figure below.



Visualization product architecture

In the visualization product architecture, VUIMS is one of a set of software layers. Each layer has a specific, independent function described as follows:

- **graphic device layer** -- manages the input and display devices and abstracts hardware-specific implementation details from the user interface.

- **VUIMS** -- is responsible for presenting graphics and user interface controls to the user and translating user actions embodied as input events into work requests for application modules.

- **application modules** -- respond to work requests from the VUIMS and perform the specific work that the user requests. A given product may have several application modules.

- **data management layer** -- reads and writes application data to the file system and coordinates data access between application modules.

As shown in the preceding architectural diagram, VUIMS communicates with the user through an abstract graphics layer. This layer is responsible for presenting all visual elements that the user sees on the screen and for translating all user input events into meaningful work requests for application modules.

Related Work

The inspiration for VUIMS came from two bodies of related work: UIMS research and object-oriented programming. VUIMS was also influenced by our desire to create a portable, extensible, user interface layer for our products that could be customized for specific user classes and international locales.

VUIMS has drawn from several threads in user interface research. It is not a pure model of any particular UIMS approach or object-oriented programming paradigm, but rather combines a number of good ideas into an object-oriented UIMS that is appropriate for developing commercial products.

User Interface Management Systems. UIMS research has been concerned with developing a layer of software to handle user interaction separately from the application [PFAF85]. UIMS software typically provides a set of standard components and some form of external description. A user interface designer builds these components using the external description. The UIMS then reads this description and generates the user interface for the application. Two difficulties encountered with UIMS are a fixed design vocabulary and a limited ability to control the application because of a split between the user interface and the application.

VUIMS overcomes both of these problems. The first problem is addressed by supporting multiple levels of configuration. The second is addressed by having the token language drive both the interface and the application modules.

Two areas of current interest in UIMS research are developing interactive design tools [MYER89, SING88] and measuring user interaction [OLSE88]. VUIMS currently provides minimal support for interactive design tools and no support for measuring user interaction. We plan to strengthen these areas in the future.

Object-oriented programming. Object-oriented programming techniques have been viewed as particularly appropriate for graphic user interfaces [SCHM86, COX86] and for overcoming some of the problems of standard UIMS [KNOL89]. In addition, UIMS have been implemented using object-oriented techniques [SIBE86]. VUIMS implements basic object-oriented programming concepts such as classes, instances, data encapsulation, and message passing, in the spirit of Smalltalk-80 [GOLD89].

The InterViews C++ user interface toolkit [LINT89] demonstrates many of the concepts of hierarchical object composition that are embodied in VUIMS.

Further sources of inspiration were a multi-process paint system [BEAC82] and a multi-process user interface [TANN86] in which user interfaces were composed of networks of small processes, each focused on a specific task. Although VUIMS is implemented as one single-threaded process, it is organized into discrete objects to simulate multiple threads of control and concurrency.

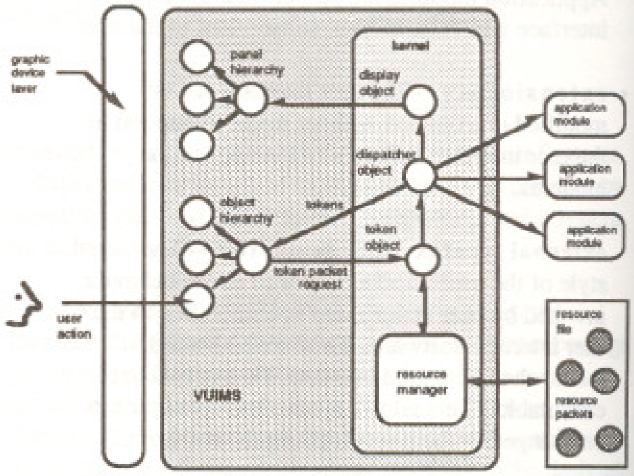
VUIMS Architecture

Structure and components. VUIMS consists of a hierarchical collection of ***objects***. Objects may be physical, in that they have a visual manifestation on the display device, or logical, in that they provide services to other objects or manage relationships between objects. Each object contains internal state data and methods that transform the object's state or its visual representation.

Objects communicate with each other by sending messages embodied in ***tokens***. Tokens are character strings that are interpreted by objects to invoke methods. These methods modify the state of the object to which they belong. In effect, tokens are asynchronous messages that objects use to communicate with each other.

Tokens may contain ***symbols*** which refer to the internal state of objects. Physical objects manifest themselves on the display device through a hierarchical collection of ***panels***. Panels are rectangular regions of the screen that are implicitly connected to physical objects. When a physical object receives a request to display itself, it notifies the panels associated with it by sending a message to the ***display object***. The display object manages the entire panel hierarchy. When the display object receives a request to display a panel, it also manages damage to nearby regions caused by the panel redraw.

The architectural diagram that follows shows the internal structure and the major components of the VUIMS.

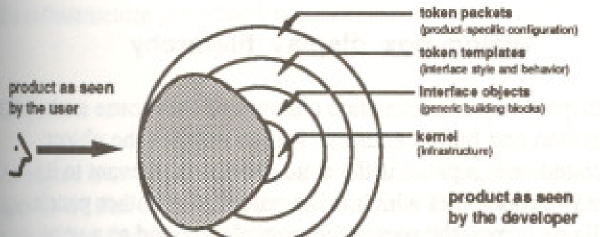


Structure of VUIMS

When an external event is caused by a user action, such as clicking the mouse or typing on the keyboard, an object handles the input event. The object normally requests that a ***token packet*** be dispatched in response to the event. The token packet is a stream of tokens that are sent to other objects (to indicate a visible change in state, for example) or the application object for application actions. Token packets are managed by a ***token object*** which requests them from a ***resource manager*** that stores them as generic data packages in a resource file. The actual routing of tokens and management of the object hierarchy is done by an object called the ***dispatcher***. The object and token management services are gathered into a subsystem called the ***kernel***.

Note that the ***application modules*** are treated as objects by VUIMS. They receive tokens and respond to them like any other VUIMS objects. The only difference is that they have no knowledge of other objects and cannot directly interact with them.

Configuration layers. Interface style and application control are embodied in tokens. Since the tokens are interpreted rather than compiled, they can be manipulated interactively without a recompile. To make the token mechanism more powerful, ***token templates*** are used. Token templates are sets of tokens that specify both visual style and object behavior. They are executed to establish initial object configuration. An entirely new style can be implemented by modifying the underlying token templates.



User Interface is layered to allow multiple levels of product configuration

As the preceding drawing illustrates, VUIMS can be viewed in layers. As one moves in from the surface, the layers become increasingly generic. The kernel is a generic set of mechanisms used in every product. The outer layer is product-specific. The following layers are shown:

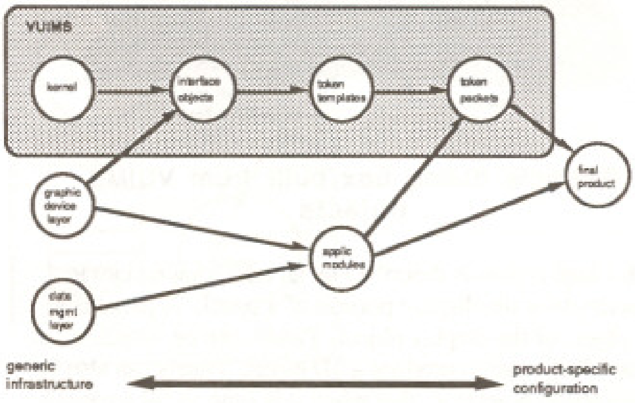
- **Token packets** glue the lower, more generic elements together to form products. They embody the presentation and interaction that the user experiences. Token packets are product-specific and invoke application actions.

- **Token templates** implement collections of generic objects and functionality. They define composite objects and specify actions. They embody a specific interface style. Token templates are generic to a specific style of user interface but independent of a particular product.

- **Interface objects** are simple, generic interaction and presentation primitives. They are implemented in C code and have well formed and bounded behaviors. They are the primitive elements from which all user interfaces are built.

- **The kernel** consists of the dispatcher, the token object, and the resource manager. It orchestrates the activity of the objects and the token templates and packets. The kernel is completely independent of all applications and higher level interface constructs.

Configuration process. In the following sections, we will examine each of the major subsystems of the VUIMS in detail. Before we do that, let's look at the process of assembling a product using VUIMS.



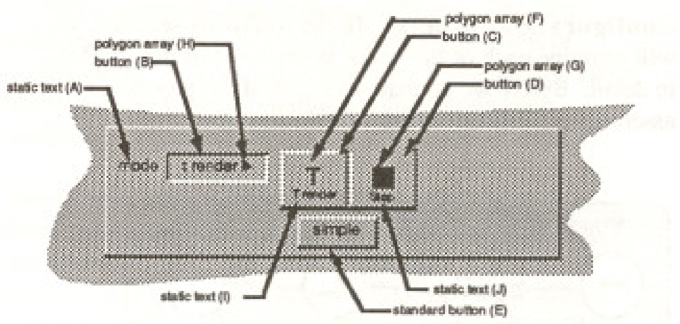
Generic components are assembled to form a specific product

As products are built using VUIMS, the product developers take generic components and assemble them with increasing levels of specificity to form the final product. The product developer is primarily concerned with ***configuring*** a specific product from generic components. Thus, he or she works to integrate high-level chunks of functionality as embodied in token templates and application modules. The developer configures the product by writing token packets that embody interface layout, application actions, and interface behavior.

Templates and application modules are typically produced by someone other than the product developer. Thus, the product developer is really the consumer of components produced by others. This distinction is further reinforced by the fact that product assembly typically does not require programming skills, while production of the base components is a programming activity.

Example

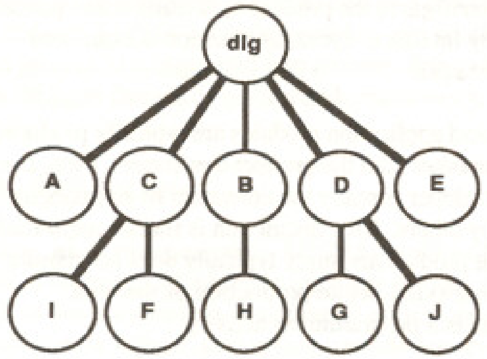
A simple example illustrates the composition of generic objects to form a complex object. Following, a simple dialog box is constructed from a hierarchy of VUIMS objects. Each button in the dialog box, including the depressed button, is a hierarchy of smaller objects. The generic objects in this example are the button, static text, and the polygon array.



Example dialog box built from VUIMS objects

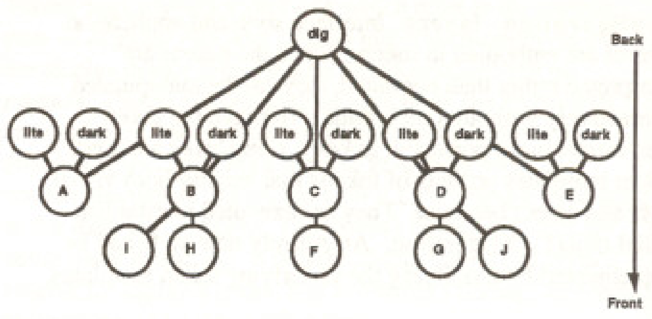
Each visual region is described by an enclosing rectangle. A rectangle is the display portion of a panel. A panel is a sub-object of the display object. Panels can be combined in hierarchical order to produce a 3D effect. Panels can also be connected to objects so that when it is time to fill a panel's bounding rectangle, its associated object is notified to update the region. A panel without an object can only have a color attribute. Thus, the shadows, borders, and frames have no objects associated with them.

The object and panel hierarchies are illustrated in the following two diagrams. The two trees are disjointed but related since panels are associated with objects. It is possible to modify the object hierarchy without affecting the display tree.



Dialog box object hierarchy

In the display hierarchy, child panels can be either behind or in front of the parent, (e.g. shadow and light bars). Panels such as I, J, F, and E can be made transparent to mouse events so that a selection of button C is not intercepted by the intervening panels.



Dialog box display hierarchy

The display object tracks state changes of the mouse such as position and button actions. It then notifies the object associated with a panel if the state change is relevant to it. There is a root panel which is the parent of all other panels. For dialog boxes, the root is temporarily defined as a very small subset of the display tree, in this case, "dlg." Activity outside of this group is then undefined. The dialog box example illustrates the use of hierarchy to control appearance as well as input flow.

The VUIMS Kernel

The kernel is a suite of basic services that controls object creation, deletion, position in the tree, message storage and recovery, symbol translation, panel creation, message delivery, and input focus. These services comprise the infrastructure upon which the objects embodying VUIMS functionality are built. The elements of the kernel are themselves embodied in and controlled through objects. The following objects comprise the kernel:

- The **dispatcher** creates and deletes objects and maintains their hierarchy. It also translates symbols and delivers messages.
- The **display object** is the graphic counterpart of the dispatcher. It maintains the hierarchy of visible panels on the screen. It also orchestrates the flow of actual input devices such as the mouse and keyboard.
- The **resource manager** allows essentially random access to blocks of data stored in a file. Every object can store its state as a resource. The resource manager controls all access to the data stored; it has no knowledge of the resource contents.

- The **token object** is the object that manages the creation and distribution of token packets. The token object uses the resource manager to save and restore token packets in a resource file.

These objects comprise the backbone of VUIMS. They coordinate object management and the flow of communications in the form of token streams between objects. The remaining functionality of VUIMS is embodied in individual objects working together through the infrastructure provided by the kernel.

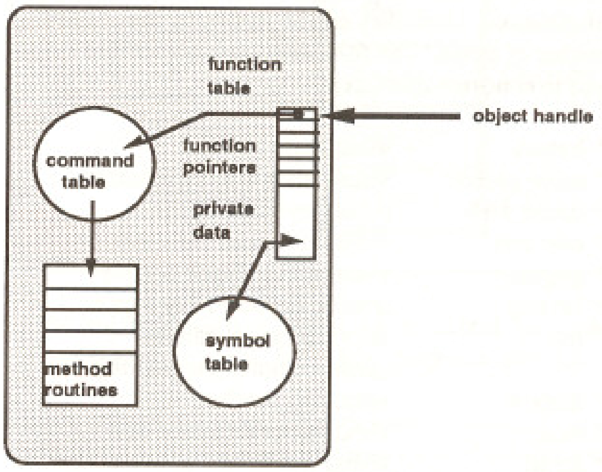
Objects

An object is an encapsulated set of data and functions that operate on that data. Access to objects is directly through a set of functions called only by the dispatcher. Communication between objects occurs through messages that invoke methods unique to each object class as well as those common to all objects.

Object access functions. Each object is uniquely identified by a ***handle***. The handle for each object points to data private to the object and a table of object access function pointers. These functions are called by the dispatcher in response to various events occurring in VUIMS. VUIMS passes the object handle to each routine so that the routine can access the object's private data. The following object access functions may be available for an object. The ***init*** function is required to create an object instance. All other functions are optional.

• **init**	-- allocates memory to hold state information and initialize function pointers
• **exit**	-- releases object's memory
• **display**	-- draws the object on the display
• **action**	-- processes messages sent to the object
• **signal**	-- handles asynchronous interrupts
• **input**	-- requests input from the object
• **output**	-- sends output strings to the object
• **error**	-- sends an error message to the object
• **save**	-- saves object state in the current resource file
• **restore**	-- recovers object state from the current resource file

The following figure illustrates the components of an object. The function table is owned by the object and only accessible from the dispatcher. Private data is allocated and maintained by the object and some of it may be accessible through the symbol table provided. The command table is used to parse messages which arrive from the dispatcher. These messages invoke method routines in response to the message contents.



Anatomy of an object

Object types. There are four types of objects: built-in, class, instance, and pseudo. These types are defined as follows:

- **Built-in objects** are one-of- a-kind, typically providing some central service. For example, the color object allows interface color tables to be defined, edited, saved, and restored as resources by the interface designer at any time. The following built-in objects are provided with VUIMS:

* array	- manipulates array data resources
* color	- interface color control
* condition	- error trapping
* cursor	- cursor shape and visibility control
* device	- frame buffer control and access
* dispatcher	- token / input flow control
* display	- panel manager
* file	- file access
* if	- logical if manager
* interrupt	- physical device (keyboard, mouse) handler (journaler)
* null	- bit bucket
* parser	- language
* program_sym	- program symbol access
* program	- program access
* reader	- script file reader
* resource	- resource file control
* signal	- handle UNIX signals
* switch	- logical switch manager
* token	- token builder/emitter

• **Class objects** are templates used to create instance objects. They do not maintain any state information. Thus, they can be instanced any number of times. The following class objects are used to construct interfaces:

* button	- visual base button
* color_picker	- visual color display/selection
* debug_kbd	- debug keyboard
* edit_text	- visual mouse-editable text
* graphic	- visual text/polygon display
* keytrap	- specific keyboard key trapper
* list	- information list handler (multi-column/non-graphic)
* listport	- visual list handler
* meter	- visual meter for scalar data
* scroll	- visual scroll bar
* static_text	- visual static text
* symbol	- symbol table
* textfilter	- context-sensitive text filter
* viewport	- visual display list port

• **Instance objects** are incarnations of a particular class definition. The dispatcher holds the state of an object which is passed to the class whenever it is invoked.

• **Pseudo objects** are pointers to other objects. By default, three pseudo objects are created: input, output, and error. At any time, a pseudo object can be pointed to another object. Whenever a pseudo object is referenced, it is redirected to the actual object to which it points. The following pseudo objects are used to route events:

* input	- pointer to object to handle input
* output	- pointer to object to handle output
* error	- pointer to object to handle errors

Message parsing. All messages arrive at an object from the dispatcher as tokens and are parsed by the object's ***action*** function. This function parses the message by passing it and a pointer to the object's command table to a library routine which finds the appropriate table entry and executes the corresponding method function. Messages perform one basic purpose: they alter the state of an object.

Symbol table. Objects contain a symbol table that holds both user-defined and intrinsic symbols. The dispatcher uses this symbol table when translating symbols embedded in tokens. Intrinsic symbols are automatically created by the object to reflect object state. They generally allow messages to use the data associated with the object's state. For example, in a dialog where the user is expected to type in a string and verify acceptance, the edit text object contains a symbol that indicates the number of characters in the edit buffer. This symbol can be used to determine whether verification is a valid choice.

Display. If an object has a visual component, it must be associated with a panel in the display tree. That panel determines back to front order, position, and size. When the panel is drawn, the object is asked to draw its contents in a rectangular region that is passed to the object's ***display*** routine.

The Token Language

Tokens are the delivery vehicle for messages. A token consists of the following parts:

• **source**	- object that sent the message (implicitly maintained by the dispatcher)
• **target**	- object for whom the message is intended
• **priority**	- order of delivery
• **message**	- a string that is parsed by the target object to invoke a method

Messages alter the object state and may contain symbols that refer to state data belonging to the source object or any other object.

Symbols. Symbols are references to object data and are thus object-specific. They are used in messages to allow actual content to be substituted at the time the message is delivered. Symbols make it possible for objects to share knowledge of their state so that multiple objects can work together. Thus, object functionality can be kept small and generic. Small objects can then be combined to form complex composite objects. Symbols are differentiated from other words in the message string by a leading "$" character, followed by the object path and the name.

$/a/b/c/d - path /a/b/ : object c : name d

The object path reflects the position of the object in the dispatcher's object tree. Symbol definitions have an implied path equal to the path of the source object of the message. A path can move back up the tree with the "\" character. For example, if object b was the source of the previous message, the alternative reference would be:

$c/d - (*path /a/b/*) : object c : datum d

Priority. Each token has a delivery ***priority***. The priority indicates the order in which the token is delivered relative to other tokens. A token may have one of the following priorities:

• **immediate**	- bypass tokens awaiting execution and deliver synchronously
• **interrupt**	- deliver before other tokens
• **normal**	- deliver in sequence sent
• **defer**	- deliver after other tokens

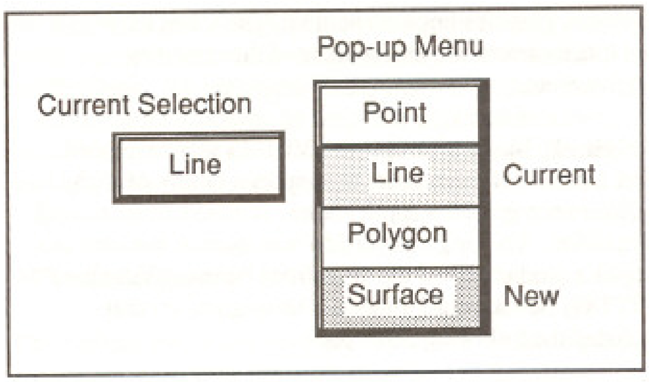
Token packets are coherent groups of tokens that are sent en masse. They have unique names and are maintained by the token object. All objects that can send messages in response to an external user event, such as a mouse down, maintain the name of the token packet associated with that event. When the object receives an event and wishes to emit its associated token packet, it issues a request to the token object. The token object then emits the packet by delivering each token in the packet to the dispatcher. The dispatcher then distributes the tokens to their respective targets.

The following example packet is issued from the selection of an item in a pop-up menu. The initial button that brings up the pop-up contains the value of the current setting. Each token has normal priority. The sender of the token packet is the surface button.

target	message
display	color $fullname $/color/fil_on
display	color $\current $/color/fil
program	mode $text
$\fullname	define current $fullname
$\fullname	text $text

The tokens above perform the these actions in the following example pop-up menu:

- change color of self to selected color
- change color of last selected to neutral
- inform program of current mode
- define new current definition in parent
- define new default text to be seen



Example token packet is emitted by the surface button on this pop-up menu.

Direct Manipulation. We originally allowed applications to directly inquire input values from the mouse to handle direct manipulation of graphic elements. We found, however, that token performance was sufficient to allow all direct manipulation to occur through token processing. Now, even the smallest interaction is handled through token processing. This frees the application from having to know specifics about the input device. And makes it much easier for the developer to control interaction.

Performance. In all interaction, token processing time is negligible. The amount of time necessary to process a token is fairly low, even with complex tokens. We profiled performance-critical loops where a great deal of graphics activity is occuring (moving shaded objects in real time in perspective, updating screen values, and swapping display buffers for smooth motion) and found that the overall cost of message parsing was 10-12%. That consisted of numerous smaller components in the 1.5% or less range. The large time components for graphic interaction occurred in the actual graphics display code.

Tokens are the controlling mechanism for VUIMS. They define and control all of the activity generated by user actions. They carry messages between objects and to the application modules. Thus, the process of configuring a product-specific user interface using VUIMS is really a process of programming using the token language.

Implementation and Evolution

The VUIMS components (kernel services and objects) are implemented in C on the UNIX operating system. They are designed to be portable and have been ported to a number of graphics workstations. The graphics device layer insulates the VUIMS from the details of the particular hardware graphics implementation. The token templates and token packets are independent of the operating environment.

Original implementation. VUIMS was developed over a period of three years to support a family of high-performance graphics applications. It is now in its second generation. The original version was used to implement a graphics product called the Wavefront Personal Visualizer™ [PITT89] (see attached slides). The original version included the following concepts:

- strict separation of application and interface
- presentation and interaction embodied in a collection of objects
- a dispatcher to control token traffic and object instancing
- symbols to supply object semantic information to tokens

In implementing the Personal Visualizer, we found several problems with the original version of the VUIMS:

- **object granularity** -- The objects we provided directly embodied too much interface style and behavior. Thus, they were unwieldy and limited.

- **configuration complexity** -- The process of configuration was difficult. Because there was no hierarchy of configuration operations or support for token language modularity, it was a laborious hand-crafted process.

- **object and display hierarchy** -- Built-in mechanisms to manage an object and display hierarchy did not exist.

- **token flow** -- Mechanisms to control and sequence tokens were insufficient.

- **semantics** -- The symbol mechanism was not rich enough to provide implicit access to object state.

- **token management** -- Token packets were stored in text files and read on system startup. This led to a long delay on system invocation.

- **programming consistency** -- As the first version of VUIMS evolved over a period of time and was worked on by several developers, various aspects of the system grew inconsistent. In particular, each object was implemented differently. The product designers and implementors had to understand the quirks and personality of each object.

Second generation. To overcome these shortcomings, we rewrote the VUIMS to incorporate a number of new features and innovations. These innovations included:

- resource management
- a sophisticated object and display hierarchy
- an extended symbol system
- later binding of symbols
- improved token flow control mechanisms
- smaller, more compact, generic objects
- token templates
- interactive configuration aids
- elimination of direct control of the mouse by the application

The second generation of VUIMS was used to develop a new version of the Wavefront Personal Visualizer™ and a new scientific visualization package called the Data Visualizer™. In addition, we were able to develop an online help system using the second generation VUIMS that was primarily user interface. This help mechanism required very little application code to support it.

The second generation was built on the foundation created by the first version. We learned from our implementation experiences that the difficulties encountered with our initial version were failures of timidity rather than failures of boldness. The second generation of VUIMS took the original design much further and resulted in a much more successful implementation. In addition, the flexibility of the second generation allowed us to create a more modern visual style.

Conclusion and Future Development

VUIMS has enabled us to successfully build a set of commercial applications with rich, effective user interfaces. As we have developed our applications, VUIMS has supported the prototyping and iterative design necessary to create compelling graphic products. In addition, it has allowed us to experiment with several different user interface styles.

We have developed a library of user interface objects for use in future applications. By providing a proven, reliable set of user interface components that can be reconfigured with minimal effort, we hope to reduce user interface development time and increase the user interface quality in our products. VUIMS has changed our software development process from that of handcrafting software to one of assembling products from a set of known components. We believe that VUIMS provides a model that can be used to improve software product development and productivity.

We plan to expand and enhance the capabilities of VUIMS. There are several areas of future development planned for VUIMS:

- **multi-process model** -- Currently, VUIMS and its application modules are implemented as one monolithic process. We plan to migrate it to a multi-process model in which VUIMS acts as a user interface server for a number of client application processes. Eventually, we hope to further decompose VUIMS into a multi-process architecture similar to the architectures proposed by Beach [BEAC82] and Tanner [TANN86].

- **interactive configuration tools** -- One of the most powerful notions behind VUIMS is the ability for non-programmers to build products. Currently, the methods for configuring products are still somewhat cumbersome and complex. We plan to make this process easier by providing graphic constraint-based tools similar to Prototyper™ by Smethers-Barnes [SMET89], the NeXT user interface builder [THAK90], and those described in the research community.

- **multiple interface styles** -- Although we have isolated visual style issues to the token templates, we have not experimented with a wide range of styles. We would like to try to emulate several popular interface styles such as Motif™ and OpenLook™ to see how truly malleable our system is.

- **mixed control** -- We wish to experiment with allowing the application to participate in controlling the interaction in certain cases, such as steering a large scientific visualization application.

- **support for more visual data types** – Currently, we support standard menu and icon-based user interfaces. In the future we plan to offer more visual representations for data. This includes more pictorial icons and visual representations of animation sequences and components.

- **international support** -- We plan to provide support to allow customization for multiple international locales.

We plan to keep developing and enhancing VUIMS and continue using it in commercial graphics and animation applications. We anticipate that additional implementation experiences will provide us with new opportunities and challenges in making VUIMS a truly useful and effective tool to support high quality user interfaces.

Acknowledgements

The authors would like to thank Wavefront Technologies, Inc. for its support of the work presented in this paper. VUIMS was developed for Wavefront's Personal Visualizer™ product line.

Derry Frost was the product designer for the Personal Visualizer™ and was very influential on the external design of the user interface. Julie Daily and Susan Papallardo configured products using the VUIMS. Their efforts made the VUIMS successful. In addition, Julie's comments and experience proved invaluable in evolving the VUIMS architecture. Trish Soriano provided programming support. Kim Shelley was involved in the early stages of the project and built several of the early objects for VUIMS. Finally, Roy Hall, Dave Immel, and Matt Arrott provided valuable design input and guidance in the early stages of the project.

The Data Visualizer™ was developed by Don Brittain, Josh Aller, and Brad Weed. Brad also developed the on line help mechanism with VUIMS.

Diane Ramey and Karen Gadway provided editorial assistance in writing this paper.

References

[BEAC82] Richard J. Beach, John C. Beatty, Kellog S. Booth, Darlene A. Plebon, Eugene L. Fiume. The Message is the Medium: Multiprocess Structuring of an Interactive Paint Program. *Computer Graphics (Siggraph '82)* Volume 16, Number 3. July 1982.

[COX86] Brad J. Cox. *Object-Oriented Programming, An Evolutionary Approach.* Addison-Wesley. 1986.

[GOLD89] Adele Goldberg and David Robson. *Smalltalk-80, The Language.* Addison-Wesley. 1989.

[KNOL89] Nancy T. Knolle. Why Object-Oriented User Interface Toolkits are Better. *Journal of Object-Oriented Programming*. November/December 1989. Pg. 63-67.

[LINT89] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing User Interfaces with InterViews. *IEEE Computer*. February 1989. Pg. 8-22.

[MYER89] Brad A. Myers, Brad Vander Zanden, Roger B. Dannenberg. Creating Graphical Interactive Application Objects by Demonstration. *Proceedings of the ACM Siggraph Symposium on User Interface Software and Technology (UIST '89)*. Nov. 13-15, 1989. Pg. 95-104.

[OLSE88] Dan R. Olsen, Jr. and Bradley W. Halversen. User Interface Measurements in a User Interface Management System. *Proceedings of the ACM Siggraph Symposium on User Interface Software*. October 17-19, 1988. Pg. 102-108.

[PITT89] Jon H. Pittman. The Render Button: How a high-end graphics company developed a personal graphics product. *Monterey Computer Graphics Workshop Proceedings*. Usenix Association. November 1989. Pg. 99-113.

[PFAF85] Gunther E. Pfaff, Ed. *User Interface Management Systems*. Springer-Verlag. 1985.

[SCHM86] Kurt J. Schmucker. *Object-Oriented Programming for the Macintosh*. Hayden Books. 1986.

[SIBE86] John L. Sibert, William D. Hurley, Teresa W. Bleser. An Object-Oriented User Interface Management System. *Computer Graphics (Siggraph '86)*. Volume 20, Number 4. August 1986. Pg. 259-268.

[SING88] Gurminder Singh and Mark Green. Designing the Interface Designer's Interface. *Proceedings of the ACM Siggraph Symposium on User Interface Software*. October 17-19, 1988. Pg. 109-116.

[SMET89] Smethers-Barnes. *Prototyper™ Manual*. Smethers-Barnes. 1989. Portland, Oregon.

[TANN86] Peter P. Tanner, Stephen A. MacKay, Darlene A. Stewart, and Marceli Wein. A MultiTasking Switchboard Approach to User Interface Management. *Computer Graphics (Siggraph '86)*. Volume 20, Number 4. August 1986. Pg. 241-248.

[THAK90] Umesh Thakkar, Gary Perlman, and Dave Miller. Evaluation of the NeXT Interface Builder for Prototyping a Smart Telephone. *SIGCHI Bulletin*. Volume 21, Number 3. January 1990. Pg. 80-85.