Khoros: A Problem Solving Environment for Scientific Computing and Visualization

Danielle Argiro · Steve Kubica · Mark Young

· Jeremy Worley · Steve Jorgensen

Khoral Research, Inc. *

Version 2

March 17, 1998

The meaning of the word "Khoros"

"Khoros" is not an acronym; rather, it is the Greek spelling of the word chorus. A definition of a chorus is: a company of singers or dancers who sing or dance in concert. The name Khoros is symbolic of the objectives for the khoros project and software. The singers or dancers are the interacting systems of Khoros; the objective is to provide a group of systems that will, when used in concert, provide a harmonious environment for its many different kinds of users. These same themes are expressed in the definition: a composition to be sung

*http://www.khoral.com/

by a number of voices in concert. A different definition of chorus, a company of singers and dancers in Athenian drama participating in or commenting on the action, is symbolic of the open technology of Khoros and the participation of the user community in the development of Khoros.

1 Introduction

Khoros [1] is an integrated problem solving environment which allows researchers and scientists to perform a variety of tasks related to scientific computing, including image and signal processing, data manipulation, and scientific visualization. Khoros contains over 300 operators for information processing, data exploration, image and signal processing, and data visualization. Generalized to form a broad base-level technology, these operators facilitate problem solving in a wide variety of application domains used in research, science, government and industry. All Khoros operators may be run independently from the command line or executed from within the visual programming environment, *Cantata*. With Cantata, visual programmers can combine the scientific operators of Khoros in a graphically expressed, data flow visual language for experimentation, data exploration, and to rapidly prototype new solutions.

[1] KRI supports two versions of the Khoros technology: Khoros Pro and Advanced Khoros. Khoros Pro is the commercial version of Khoros; as of this writing, the current version is Khoros Pro 2.2, released in July of 1997 on CD-ROM. Advanced Khoros is the rapidly changing R&D version of the Khoros technology, available over the internet to a select audience of government customers; as of this writing, the Advanced Khoros 2.3 is scheduled for release in summer of 1998. In this paper, the word "Khoros" will be used to refer to the technology underlying both Khoros Pro and Advanced Khoros.

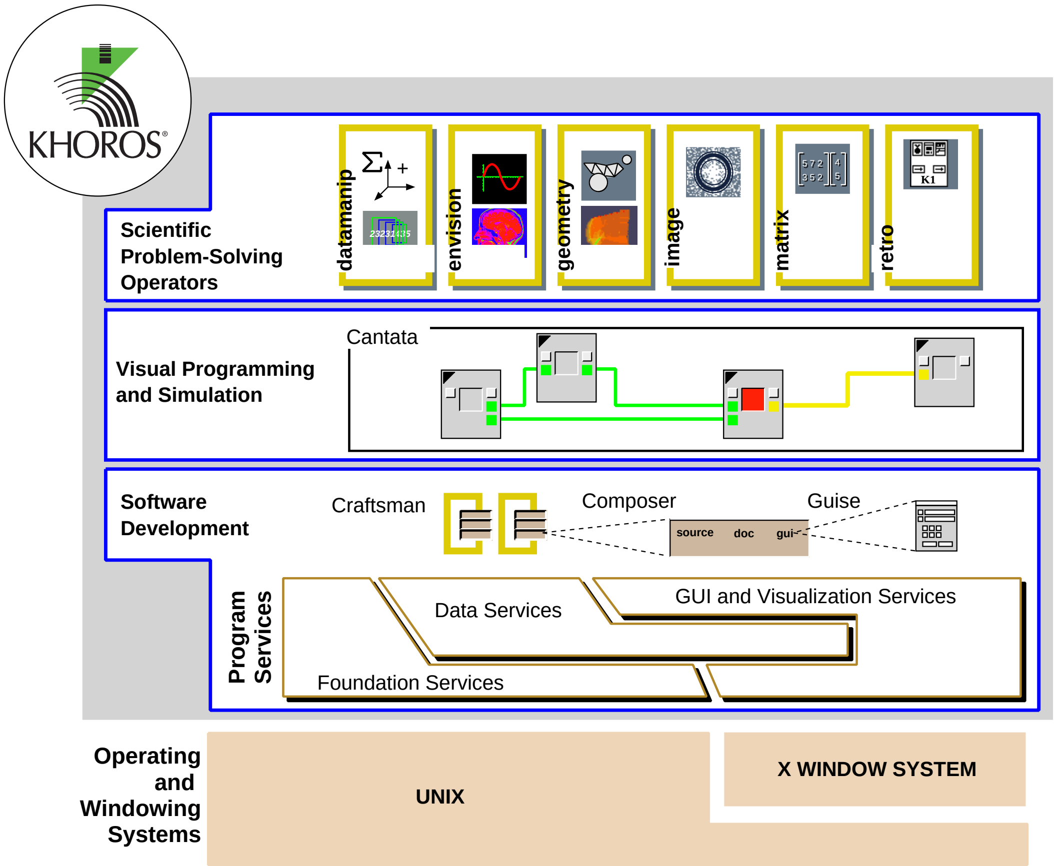

Figure 1: Khoros is designed to support users at many different levels. Application end users can use Khoros as a problem solving environment, using any of the hundreds of data processing and visualization operators available. The visual programming language, Cantata, can be used to interactively connect these operators into any configuration. Khoros also contains a software development environment which allows users to extend the system if needed by developing new custom operators or applications.

Designed to act as an integrated software development environment, Khoros allows you to write new data processing operators and create new visualization applications. The *Cantata* visual programming environment will automatically include any new user created operators, making it effective as a software integration environment. All the data processing and visualization routines in Khoros utilize the capabilities provided by the Program Services that comprise the Khoros system infrastructure. The three major Program Services of Khoros are Foundation Services, Data Services, and GUI & Visualization Services, which deal with portability and operating system issues, data access and manipulation, and scientific visualization, respectively.

2 Scientific Problem-Solving Operators

For problem solving by researchers and scientists, Khoros provides a wide variety of programs, or operators, for information processing, data exploration, and data visualization. Khoros programs are typically generalized to form a broad base-level technology rather than any specific scientific area. Some of the application domains in which Khoros is used include: high performance computing, modeling & simulation, applied research, medical imaging, image understanding & analysis, remote sensing, and automatic target recognition.

All Khoros programs provide automatic data format conversion, data type conversion, file format conversion, support of large data sets, and support for multidimensional data. Khoros programs offering end-user solutions to scientific problems fall into a number of categories:

- **Multidimensional (1D to ND) data manipulation operators**, including data import, data export, data generation, single & double operand arithmetic, trigonometric and nonlinear functions, bitwise and complex operators, comparison operators, linear transforms, histograms, size & region operators, threshold & clip operators, noise generation / introduction, clustering operators, FFTs and convolutions.

- **Image processing** of multiband images, including convolution, data compression, feature extraction, frequency filters, geometric operations, nonlinear filters, and spatial filters.

- **Matrix operations**, including extraction of rows, columns, and diagonals, computation of eigenvectors and eigenvalues, computation of the inverse matrix, computation of the least-squares solution to a set of linear equations, LU decomposition, computation of the matrix product, and singular value decomposition.

- **Data visualization applications**, including an image sequencing/animation program, an image display & manipulation package, an ROI extraction program, an image capture program, a 2D/3D plotting package, a non-interactive data display tool, a colormap alteration tool, and an interactive image/signal classification application.

- **3D visualization and software rendering** which consists of a 3D geometry library, a number of data processing routines for 3D visualization, and a software rendering application.

The primary objective in developing these operators was to facilitate domain-specific work while simultaneously enabling cross-domain collaboration. When designing the data processing and visualization operators, several issues critical to increasing the productivity of the scientist in solving data processing and analysis problems were addressed. These issues include providing domain interoperability in order to promote the reuse of software solutions over diverse domains; providing format and system independence, including the capability to process very large data sets; providing the ability not only to visualize data using traditional methods, but to allow for data exploration; and instilling confidence in these tools by ensuring reliability and stability. This approach provides flexibility to the end user and promotes collaboration within and across application domains.

The data operators in the Khoros application toolboxes are all written using a common polymorphic data model that ensures consistent data interpretation across a broad set of domains. This model can be used to represent data for application domains as diverse as image processing, volume processing, signal processing, computer vision, and numerical analysis. The polymorphic model exploits the common interpretation of data across these different domains to facilitate the interoperability of data manipulation routines. This interoperability between the data in the different data models provides the benefit of reuse. For example, a single FFT routine can be used to process signals, images, volumes, and even animations.

The use of the polymorphic data model (see section 6.4) allows different disciplines to share solutions to problems that differ with respect to application domain but not with respect to algorithmic technique. The ability to mix domains and disciplines means that scientists can take advantage of research done in the broader scientific community and quickly achieve results to advance their own research.

3 Visual Programming Environment

Visual programming environments provide iconic elements which can be interactively manipulated according to some specific spatial grammar for program construction. Data flow is a naturally visible approach in which an algorithm is described as a directed graph where each node represents an operator or function and each directed arc represents a path over which data flows. By connecting the data paths between nodes, users can interactively draw out a solution in an intuitive manner that matches their mental representation of the problem. Thus, a visual environment for problem solving introduces a level of abstraction which provides accessibility to the functionality represented by the underlying operators, regardless of the developer's programming experience; moreover, it increases the productivity of both experienced and novice developers.

The extensive image processing, data manipulation, matrix processing, and scientific visualization operators included in the Khoros system can all be interconnected within *Cantata*, the Khoros visual programming environment. Visual hierarchy, iteration, flow control, and expression-based parameters extend the data flow paradigm to provide a powerful prototyping, experimentation, simulation, and problem solving system.

Programs from the Khoros system are represented as visual objects called *glyphs*. The program which corresponds to the glyph is called an *operator*. To create a visual program, the user selects the desired operator, places the corresponding glyphs on the workspace, and connects the glyphs to indicate the flow of data from operator to operator, forming a *network* within a *workspace*.

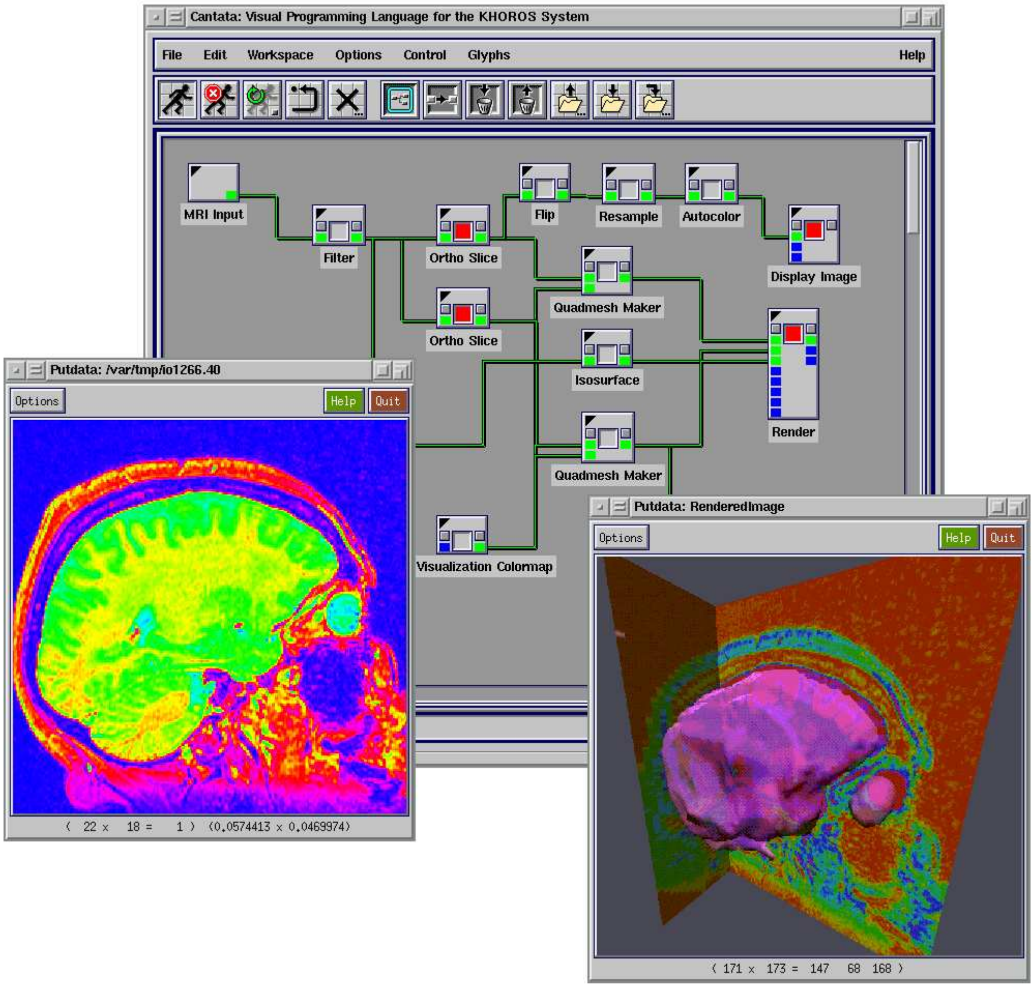

Figure 2: A visual program in Cantata.

Control structures can be used to branch and merge data flow, or to implement loops. Workspaces can be executed, saved and restored to be used again or modified later. Workspaces may also be encapsulated into independent applications with a very simplified graphical user interface so that they may be treated as stand-alone Khoros applications.

Visual hierarchy, iteration, flow control, and expression-based parameters extend the data flow paradigm to allow Cantata to be used as a simulation and prototyping system. Data and control-dependent program flow is provided by *control structure glyphs* such as if/else, while, count, and trigger. Visual subroutines, or *procedures* are available to support the development of hierarchical data flow graphs. Variables may be set interactively by the user, or calculated at run time via mathematical expressions tied to data values or control variables within the visual network.

By combining a natural environment of visual constructs with the programming features typically found in textual languages, Cantata provides a powerful problem solving and prototyping system. Visual hierarchy, iteration, flow control, and expression based parameters augment the traditional data flow paradigm so that Cantata can be used effectively in a number of domains, including process control, simulation, and system integration. When combined with the data processing and data visualization programs available in the Khoros system, Cantata is particularly well suited for scientific data processing and visualization.

3.1 Data/Control Connections

Glyphs contain input and output *data connection nodes.* When two glyphs are connected with a *data connection*, it is implied that the output of the first will become the input of the second; thus, the data connection represents data flow in the visual program.

A visual program requires data connections between glyphs in order to form the network and to define where each process will obtain its data. In contrast, *control connections* are not necessary as part of a fully operational network. They do, however, allow one to constrain the operation of a visual program and provide additional control over the order in which processes are executed.

Glyphs contain input and output *control connection nodes* which cause the second, or *controlled* glyph, to "wait" on the execution of the operator represented by the first glyph. Control connections provide a simple way of specifying an order for process execution when one is not already dictated by the data flow, as is frequently the case in networks with a number of parallel paths.

3.2 Variables

Variables can be defined and expressions utilizing those variables can be evaluated within Cantata. Once defined, variables and expressions can be used in place of integer, float, and double arguments of glyphs. Valid expressions may include variables, standard arithmetic operators, and logicals, as well as predefined constants and functions. The use of variables and expressions is required by many control structures.

4 Control Structures

To be a true programming language, a [visual programming] system must include the ability to handle conditionals and iteration.

Control structures allow one to create more complex visual programs by providing constructs that can be used to direct the flow of your visual program. In this context, "flow" refers to both "control flow" and "data flow". When there is more than one possibility for data flow, control structures are used to determine which path the data flow will take.

Control structures fall into two major categories: *looping constructs* and *conditional* *constructs*. Looping constructs cause control flow of the visual program to iterate through a particular section of the network multiple times. Looping constructs depend on the value of variables to determine how many times the control flow will iterate on the specified section of the network. Looping constructs include *while* loops and *count* loops.

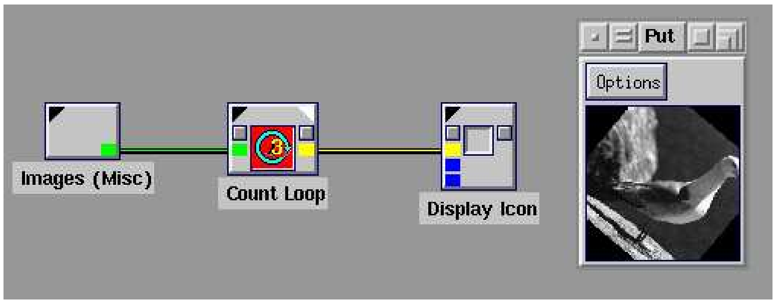

Figure 3: A visual program utilizing a count loop glyph to repeatedly rotate an image.

In contrast to looping constructs, conditional constructs imply that control flow of the visual program will be directed one way or another, depending on the value of variables defined for the workspace. The available conditional constructs include *if/else*, *merge*, *switch*, *trigger*, and *expression*. The values of defined variables may be changed by directly by the user from a variables pane or by an operator from an expression glyph. Looping constructs have built-in mechanisms for variable value update.

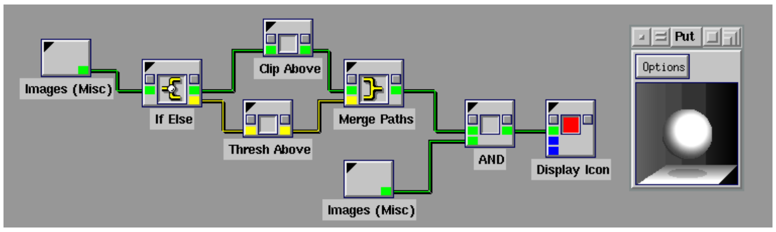

Figure 4: A visual program utilizing control glyphs.

4.1 Procedures

Hierarchy is supported within the Cantata environment in the form of procedures. Similar in concept to a subroutine in a textual programming language, a visual programming procedure allows you to modularize a visual program so that certain operations are confined to a particular location within the visual program. Procedures promote readability of a visual program. In addition, they allow a certain portion of a network that performs a particular function to be used multiple times within a visual program. The use of procedures is especially effective with large and complex workspaces, as well as with workspaces that are required to perform a particular task a number of times.

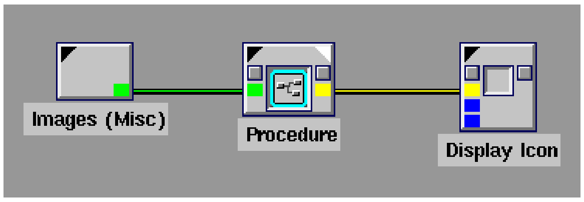

Figure 5: The procedure has been created to contain the "Shrink" and " Rotate" glyphs.

When the procedure workspace is open the main Cantata workspace, displaying the "outer" visual program, will be replaced with the procedure workspace, displaying the network that comprises the procedure. If desired, procedures may be nested.

4.2 Visual Network Scheduling

Cantata interprets the visual network dynamically to schedule glyphs and then dispatch them as processes. The Cantata scheduler is event driven, rather than data driven or demand driven. Once a glyph has been scheduled, the dispatcher is responsible for determining the data transport, the communication protocol, and the process execution mode. The data transport between the glyphs can be permanent (using files or shared memory) or non-permanent (using sockets or streams). The communication protocol between Cantata and the different glyphs can be as simple as initiating process execution, but may be more complicated if glyph parameters must be continuously updated as the process executes continuously. Glyphs may be executed locally or remotely to efficiently utilize a heterogeneous network of computers.

4.3 Encapsulated Workspaces

Once a visual program has been completed, is may be used to create an independent application that can be run without display of the network of glyphs that defines the visual program. The functionality embodied by the visual program may be utilized as desired, independently of the Cantata environment. The resulting program is called an encapsulated workspace. Like any other program in the Khoros system, it can be edited and maintained within the Khoros software development environment. Encapsulated workspaces may be be brought into Cantata as new operators; therefore, visual programmers may create encapsulated workspaces simply to have them accessible as independent programs.

An encapsulated workspace is not composed of either a C program or a scripting language interpretation of the visual program. In fact, the key components of an encapsulated workspace are:

1. The network of glyphs as they appeared in the Cantata workspace when the encapsu-

lated workspace was created, written out as a saved workspace file, just like any other saved workspace file.

2. The user interface description file which simultaneously defines the workspace graphical user interface and the command-line user interface of the encapsulated workspace.

3. An automatically generated shell script which provides the command line user interface to the encapsulated workspace.

Creating an encapsulated workspace from a completed visual program is a convenient way to allow others (who may not be familiar with visual programming) to use the visual program without being concerned, or even aware, of its origin. Because a program created from an encapsulated workspace need not display the graphical user interface of the visual program, it can also be run in "batch" mode; for example, the program could be executed repeatedly by a script that could be run overnight.

5 Software Development Environment

Each program and library in the Khoros system is contained within a *toolbox*. A toolbox is a collection of programs and libraries that are managed as an entity. A toolbox imposes a predefined directory structure on its contents to provide consistency and predictability to software configuration. Typically, a toolbox contains programs and libraries that are characteristic of a given application domain. Imposing a standardized directory structure, configuration files, and a database which holds control data required by Software Services, toolbox objects allow a large amount of software to be configured and managed as a single entity.

Similarly, a *software object* is an encapsulation of the files associated with a particular library or program. These files are organized in a standard directory structure, and a database of configuration management information associates semantics with the collection of files and directories. A software object can be a executable program, a library of functions, or a "pane" object (which provides an alternative user interface to another existing program).

The Khoros software development environment supports the organization depicted above in a way that is designed to reduce the detail and complexity inherent in a large-scale software system. It incorporates support for the software process itself with automatic code generation for application command line user interface (CLUI) and graphical user interface (GUI) with standardized documentation and interactive configuration management.

The Khoros software development environment consists of are three interactive tools which operate on toolboxes and software objects: *craftsman*, the toolbox creation and management tool, *composer*, the software object editor, and *guise*, the direct manipulation GUI design tool.

To customize and extend the scientific capabilities to the specific needs of a particular application domain, these software development tools allow Khoros developers to build and integrate new operators which may then be accessed from the *Cantata* visual programming

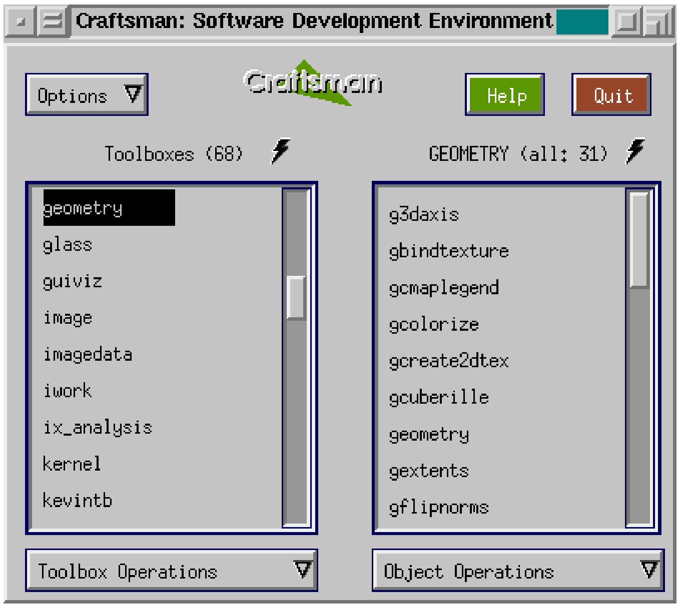

Figure 6: Craftsman is the toolbox and software object management tool that is most commonly used as a front end for the Khoros software development system.

environment. Using GUI & Visualization Services, developers may also create interactive scientific visualization applications. The Khoros software development system is often used for software integration, where existing programs can be brought together into a consistent, standardized, and cohesive environment. By integrating the user-interface code generators, the configuration management software, and the software process into one cohesive system, the software development tools support the developer throughout the software life cycle, from initial prototyping to final delivery [1].

6 Program Services

The Khoros *Program Services* is a set of libraries that are layered to provide the software developer with a variety of application programming interfaces (APIs)that trade off reduced

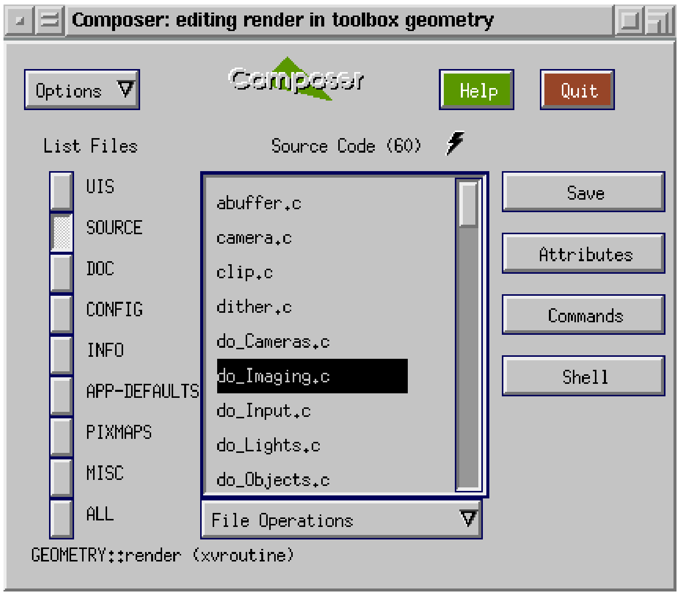

Figure 7: Composer is the software object editor used to edit, manipulate, and compile existing software objects.

complexity against detailed control. Each program service has a name indicating its main purpose in providing functionality to the developer; each program service is made up of one or more distinct libraries. There are three major program services in Khoros:

- **Foundation Services** [2], which provides Khoros with data transports, distributed computing, math utilities, a symbolic expression parser, and portability via a layer of abstraction from the operating system.

- **Data Services** [3], which allows the developer to access and operate on data independent of its file format or its physical characteristics such as size or data type, and is designed to address the needs of a large number of application domains, from image and signal processing to geometry visualization and numerical analysis.

- **GUI And Visualization Services** [4], which provides all capabilities related to graphical display using X Windows, containing all the functions needed to create, interactively change, and maintain a graphical user interface, and offers a variety of visualization capabilities, including image display and manipulation, colormap control, 2D plotting, 3D plotting, surface rendering, and annotations.

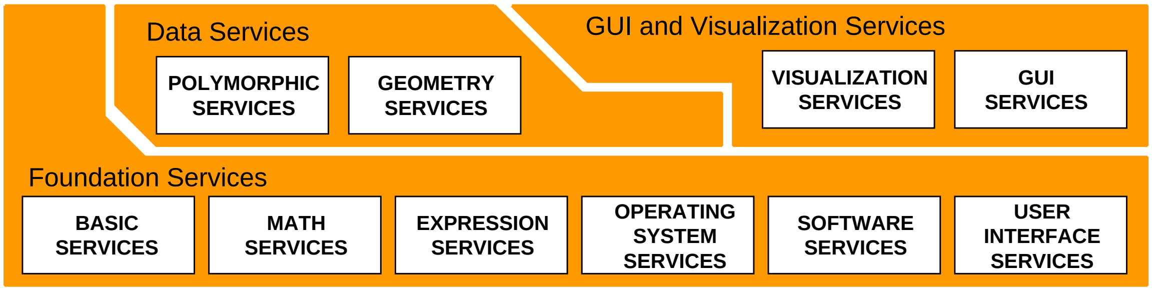

Figure 8: The Khoros software infrastructure is composed of three major program services: Data Services, GUI & Visualization Services, and Foundation Services. These three program services are composed of more specific program services and are layered as shown.

6.1 Foundation Services

Foundation Services is a collective name for six more specific program services, which include: Basic Services, Math Services, Expression Services, Operating System Services, Software Services, and User Interface Services. Each library provides an API that offers a specific functionality set to the application developer.

Perhaps the most important goal of Foundation Services is to provide portability between a large variety of Unix-based workstations running X Windows. An application program developed using these function calls will automatically gain the portability provided by Foundation Services. Similarly, an application program using Foundation Services will transparently obtain the capability to support a variety of data transport mechanisms, including files, mmap, and shared memory. In addition, Foundation Services offers a variety of mathematical functions, a symbolic expression parser, and a number of commonly used functions and utilities.

Basic Services Basic Services offers a broad range of commonly used functions and utilities. Some of the string utilities are more robust, flexible versions of those provided in *libc* with increased error checking, while others provide functionality that goes beyond the offerings of *libc*. Tokenized string utilities allow the programmer to greatly increase efficiency of code. Program information utilities report general information about the program being run, while a standardized message reporting system encourages predefined, standard methods of prompting, reporting errors, giving warnings, and broadcasting general information. Memory allocation utilities standardize the result of zero byte allocation requests, maintaining portability. Other utilities support array creation and manipulation, in a variety of data types. Another group of of convenience functions involve file path and directory path manipulation, and provide routines to expand a path, expand a filename, create a unique name for a file in the temporary directory, among others. All Basic Services are written to ensure portability to a variety of Unix platforms.

Math Services Math Services is a collection of mathematical functions and utilities. Its provides a portability layer on top of the standard C math library *libm*, and a collection of routines that are not ordinarily part of the C math library, but are commonly used in scientific data processing. Math services augments the standard math utilities by offering functions for manipulating single and double precision complex data, generating random numbers and sequences with a variety of distributions, doing operations on matrices, generation of periodic sequences, performing interpolation, converting data types, and performing both scaling and normalization. In addition, math services provides C implementations of some FORTRAN 77 math functions that are not typically included in *libm*.

Expression Services Expression services provides a symbolic mathematical expression parser implemented using Lex and YACC. The expression services library contains routines to set variables, to parse mathematical expressions utilizing variables that have been previously set, and to evaluate these expressions. The expression parser supports a full variety of data types, including byte, unsigned byte, short, unsigned short, integer, unsigned integer, long, unsigned long, float, double, complex, double complex, and string. All mathematical functions defined by Math Services can be evaluated by the expression parser.

Operating System Services Operating system services isolates Khoros from the operating system, and extends the capabilities of the operating system; furthermore, it allows all applications that use it to transparently support distributed computing. Operating Services provides an application programming interface (API) which hides the details of data transport, distributed computing, and process execution. The API is modeled after UNIX function calls and therefore existing applications can be quickly and easily converted over to be distributed.

Data transport refers to the method used to transfer data between processes. Data transport mechanisms can be local or remote. Different transport mechanisms have been abstracted underneath the UNIX file function calls, allowing a single program to transparently handle local tranports such as files, shared memory, pipes, streams, and unix domain sockets and remote transports such as sockets, tli, and rpc, simply by appending a prefix onto the filename. Remote data transport provide the ability to get input from and output to remote machines, making distributed processing possible.

Distributed processing is the ability to specify remote machines on which to execute individual Khoros programs. The capability to do distributed processing is implemented via employment of the remote data transport mechanisms. With distributed processing, one needs a method to execute jobs remotely, as well as a mechanism to transport data back and forth from the remote machine. A remote daemon is started on the remote machine, takes requests to execute a job, and transports data involved with that job using the remote data transport mechanisms.

Specification of data transports and distributed processing directives may be issued from the command line; most often, however, they are used from within the *Cantata* visual programming environment.

Software Services The goal of Software Services is to provide an object-based abstraction for dealing with software components at a level higher than the operating system. While equipped with a public API, Software Services was designed more for internal use by the Khoros software development system than for external use by application developers. Used heavily by *craftsman*, *composer*, *guise*, and *Cantata*, Software services defines the three object classes used by the Khoros software development system: the *toolbox object*, the *software object*, and the *file object*.

User Interface Services User Interface services provides low-level support necessary to maintain the standardized command line user interface (CLUI) of Khoros programs. User Interface services also handles translation of User Interface Specification (UIS) files into program CLUIs, automatic generation of code to support both the CLUI and the graphical user interface (GUI) of a program, and automatic generation of documentation. It has no public API, but is central to the operation of the Khoros software development system.

6.2 Data Services

Data services consists of a collection of libraries that, together, comprise a comprehensive system for accessing and manipulating data. The objective of data services is to provide the application programmer with the ability to access and operate on data independent of its file format or its physical characteristics, such as size or data type. Data services is designed to address the needs of a large number of application domains including image processing, signal processing, geometry visualization, and numerical analysis.

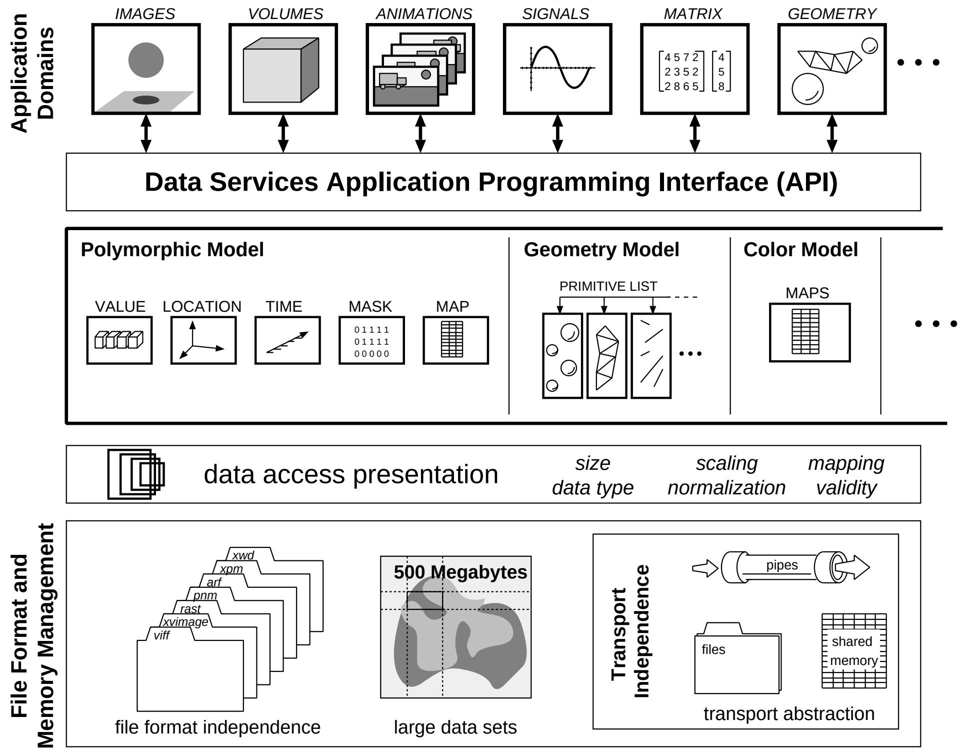

Figure 9: Many diverse application programs can be written to use data services. The polymorphic data model ensures consistent data interpretation across the diverse domains. Data services handles the complexity of data presentation; it also has the ability to deal with large data sets and numerous file formats. Its use of Khoros Operating System Services provides data services with data transport independence.

Using an object-based paradigm, the data services API consists of a set of simple library functions that provide access to an abstract data object. This API allows you to store and retrieve data from the data object and to access characteristics of the data without having to worry about complicated data structures or intricate file handling. This API encapsulates functionality that efficiently handles the tedium of data access and presentation, enabling the Khoros developer to concentrate on the details of specific algorithm implementation without needing to know how to access the data on which the algorithm is operating.

Many different application domains are able to utilize data services. Each domain performs all data access through the data services API. Data is interpreted according to the data model dictated by the domain. Data services has a series of data models available, each of which is designed to meet the needs of a single domain or family of domains. The most versatile of these is the polymorphic data model, (see section 6.4) which provides consistent interpretation of data across many diverse domains. A geometry data model and a color data model are also available.

Data presentation routines, embedded into the data services API, handle casting, padding, resizing, scaling, and normalizing data. Data can be easily presented in whatever form is most convenient. At the lowest level of data services is support for reading and writing several data file formats as well as a memory management system for accessing very large data sets. The entire system is built on the Khoros transport abstraction; data objects can be accessed independent of their underlying transport, whether it be a file, pipe or shared memory.

6.3 GUI & Visualization Services

GUI & Visualization Services provides Khoros with all capabilities related to graphical display using X Windows. The libraries in the GUI and Visualization Services rely on X Windows (X11R4, X11R5, or X11R6), and the X Intrinsics Toolkit. It contains all the functions needed to create, interactively change, and maintain a graphical user interface, and offers a variety of visualization capabilities, including image display and manipulation, colormap control, 2D plotting, 3D plotting, surface rendering, and annotations.

GUI & Visualization Services is layered on top of Foundation and Data Services in order to provide flexibility in its visualization capabilities. For instance, the Image object can read or write a file, and can access that file via conventional disk-based file, a pipe, or a shared memory transport. In addition, it can access that file regardless of its data type or format through the transparent data conversion provided by Data Services.

Like Data Services, GUI & Visualization Services is *object-based*, meaning that it was designed using an object-oriented paradigm. It gives the application developer access to *GUI objects* and *visual objects.* These GUI and visual objects are self maintaining; they refresh and resize themselves, and have *attributes* which the developer may set to control size, location, appearance, and the data to be displayed, in addition to characteristics particular to the object in question. The objects may be made interactive through the installation of event handlers, callbacks, timeouts, and input handlers. Characteristics of GUI and visual objects are controlled by setting and retrieving attributes.

GUI objects act as graphical user interface components. GUI Services provides a large number of GUI objects. Some of the more basic GUI objects include the button, the scrollbar, and the text object, while more specialized GUI objects include the file-keyword browser, the online help object, and GUI objects intended for interactive notification, specifically the error object, the info object, and the warn object. Visual objects, on the other hand, perform scientific visualization of graphics, images, annotations, plots, colormaps, and other types of data. The simpler visual objects include annotations such as lines, circles, and rectangles. Examples of more sophisticated visual objects include the 2D plot object, the 3D plot object, the axis object, the image object, and the animate object.

Some of the features of the visual objects provided by GUI & Visualization Services are listed below.

- The GUI objects of the Khoros Widget Set are designed to have a Motif look and feel and to be Motif-compliant. However, there is no dependence on a third party widget set; this reduces compilation complexity an increases speed.
- The Image object allows the user to roaming through very large image files rapidly due to buffering algorithms provided by the Data Services libraries.
- Interactive extraction of regions of interest (ROIs) are directly supported by the Image object. ROIs may be rectangular, circular, polygonal, linear, or drawn freehand.
- The Image object allows use of image mask files to suppress display of masked out areas.
- The Position and ColorCell objects provide for common user-interaction needs to do precise pointing and querying of individual pixels in an image.
- Display of scaled images is supported by the ImageIcon objects, which can be used to display a miniature version of an image, the PanIcon objects, which allows you to pan around large images, and the Zoom objects, which supports zooming in on an image.

Since these are all subclasses of the Image object, they inherit the capabilities of the Image object.

- Multiple visual objects used in the same application such as the PanIcon, Zoom, and Image objects, can all be set to "look" at the same data object. When that data is updated, all visual objects are automatically updated.

- The Pseudocolor and Threshold objects allow interactive editing of a colormap. When used in conjunction with an Image object that is "looking" at the same data object, interactive modification of the colormap using the Pseudocolor or Threshold object is automatically reflected in the displayed image.

- All visual objects provide support for 8-, 16-, and 24-bit visuals.

- All visual objects provide dynamic resource editing with associated "menuforms". Users can bring up these menuforms to set attributes of visual objects interactively.

- The visual objects are designed to be compatible with one another, allowing you to mix images and graphics. For example, Image objects can be annotated with objects such as lines, rectangles, circles, polygons, text, and pixmaps; child objects inherit the world coordinate system of their parent, which makes positioning convenient.

GUI & Visualization Services also contains *xvlang*, the graphically expressed, data flow visual programming toolkit which provides the visual programming objects used by *Cantata*. It implements an object-oriented approach to the design and implementation of visual programming environments. While the specific operation of the visual programming objects in the *xvlang* visual programming toolkit were strongly influenced by the needs of the data-flow oriented Cantata, the visual programming toolkit itself was designed to be as general and as flexible as possible in order to support creation of new applications that follow the visual programming paradigm [5]

GUI & Visualization Services provides a working environment for the development of scientific visualization applications, addressing many of the issues associated with quickly developing X windows based applications. The layered approach of the Khoros infrastructure, and the concept of program services provide developers with the flexibility to create detailed applications while hiding the intimidating details of system programming and X windows.

6.4 The Polymorphic Data Model

The polymorphic data model is based on the premise that data sets are usually acquired from real-world phenomena or generated to model the same. As such, the polymorphic model consists of data that exists in three-dimensional space and one-dimensional time. You can picture the model most easily as a time-series of volumes in space. This time-series of volumes is represented by five different data segments. Each segment of data has a specific meaning dictating how it should be interpreted. Specifically, these five segments are *value, location, time, mask*, and *map*. All of these segments are optional; a data object may contain any combination of them and still conform to the polymorphic model.

The *value* segment is the primary data segment, consisting of data element vectors organized implicitly into a time-series of volumes. The value data may be given explicit positioning in space and time with the location and time segments. The remaining two segments are provided for convenience. The mask segment is used to mark the validity of each point of value data. The map segment is provided as an extension to the value data; the value data can be used to index into the map data.

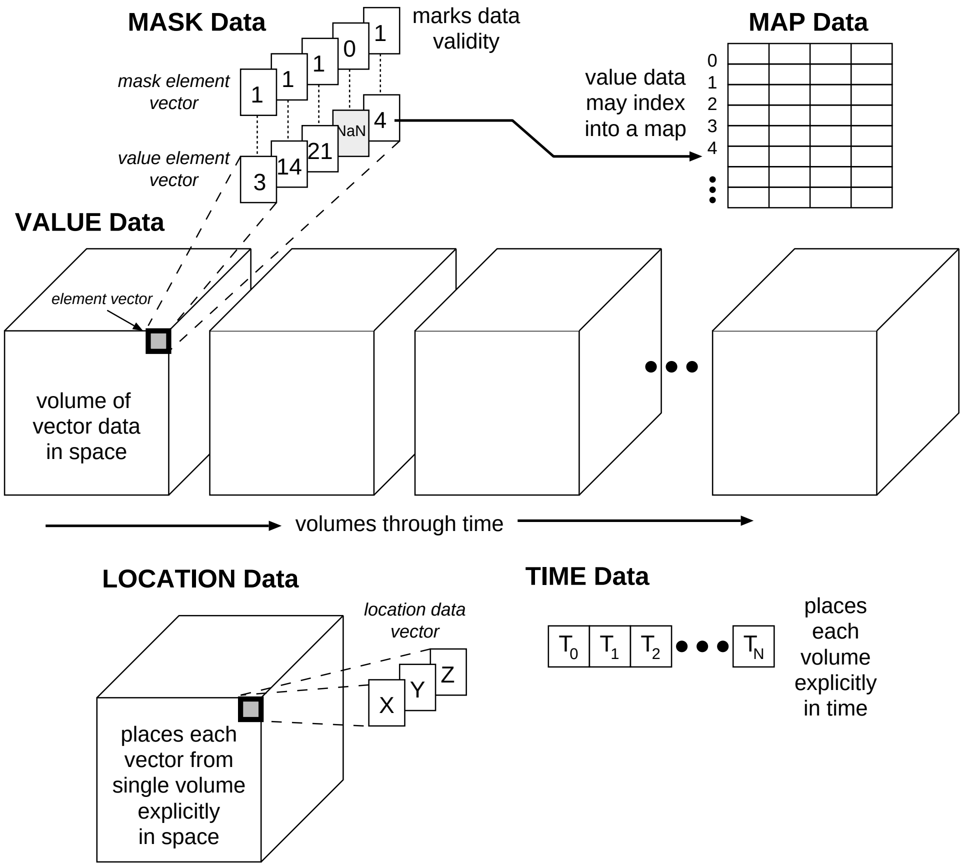

Figure 10: An overview of the Polymorphic Data Model. The polymorphic model consists of five data segments, each segment serving a specific purpose. The *value* segment consists of data element vectors organized into a time-series of volumes. The volume of value data can be given explicit locations in space with the *location* segment; one location vector is provided for each value vector in a single volume. The volumes of value data can be given explicit locations in time with the *time* segment; a time-stamp may be given for each volume in time. A *mask* segment is available for marking value data validity. A *map* segment is also provided; the value data can be used to index into the map data.

6.4.1 Value Data

The *value* data segment is the primary storage segment in the polymorphic data model. Most of the data manipulation routines are specifically geared toward processing the data stored in this segment. In an imaging context, the individual pixel RGB values would be stored here. In a signal context, regularly sampled signal amplitudes would be stored here.

The value segment consists of a time-series of volumes where each volume is composed of element vectors. Each element vector is composed of a number of value points. The size of the value segment is determined by the width, height, and depth of the volume, by the number of volumes through time, and by the number of points in the element vector.

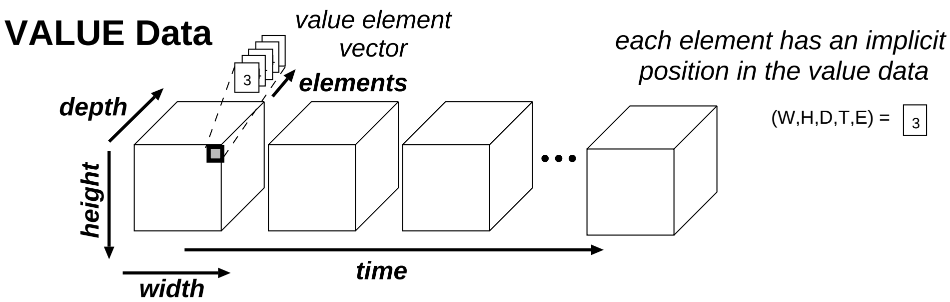

Figure 11: The value segment of the polymorphic model is best pictured as a time-series of volumes. Each volume consists of element vectors oriented implicitly along the width, height, depth, time and elements dimensions. Each element vector can be indexed by a four-dimensional designator while each specific value comprising the element vector can be indexed by a five-dimensional designator.

6.4.2 Location Data

The value element vectors in the value segment are stored implicitly in a regularly gridded fashion. Explicit location information can be added using the *location* segment. If the value data is irregularly sampled in space, the explicit location of each sample can be stored here. Specifically, the information stored in this segment serves to explicitly position the value data in explicit space.

The location segment consists of a volume of location vectors. The width, height, and depth of the volume are identical to the volume size of the value segment. Different location grid types are supported. A curvilinear grid allows for explicit locations to be specified for each vector in the value data. A rectilinear grid allows for explicit locations to be given for the width, height, and depth axes. A uniform grid allows for explicit location corner markers to be specified. Note that the location data only explicitly positions a single volume; the position then holds for each volume through time.

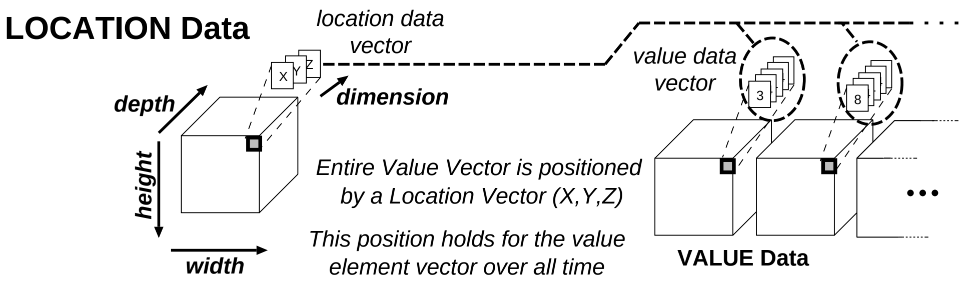

Figure 12: The location segment of the polymorphic model is used to explicitly position the volume vectors in space. The location segment consists of a volume of location vectors; the width, height, and depth, of this volume is shared from the value segment. The location vector is of size *dimension*.

6.4.3 Time Data

Explicit time information can be added using the *time* segment. If each volume of value data is irregularly sampled in time, an explicit timestamp for each volume can be stored here. This is useful in animations where each frame of the animation occurs at a different time.

The time segment consists of a linear array of timestamps. The number of timestamps matches the time size of the value segment.

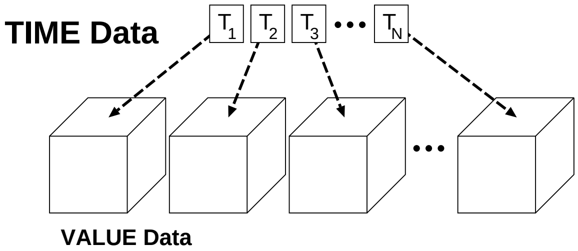

Figure 13: The time segment of the polymorphic model is used to explicitly position the value volumes in time. The location segment consists of a linear array of time stamps; the number of timestamps is equivalent to the *time* size of the value segment.

6.4.4 Mask Data

The *mask* segment is available for flagging invalid values in the value segment. If a processing routine produces values, such as NaN or Infinity, these values can be flagged in the mask data so that later routines can avoid processing them. A mask point of zero is used to mark invalid value points, while a mask point of one is used to mark valid value points.

The mask segment identically mirrors the value segment in size; there is one mask point for each value point. Thus, a value in any given element vector at any given location or time can be marked as invalid.

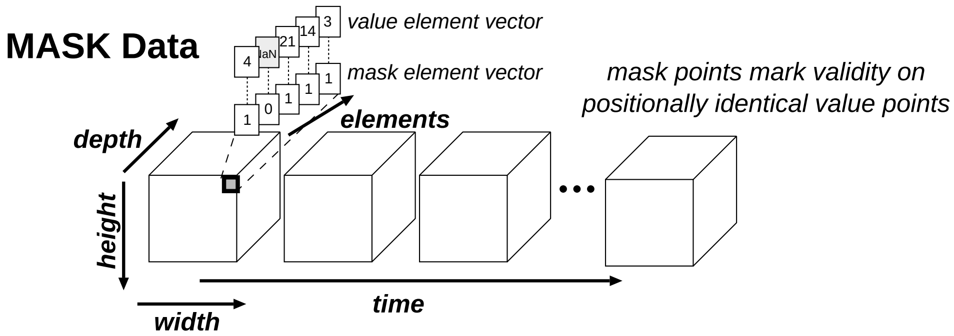

Figure 14: The mask segment of the polymorphic model is used to mark data validity of the value points. The mask segment is exactly the same size as the value segment.

6.4.5 Map Data

In cases where the value data contains redundant vectors that are duplicated in different positions within the volume, the *map* segment may be used. The value vectors are replaced with values which index into the map; the map then contains the actual data vectors. In this sense, the map is an extension of the value segment.

The map segment consists of a number of width-height planes. The values from the value segment map into the map height indices. The map vector runs along the map width. A simple map would consist of just a single width-height plane; a more complicated map would have a width-height plane for every depth, time, and element plane in the value segment. This provides a great deal of mapping flexibility. For example, every plane in a volume or every image in an animation could have a separate map.

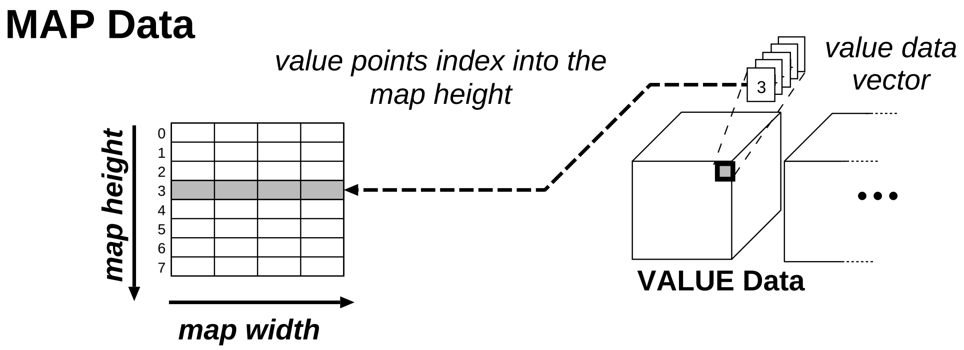

Figure 15: The map segment of the polymorphic model is used store a lookup-table of map vectors. Values in the value segment are then used as indices into the map; the value points map to indices along the map height. The map vector runs along the map width. A number of (map width x map height) planes may exist; the map size may match the depth, time, and element size of the value segment by specifying the appropriate map depth, map time, and map elements.

7 Advanced Khoros

The ongoing development of the Khoros technology is planned through the year 2000 with the Advanced Khoros system, the rapidly changing R&D version of Khoros. The continually evolving Advanced Khoros technology expands its infrastructure and integration technology to advanced systems. The objective of Advanced Khoros is to provide a single integrated environment that supports a systems developer through the life cycle of a software system. This means that, from algorithm development, through prototyping, and finally delivery of an application, Khoros will offer the tools and infrastructure required.

The key development areas of Advanced Khoros are high performance computing, distributed systems, embedded and real time systems, a workspace compiler for the Cantata visual language, support for parallel processing development, and interoperability with CORBA.

Funding and direction for the development of Advanced Khoros comes from a variety of sources, including government contracts, commercial contracts, and the Khoros Consortium. Advanced Khoros is available over the internet to government customers that have helped to fund its development.

Specific project areas now underway within Advanced Khoros include :

- Support of High Performance Computing (HPC) platforms such as the IBM SP2, Origin 2000, and Cray T3E
- Continually increasing support for software development in C++

- Extensions to the visual programming libraries to better support simulation and process control oriented applications within Cantata. One goal of this effort is to adhere to the Department of Defense's emerging standard for simulation systems, High Level Architecture (HLA).

- A workspace compiler which can produce independently running, high performance applications from visual programs.

- Extensions to the data services libraries based on existing message passing technologies which will simplify development of data parallel applications, in the form of a Parallel Data Services.

- Extensions to the data services libraries to support streaming

- A toolbox of parallelized data manipulation routines

- A toolbox of vector image and signal processing routines based on the VSIP (see http://www.vsip.org) standard

- An application environment which will make parallel and distributed programs running on high performance computers more accessible to the average end-user.

- Distributed systems capabilities to support distributed computing and network based distributed collaboration via a CORBA interface.

- Support for visual program prototyping on embedded and real-time systems. This will promote the development of software which not only will be portable across different systems but will also be more easily migrated to the new generations of embedded and real-time hardware as they become available.

- Integration of the visualization libraries with the high performance OpenGL graphics library. This will combine portability for the graphical components of Advanced Khoros with the ability to take advantage of custom graphics hardware if it is available.

7.1 The Khoros Technology and Khoral Research, Inc

The Khoros software project originated at the University of New Mexico in Albuquerque, New Mexico. Khoros 1 was released via anonymous ftp in 1991, and by early 1993, an estimated 10,000 users across the world were actively using the Khoros system. Khoral Research, Inc. (KRI) was founded in 1992 by the group of individuals who were the most involved in the creation of the Khoros system. A technology corporation headquartered in Albuquerque, New Mexico, KRI moved to an off-campus location in March of 1994. KRI continued to evolve the Khoros software over the next four years, releasing Khoros 2.0 and a number of subsequent Advanced Khoros releases over the internet. KRI is now a dynamic, growing software technology company that has tripled in size since its start.

The first commercial version of Khoros was made available on CD-ROM in July of 1996. KRI plans commercial releases based on selected portions of the Advanced Khoros system. The currently available commercial version of Khoros is KhorosPro 2.2, released in July of

1997. In early spring of 1998, a new version of the Khoros technology for commercial users, Khoros Pro 2000, will be released. This version will be compatible with Windows NT via the third-party porting tool, OpenNT.

7.2 Authors

Mark Young, KRI's Chief Scientist/CTO, leads the design of Advanced Khoros infrastructure. Steve Kubica leads the Data Processing group, while Steve Jorgensen leads the Tools group. Danielle Argiro leads the GUI/Visualization group, and manages the KRI R&D group. Jeremy Worley is Vice President of R&D, in charge of government contracts and obtaining new business for the KRI R&D Group.

References

[1] Steven Jorgensen. An object-oriented approach to tool integration in an integrated case environment. Master's thesis, EECE Department, University of New Mexico, 1994.

[2] Khoral Research, Inc. *Khoros Manuals: Programming Services Volume I*, 1994.

[3] Khoral Research, Inc. *Khoros Manuals: Programming Services Volume II*, 1994.

[4] Khoral Research, Inc. *Khoros Manuals: Programming Services Volume III*, 1994.

[5] Mark Young, Danielle Argiro, and Jeremy Worley. An object oriented visual programming language toolkit. *Computer Graphics, A Publication of the ACM Siggraph*, May 1995.