HDVS:
A Multi-Input Heterogenous Data Visualization System

James Robert Blakely
B.S. Whitworth College 1988
M.S. University of New Mexico, 1993

A thesis submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science and Engineering

July 1996

© Copyright 1996 by James Robert Blakely
All Rights Reserved

The thesis "HDVS:
A Multi-Input Heterogenous Data Visualization System" by James Robert Blakely has been examined and approved by the following Examination Committee:

David G. Novick
Associate Professor
Thesis Research Adviser

James Hook
Associate Professor

Jim A. Larson

Acknowledgements

I would like to thank David Novick for advising me on a project that held much more interest for myself than it did him. I would also like to thank Ken Maupin, for without his editing and support this thesis would have taken much longer to complete and would not have been nearly as good.

Contents

List of Figures

Abstract

HDVS:
A Multi-Input Heterogenous Data
Visualization System

James Robert Blakely, M.S.
Oregon Graduate Institute of Science & Technology, 1996

Supervising Professor: David G. Novick

Data visualization is the use of visual techniques to represent different types of data. While specific tools are available for visualizing and animating data, there are limitations associated with many of them, the most notable of which is the close integration of the data-generation and visualization aspects into one system. This thesis describes the Heterogeneous Data Visualization System(HDVS), a next-generation data visualization system that separates these two fundamental components and supports heterogeneous data types, multiple data inputs and interactive viewing windows, resource distribution, and hardware independence. With HDVS, users overcome the limitations of integrated visualization systems because the data generation has been decoupled from the data visualization system. This separation enables greater flexibility in visualizing data, as well as a system that is easier to extend and modify.

Chapter 1

Motivation

Data visualization is the use of visual techniques to represent different types of data [23]. The data can take any form but are most often numeric based. The method of visualization can range from a simple bar chart to a complex multi-dimensional animation. Data visualization enables one to absorb information from large data sets easier and faster than one would be able to by analyzing raw data. Data visualization has existed for hundreds of years as a way of coping with the complexity of the world around us.

The widespread availability of high-performance computers has greatly stimulated interest in data visualization. The average user now has access to computers that are capable of displaying and animating large data sets, and computers themselves are capable of generating huge data sets that can only be understood by data visualization. Not only can one gain more information by viewing data graphically one can also see trends and tendencies that would be obscured by the sheer bulk of data that can be generated.

While one can represent data visually in two dimensions, one can add more expressive power by moving to a 3D representation. It has been found that the number of objects or relationships that can be assimilated increases dramatically when moving from 2D to 3D [18]. With the large data sets that can be generated today it is often necessary to make use of 3D models. Furthermore, by utilizing animation one can convey the time evolution of data and increase the information carrying capabilities still further.

While specific tools are available for visualizing and animating data, there are limitations associated with many of them. Low-level graphics libraries are the most flexible but require extensive knowledge of the actual libraries as well as 3D rendering techniques. Many of the higher-level tools are very specific, providing support for only certain forms of data. Some systems limit the visualization to only the static display of data while others require the power of expensive graphical workstations.

There appears to be a need for a simple method of displaying and animating heterogeneous data sets for the average user [6]. Such a system would be able to animate data output by existing programs as well as data that have been stored in files. The performance of the system should enable the real-time animation of many different types of data using high-end PCs or low-end computer workstations. In addition, there is a need for a system that could display and animate data from many sources simultaneously. The system would provide a mechanism for viewing multiple data sources residing on different machines, and enable multiple users on networked machines to view and interact with the same data. The system should be simple to use and widely available. Finally, to be useful to a large number of people, the system should also be easily portable.

Portability can be achieved by choosing an operating environment that is available to many users. The X Window environment [17], is available on virtually every UNIX workstation and many high-end PCs; as such, it is an obvious candidate for the operating environment of a data visualization system. Many current data visualization systems are built on top of the X Window System, illustrating its suitability to this task. There are, however, some limitations that come with the X Window System. Though X supports various drawing primitives and advanced rendering techniques (in the PEX extension [11]), these are often too slow to be used for animation. This may require the user to build a rendering engine from scratch. Another problem with the system is its complexity. Programming at the Xlib level is the graphics analog of assembly-language programming. While the higher-level toolkits such as OSF Motif [9], require the user to write fewer lines of code to implement a given task, it still difficult to learn. Most importantly, these toolkits are designed for the construction of GUIs, not data visualizers. The new OpenGL [2] standard does address the issue of rendering support and is available on X and non-X workstations but is not universally available, requires hardware support for reasonable performance, and still requires a large time investment to learn and use. Other off-the-shelf systems either display data only statically or require certain data formats and specific visual representations. While the X Window System provides a good base for a visualization system, there is a need to fill the gap between the low-level graphics libraries and the higher-level tools available.

In particular, a visualization system is needed that enables the user to configure easily the display and animation of heterogeneous types of data [16]. Ideally this system would be able to display many types of data with no restriction on the data format, enabling the data or program to have their data visualized without modification. In addition, the ability to integrate and display multiple data sets on multiple host computers would add a new aspect to the role of data visualization. Providing a flexible system would grant the power of data visualization without requiring extensive graphics experience, low-level programming, or code modification. The motivation for building a new data visualization system is to provide a simple method for displaying and animating many types of data, filling a void in the current realm of data visualization.

Overview of This Thesis

This thesis is organized as follows:

- **Chapter 2, Related Work** discusses the evolution of data visualization systems, establishes criteria for successful systems, and measures existing data visualization systems against these criteria.

- **Chapter 3, System Goals** proposes goals for a data visualization system which addresses limitations in existing systems and introduces the Heterogeneous Data Visualization System (HDVS), a new system which attempts to satisfy these goals. Notable features of the HDVS include the ability to display heterogeneous data types, support for multiple data inputs and outputs, resource distribution, hardware independence, and a flexible user interface.

- **Chapter 4, The HDVS System** provides a detailed description of the HDVS architecture and user interface, and identifies practical applications.

- **Chapter 5, Discussion** evaluates how successful the HDVS implementation is in meeting the design goals and discusses possible directions of growth for HDVS.

Chapter 2

Related Work

The purpose of data visualization is to provide a method that will enable users to understand the underlying nature of the data better. The power of data visualization is in the transformation of data sets to a graphical representation. This graphical representation often enables the identification of trends normally hidden by the sheer bulk of data [3].

In the past, data visualization was limited to those having access to state-of-the-art graphical workstations and the support of a graphics expert. Today, the computational power of a typical workstation and the graphical tools available make data visualization possible for the average user.

The act of data visualization is the transformation of numbers to some geometrical object representation. This geometry of objects forms the basis for computer animation and visualization. Many models exist for translating data to a graphical representation. The graphical and decomposition models are two geometrical models often used in computer visualization [4]. The graphical model consists of objects that are built of points, lines and polygons. Decomposition models have objects that are constructed from other objects, where all "higher level" objects are ultimately created from a set of base objects. Depending on the model chosen, different data structures are used. For example, graphical models generally include information on vertices and polygons. The graphical model has some implementation advantages because most rendering systems ultimately draw objects as a series of lines and polygons. This enables the data structures being used to represent an object to also be used to render the object.

Animation and object characteristic modification are two techniques that enhance the usefulness of data visualization by attempting to increase or clarify the information being represented. Animation enhances data understanding by illuminating the time evolution of a data set. Hidden trends and subtle nuances are often uncovered during animation. Animation techniques are used extensively in scientific visualization because the objects or environments being modeled often change with time [16]. Animation is also useful in providing a mechanism for movement around and through graphical data sets. The ability to navigate through data and focus on particular areas of interest can be quite useful in isolating critical areas when viewing large data sets [19].

Object characteristics can also enhance the usefulness of data visualization. Characteristics of the graphical objects such as color, texture and shininess can be altered enabling different materials and parameters to be represented [8]. These new techniques in data visualization provide more realism by enabling simulations to closer match the phenomena they represent.

2.1 Evolution of Data Visualization Systems

The evolution of data visualization primarily occurred in two areas. First, there was a continual move toward the separation of the section of the program generating the data from the section of the program responsible for displaying the data. Secondly, the graphics became more sophisticated and easier to employ as higher level graphical tools and systems became available.

For greater clarity, it is useful to introduce new terminology regarding the actual computer code that is used in visualization systems. The section of the computer program that generates the data to be visualized will be referred to as the *data generation code* (DGC). The section of the code that is responsible for the graphical visualization of the data will be referred to as the *visualization code* (VC).

The system evolution can be roughly separated into four stages:

Phase One: DGC and VC contained in the same program. Graphical primitives were the responsibility of the programmer.

Phase Two: DGC and VC still contained in the same program. Primitive libraries available for graphics.

Phase There: Some separation of DGC and VC. High-level tools used for advanced rendering.

Phase Four: Complete separation of DGC and VC. Flexible methods for inputting data.

Each phase will now be examined in greater detail.

2.1.1 Phase One

In phase One, the DGC and VC were contained in the same program. Building a system to display data involved building and integrating the display routines into the DGC. Graphical libraries were not available, requiring the programmer to have significant knowledge of graphical principles or to enlist the aide of someone who did. Because of the limitation of computer resources and the time involved in implementing the desired graphics, the end displays were usually static and quite simple [24].

2.1.2 Phase Two

In Phase Two, the task of visualizing data was somewhat easier with the utilization of graphical libraries. The graphical libraries contained fundamental procedures for rendering scenes. The user decided how data were to be displayed, and modified the source code by adding the necessary drawing routines. The program was then compiled and linked to the graphical libraries. If the image was not satisfactory, it would be necessary to go back to the source code and begin the process again [24]. Graphical libraries enabled more complex views to be rendered but still required some knowledge of graphical principles to implement and were often difficult to use. Also, different programs that visualized similar data were often unable to share rendering engines, thus leading to additional development effort and complexity.

2.1.3 Phase Three

In Phase Three, visualizing data became easier with the development of high-level graphics tools and primitive visualization systems. Graphical tools enabled the user to provide sophisticated rendering without requiring significant graphics experience. While the tools still required a significant time investment to learn, they gave the user power to display data in complex ways. The partial separation of the graphical system from the rest of the program enabled the user to modify the data-generating aspects of the program (DGC), without affecting the VC significantly. Visualization systems also provided a means of displaying data though there were some limitations. These systems were only useful if it was possible to modify the DGC to transform the data into a form that the visualization systems could interpret [6].

2.1.4 Phase Four

In Phase Four, the separation of the DGC and VC is more pronounced. There exists a means of visualizing data without modifying the DGC. These systems use the output of programs and display it as it is being generated [18]. The separation provides more flexibility and ease of use because the method of displaying the data is no longer linked with the method of generating the data. Data can be in multiple formats and can be input from files or from executing programs. Because of the separation of the DGC it is now possible to view multiple sources of data simultaneously.

2.1.5 Summary

Section 2.2 will examine several visualization systems that are currently available. It will be shown that most of these systems reside in the Phase Three state of development, with a select few bridging the gap to Phase Four.

The methods described above are general techniques used in data visualization. Different visualization systems use many of these ideas, though some implemented them better than others. For a data visualization system to be generally useful it should have different rendering schemes and algorithms and be portable and flexible. Having multiple rendering methods enable the display of data representing different phenomena. Portability enables more people to use the system. A flexible system is valuable because it enables the user to visualize data that are in different formats [3].

It is not enough, however, for a visualization system to provide the desired functions and features; it must also be usable. The complexity often found in many visualization systems may limit the ease of use due to the significant amount of knowledge that the user must have of the environment and the data. While many of the tools available today are helpful, they often cannot be used without expert assistance. Users of visualization systems are most often assumed to have knowledge of data management, computer graphics and visual perception. Additional research is needed to give scientists interactive tools they can use to combine, explore, and visualize data [21].

Many factors influence how and when people use data visualization systems. Users have widely disparate levels of ability and different needs. Some prefer a point-and-click interface, while others desire a high level programming language [16]. Whatever functionality the system provides, a number of elements are required for a system is to be useful. To follow the "golden rules" of visualization, a system should:

1. have software that is easy to learn and use,
2. provide generic methods to get data into and out of the system,
3. provide a complete set of functions and features,
4. be open so it can be modified and extended,
5. work on available computers,
6. deliver high graphics speed,
7. be affordable and,
8. provide some method or means for user support [16].

2.2 Examples of Visualization Systems

The capabilities and limitations of visualization systems can be understood more comprehensively through assessment of visualization implementations with respect to the functionality goals outlined above.

2.2.1 Wavefront Data Visualizer

The Wavefront Technology's system [25] was designed as a general-purpose visualization system. The functionality of the system is based on a set of high-level graphical tools. The tools are used to manipulate the graphical representation of the data. Some examples of the tools available in their system are the iso-surface, the cutting-plane, and the data probe tools. The tools can be grouped together in different ways to achieve the desired effect and provide an easy, intuitive method to visualize and interactively view the data. An example of tool grouping would be to group three cutting-plane tools oriented at right angles, which could then be moved together to generate a cut-away view of the data. The system is limited to two standard data formats but new file readers can be written.

This system meets many of the requirements for good visualization software. It is extensible and provides a flexible means for generating different types of graphical constructs but requires data to be in specific formats. The system is also limited to a single viewer and data can not be input from multiple sources simultaneously.

2.2.2 Vista

The Vista system [22] was developed to enable one to do dynamic selection of data for graphical display as the DGC is executed on a remote system. The system was developed partly in response to the problems encountered using the standard method of incorporating graphics libraries into existing code. The library method is quite limiting because the data to be viewed must be known when the code is written. In contrast, the Vista system can be used to steer calculations based on the intermediate results of the computation. intermediate. This interactive approach to computing is called simulation-time execution [13]. The integration of scientific computing and visualization is useful to reduce turn-around time in performing calculations.

The Vista design emphasized the following requirements. The system should have easy access to graphics. The modification of the program should be minimal and straight forward. There should be a dynamic method to determine what and when the data are to be displayed [22].

To use the visualization system, the user inserted visualization break points into the data generation code (DGC). The program was then compiled with a standard C compiler using the -g option to obtain use of the symbol table the compiler generated. When the application was executed it set up a connection with the visualization server. Variables of interest could be displayed while the program executed.

Although the Vista system satisfied many of the requirements for a good data visualization system it does have limitations. There was no easy interface method available and its use required access to and modification of the source code. While it did separate the DGC and VC somewhat the system could not utilize multiple and simultaneous sources of input. The system wasn't capable of operation in a distributed environment. The Vista system is a Phase Three system.

2.2.3 Application Visual System (AVS)

AVS [10] is a commercial data visualization product that runs on all major UNIX workstations. AVS supports 3D plots and graphs as well as advanced tools such as 3D interactive rendering and volume visualization. The AVS uses a visual programming environment where the user connects AVS modules together to build a visualization network. The network becomes an application that can be easily reused and modified. AVS provides different pre-built modules and a mechanism for the user to create new modules. The modules can include C, C++ and Fortran routines, and can also be linked with external programs. AVS provides a powerful visualization tool set, including a data viewer, a Geometry viewer and an Image viewer. AVS can be used as a visualization system or as an application builder.

AVS provides commercial-level graphics support but at a high monetary cost. This system cannot utilize simultaneous multiple sources of input and does not provide multi-host multi-viewer functionality. The AVS is a Phase Three system.

2.2.4 Polka Animation System

The Polka Animation System [20] is a general-purpose visualization system capable of animating programs. Polka is designed to balance power and ease of use. Polka is implemented in C++ and runs under the X Window system. Though Polka is freely distributed, it requires the Motif toolkit, which is not free. Polka can be used in two different ways: by using the C++ class library or the Samba front end to integrate graphics into existing programs. The C++ library is useful but requires the user to integrate the DGC and VC. Samba relaxes this constraint somewhat by providing an interpreter that reads ASCII commands and then performs the specified graphical request. A typical Samba command is of the form "command name - command parameters." For example, "move 3 5.2 8.2" is a command that moves object 3 to position 5.2,8.2. While programs must be modified to generate output that Samba can interpret, the simple format of the command language makes this a fairly straightforward task.

The main limitations of Polka involve its interface. The C++ library requires a heavy integration between the DGC and VC. Though Samba improves the data and viewing system dependences, it still requires the DGC to be modified. Because of this Polka is a Phase Three system. The system is also not capable of running in a distributed mode allowing multiple DGCs to simultaneous have their data visualized.

2.2.5 Geomview

The Geomview [5] system is capable of interactively viewing and manipulating graphical/geometrical objects. It is freely distributed and runs on most workstations that use the X Window system. Geomview can be used in two different ways: 1) as a stand-alone object viewer or 2) as a display engine for programs generating data dynamically. The object viewer can display and manipulate objects represented in many different file formats. The object viewer provides a wide variety of predefined objects and the user can define new ones as well.

Of greater interest, though, is the Geomview display engine which can display the data from a program as it executes. Programs that interact with Geomview in this way are called external modules. External modules act as an interface between the data-generation program and the visualization engine via a series of Geomview library calls. Programs must be altered to output a series of commands that the display engine can understand. The format of these commands is provide by a Geomview control language (GCL). The communication between the module and Geomview is provided by UNIX pipes. This allows commands from the DGC on standard output to be input to Geomview on standard input. The module informs Geomview of changes in the object's attributes by outputting the appropriate command.

Geomview is a powerful and useful system, but it does have some significant limitations. Because Geomview communicates via pipes, the module can not receive any input from other sources on standard input. Pipes also impose the limitation that Geomview and the module must run on the same computer. Another drawback of the system is its inability to support multiple data sources and multiple viewers. Geomview provides a useful tool for visualizing data but does suffer somewhat because of loose integration of the DGC and VC. Geomview bridges the gap between Phase Three and Phase Four systems.

2.2.6 Visualization Toolkit (Vtk)

The Visualization Toolkit (Vtk) [19] is a 3d visualization system with a focus toward application building. Though Vtk is implemented in C++, applications can be written in C++ or Tcl [15], an interpreted scripting language. Vtk provides a higher-level graphical command language than PEX or OpenGL and is available on most UNIX platforms. Vtk can be used to visualize data or to build data visualization applications. It provides a robust rendering engine with vector, scalar, and tensor visualization capability. It also supports multiple data types such as polygonal data (points, lines, polygons) and images (structured data sets).

A program visualizing data uses Vtk to control the rendering aspects and must either import or create the data to be visualized. Creating a window and initializing the rendering system is straightforward, but mapping the data to the rendering system is more complicated. The data must be mapped to a form that Vtk can render using a set of data filters that Vtk provides. The data filters are hooked together to perform whatever mapping is needed. Each data filter can perform a specific task such as sampling the actual data, color mapping based on some aspect of the data, or mapping the data to the graphics system. The user is responsible for the overall rendering pipeline. Writing a program in Tcl is done much the same way.

While Vtk provides a robust and flexible system capable of advanced rendering, there are several limitations it imposes on the user. Vtk is more of an application development library than a stand-alone data visualization system. The user must ultimately write an application which utilizes the Vtk library to visualize the data. In addition, Vtk does not provide any methods for integrating multiple input sources. The system provides a higher level of graphics than other libraries, but it is still a library, requiring significant involvement from the user. Vtk provides some separation between DGC and VC but is ultimately a Phase Three system.

2.3 Summary

Many of the visualization systems available today are quite useful, but fall short of satisfying some visualization needs. Much of the emphasis in these systems has been on representing specific types of data, rather than accepting heterogeneous data. In addition, these systems do not satisfy all of the "golden rules" presented in section 2.2. The first two rules, ease of use and generic data input and output methods, seemed to be violated frequently. The systems that came closest to meeting all of the requirements were commercial systems and, as such, violated the rules on being open and affordable. None of the systems reviewed were in the fourth phase of evolution, though the Geomview and Vtk systems had some success in separating data generation from data visualization. In light of these disparities, profound need exists for a heterogeneous, Phase Four data visualization system.

A heterogeneous system would not only provide support for elementary scientific visualization, but also for other types of data. The system would be able to display and animate data from existing compiled programs without requiring rewriting or recompilation. Furthermore, these programs could be scientific in nature or simply typical UNIX commands. In addition, the system would be able to simultaneously visualize multiple data input sources residing on different computers. While the system would not be as powerful as a full-fledged scientific visualization system for viewing scientific data, it would compensate with its versatility. The system would attempt to satisfy the eight golden rules for visualization, while targeting a more general type of data. Finally, the system would completely remove the tightly-coupled integration of the DGC and VC, placing it in Phase Four.

Chapter 3

System Goals

Providing a simple method for the displaying and animation of heterogeneous types of data is the main goal of the HDVS system. This will enable the user to visualize different types of data, from UNIX commands to scientific data, without having to modify the program generating the data. HDVS will provide a communication manager enabling the system to be distributed for better performance on large data sets and CPU-intensive data-generation programs. In addition, the communication manager will provide a means for simultaneously animating data generated on different machines in the same viewing window. HDVS will require no special purpose-graphics hardware, thus enabling wide portability. The viewing system will be built on top of a 3D engine, and will support a diverse set of graphical objects and reasonable animation speed, providing more freedom in the types of data that can be visualized.

These system goals – heterogeneous data input, a distribution of functionality across networked hosts, multiple data input sources, and graphics hardware independence – impose several constraints on the HDVS system:

- **Heterogeneous Data Input**

  The broad spectrum of allowable data makes some form of data parsing necessary. There must be methods available to map data into actions, such as modifying graphical objects or configuring the visualization system. The system will support data translation by providing a convenient means to specify rules for parsing and mapping. The system configuration will not require re-compilation of the DGC. HDVS will provide many levels of configure ability, the choice of which will be dependent on the needs and expertise of the user.

- **Distribution of Computation**

  HDVS is limited due to its distributed nature, because there is some overhead in communication between the parts of the system: it will run slightly slower when the entire system is executed on one machine. Running the system on multiple machines will provide performance enhancements only if the network bandwidth is acceptable. Multiple input sources may also cause difficulties due to network contention and timing.

- **Hardware Independence**

  Finally, the lack of a minimum hardware requirement for graphical support places a limit on the number of graphical objects that can be displayed and on the overall animation speed of the system.

3.1 Heterogeneous Data Types

The visualization system will be able to read data from files or from the standard-input interface. This will enable the system to display data from existing programs without requiring users to modify their code. The incoming data will not have to be of any specific format and in fact can be numeric, text, or a combination of both. Heterogeneous data-type support is made possible by the filtering engine; the filtering step will perform any necessary translation and mapping of data. Configuration of the filter will involve modifying a template or writing a script. The filtering will occur at run time and will not require re-compilation of the DGC or the visualization-engine system code.

3.2 Multiple Data Inputs and Outputs

HDVS will provide support for data being generated by multiple sources on different computers to be integrated, animated, and viewed simultaneously as one visualization scene. HDVS will allow data sources to be interactively connected, visualized, and disconnected at any time during the execution of the visualization system. In addition, HDVS will enable different users on different machines to connect and to view the same visualization scenario at the same time.

3.3 Distribution of Computation

HDVS will consist of at least three separate modules: one or more filters, one or more visualization engines, and one communication manager. Each can execute on a different machine. This will allow the programs generating the data and their corresponding data preprocessing filters to be on different machines, and the actual visualization system to run on still another. This will yield enhanced performance in a networked environment (assuming a reasonable network load). The ability to have many data-filter pairs will provide a flexible and novel method for viewing multiple data sources simultaneously. Performance improvements versus a non-distributed system will be greater if the method of data generation is CPU-intensive.

3.4 Communication Protocol

HDVS will provide a communication protocol and manager to support communication between the filtering programs and the visualization engines. This protocol will support bidirectional message passing between the filters and visualization engines. Messages will be able to create and manipulate graphical objects, and configure parameters of the visualization engines. The protocol will also be extensible to allow for the creation of new message types. This will allow the system to incorporate new functionality without having to develop a new method of communication. The communication manager will determine where messages are to be sent and perform whatever mapping must be done on the messages before they are forwarded to their final destination.

3.5 3D Rendering Engine

HDVS will have a 3D rendering engine that is capable of animating polygon objects, lines, and text. The engine will support multiple light sources, Gouraud-shaded objects, and a dynamic viewing position. Gouraud shading [1] is a simple lighting model which interpolates colors across the face of a polygon, reducing intensity discontinuities at the interfaces between polygons. Gouraud-shaded polygons that compose an object blend together, heightening the object's realism. The visualization engine will be built on top of the X Window System. This will enable HDVS to be ported easily to other platforms. The system will require no graphical hardware support and be reasonably fast for modest data sizes. The system will initially support the following objects:

- 3D surface plots with contour colors;
- 3D histogrid plot;
- a variety of 3D base shapes including cubes, spheres, pyramids

The visualization engine will be extensible, enabling the addition of other graphical objects such as volume rendering, texture mapping and/or other extensions. HDVS will also enable multiple viewing windows on different machines. The different visualization engines will each contain the same scene but the individual users can orient their view direction autonomously.

3.6 Flexible User Interface

HDVS will provide various methods for the user to input and filter data. Input methods will include reading data from files and piping data from executing programs directly into the system. This makes possible real-time animation of data generating programs. Data-translation methods (filtering) for the system will include a set of utility programs and a scripting language for ease of configuration and programming. The utility programs will be used for common types of data. Some examples of these might be routines for displaying typical UNIX commands such as `top(1)` and `ls(1)`, as well as routines for data sets that require no special parsing. The front end to the system will enable the use of scripts that can then be executed to filter the desired data. The scripting language will have normal control-flow statements such as if and while, as well as special-purpose commands to display the data. Finally, HDVS will provide a C Application Programming

Interface (API), making possible the addition of new display methods to the set of utility functions.

3.7 Other Systems

Below is a short summary on how the systems presented in section 2.2 achieve the above goals.

- **Heterogeneous Data Types**

  Geomview and Vtk were the only visualization systems that provided any mechanism for heterogeneous data types. Geomview achieved this capability with external modules which could perform any translation needed to put the data in a form Geomview could understand. Vtk made the use of heterogeneous types of data somewhat possible with a Tcl-based front-end scripting language.

- **Multiple Data Inputs and Outputs**

  None of the systems reviewed were capable of integrating and animating data that was generated by multiple sources on different computers. Similarly, none of the systems were capable of displaying the data being visualized on multiple computers simultaneously.

- **Resource Distribution**

  Wavefront Technology's system was the only system capable of running in a distributed mode.

- **3D Rendering Engine**

  All of the systems that were reviewed provided a 3D rendering engine. The Wavefront, AVS and Geomview systems provided the most robust display engines.

- **Flexible User Interface**

  The Vtk and Geomview systems were the only systems to provide a flexible user interface. Geomview achieved this by providing external modules that acted as an interface between the DGC and the VC. Vtk provided a flexible user interface via its Tcl front end but was more of a application development library than a data visualization system.

While many of the reviewed systems meet some of the above criteria none of them met all of the specified system goals.

Chapter 4

The HDVS System

The Heterogeneous Data Visualization System is a distributed data visualization system that is capable of displaying and animating many different types of data. HDVS can be used with data sets that are purely numeric, numeric mixed with text, and text-only.

HDVS actually integrates three components: a filtering program, a visualization engine and a communication manager.

- Filter

  The filtering program enables the user to specify what actions should be taken on incoming data. Possible actions include parsing complex data, removing unwanted data, scaling data, and most importantly, determining what graphical object is to be used in representing data. The filter is configured using a template file, which contains much of the basic startup information, including which interface is being used. The template also permits the user to select the method through which data is to be input to the filter. Input can be from a file or piped directly into the program. Allowing dynamic data input makes it possible for programs to have their output data visualized as they execute.

- Visualization Engine

  The visualization engine is a 3D renderer that is capable of displaying and animating polygon-based objects. The engine requires no special hardware to operate and provides reasonable performance for medium-size data sets, where a medium-sized data set is define as one that can be graphically represented by 10,000 polygons or fewer. The engine supports Gouraud shading, multiple light sources and dynamic movement of the view point through either mouse or command interface.

- Communication Manager

  The communication manager acts as an intermediary between the filter and visualization clients. The communication manager makes it possible to have multiple filters and visualization engines connected together simultaneously. The communication manager is responsible for managing the connections for the clients and providing any mappings necessary among filter and visualization clients.

Messages form the basis of communication among the different elements of the system. Messages are passed between the filters and the visualization engines using the communication manager as a staging area. The format of the messages is governed by the Spinning Polygon Protocol. The Spinning Polygon Protocol or SPP was developed for the HDVS and provides the functionality for all communication between the filter and visualization engine, and is explained in greater detail in section 4.3.2. SPP supports the creation, destruction and alteration of graphical objects. One can also use SPP to change various environment variables of the visualization engine. The separation of the filter, visualization clients, and communication manager into separate programs enables the system to be executed on different computers, thereby increasing performance. The only conditions necessary for system speedup are 1) a reasonable network load and 2) the computers incorporated should not be significantly slower than the fastest system available.

Inputing data to a filter client involves three issues: how data are input, how data are parsed, and how data are subsequently mapped into commands for the visualization engine. There are two primary methods for interfacing with HDVS. The first method makes use of a Tcl front end to interact with the filter. The user writes Tcl scripts, using a special set of Tcl commands for communicating with HDVS. The additional Tcl command set includes commands for creating and modifying graphical objects, and modifying visualization engine parameters. The second option uses the utility engine, which contains a set of predefined commands for visualizing specific formats of data, as discussed in greater detail in section 4.3.4.

The basic architecture of HDVS is shown in Figure 4.1.

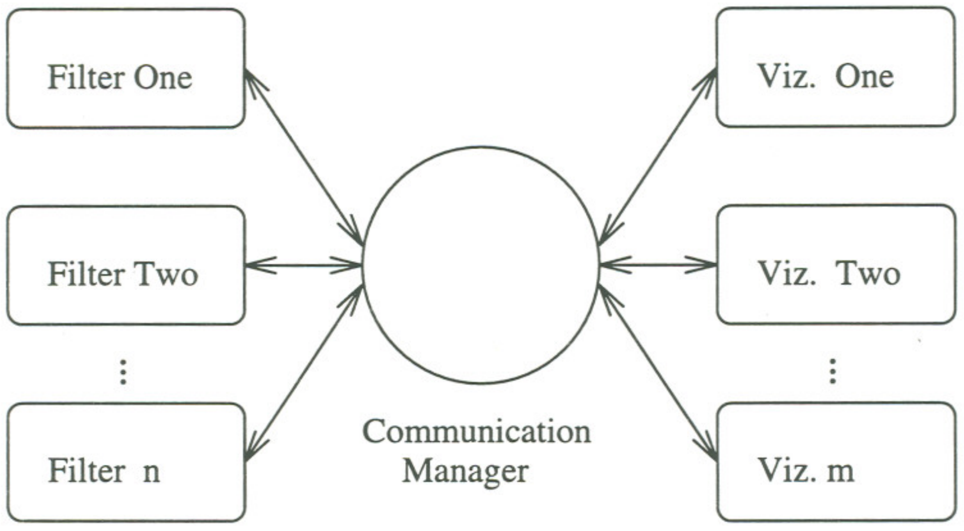

Figure 4.1: Simple View of HDVS Architecture

The boxes on the left side of the diagram represent filter clients. Likewise, the boxes on the right side of the diagram represent visualization clients. Filter clients are not restricted to executing on the same machine as the visualizers and other filters, nor must visualization clients execute on the same machine. The different visualization clients will all display the same data set though the orientation of the data and the user's viewpoint can be different. The lines between the communication manager and the two client types represent data communication pathways. The data pathways may be uni or bi-directional.

The filter will employ one of the two interface methods. Regardless of the chosen interface, data are either read from a file or piped directly into a filter client. Both interfaces communicate with the visualization clients using SPP. The filter program is also responsible for keeping various information on the objects that are currently active as well reading/parsing the template file. The visualization engine processes messages from the filter and performs all the rendering of the objects. As Figure 4.1 indicates, the SPP connection is two-way. This enables the visualization engine to make requests of the filter and to alert the filter when graphical objects are selected. This adds a new level of functionality to the system in the way of selection callbacks, as explained in greater detail in section 4.3.3.

4.1 Detailed description

4.1.1 Filter

A filter program acts as an intermediary between the DGC and the visualization engine. The visualization engine is concerned only with polygon objects and the actions that can be performed on these objects. Therefore the filter is responsible for reading data and then mapping these data to the various methods of manipulations of graphical objects available. The filter then sends any resulting commands to the visualization engine via SPP.

This data-to-object/action mapping needs to be user-configurable, as ultimately only the user knows what visual form the data are to take. The filter's function, therefore, is to provide an easy, intuitive mechanism to allow the reading, parsing, and mapping of data so it can then be viewed using the visualization engine.

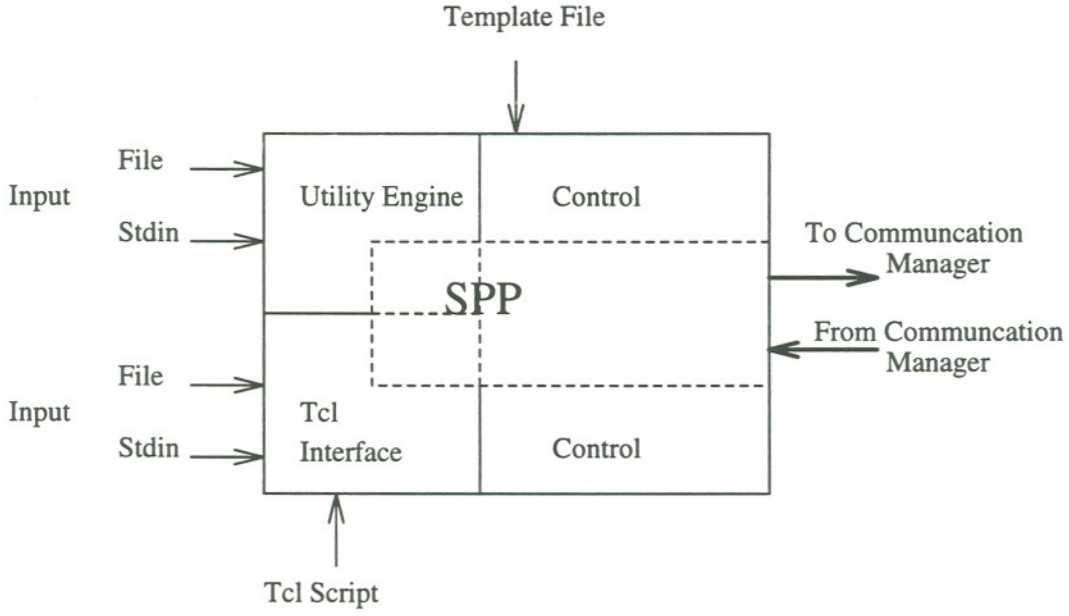

Figure 4.2: Filter in Detail

As can be seen in Figure 4.2, the filter program can be decomposed into different logical sections, each providing different functionality. The interface section handles the data input, parsing, and user-defined transformations/actions. The control section of the filter keeps a list of all of the objects, which contains information about each object necessary for filter operation. The template file contains basic object definitions, interface status information, and various parameters for the visualization engine. Both the Tcl and the utility-interface methods ultimately communicate with the visualization engine using SPP. The overall effect of the filter is to allow a stream of data to be mapped into a series of commands for the visualizer. These commands will either manipulate objects or change parameters of the visualization engine.

4.1.2 Graphical Objects

Graphical objects are really in the domain of the visualization engine, but as they are referenced often in the filter description it is helpful to describe the various objects available. The visualization engine supports two main classes of graphical objects: solid objects and sheet objects. Solid objects are generally closed geometrical objects such as spheres, cubes, and pyramids. Figure B.1 illustrates some of the different types of solid objects available. Sheet objects include the contour plot and histogrid which is a rectangular grid of cells. Each cell of a histogrid can be thought of as a 3D box or bar whose height can vary. The histogrid is vary much like a 3D bar chart. Solid objects may be moved via translations in the x,y, and z directions and rotated around their x, y, and z axes. Solid objects can also be linked together to form graphs and trees. Visually, this linkage is shown as a line connecting the two objects. Subtrees or subgraphs can be isolated to reduce the number of objects on the screen and to increase user comprehension. New types of solid objects may be added by modifying the fundamental object data file. Adding a new solid object subtype requires nothing more than adding the required data for the object to the fundamental object data file. The description for the object contains basic information on the color, vertices, and polygons of the object. The cube object, for example, contains information on the coordinates for the eight vertices, and what vertices make up the six polygons that compose the object.

Sheet objects contain the contour plot and 3D histogrid as subtypes. Sheet objects can be initially positioned, but cannot be rotated due to performance considerations. However this is not a major limitation as the view position can be rotated so the plot can be viewed from different angles. Contour plots, a subtype of the sheet object, are represented as a grid of points in the xz plane, with each point able to be moved in the y direction. Thus, one can generate a variable surface plot. The contour plot also allows color contouring based on the y value of each point. Contour color values and scaling are specified in the template. Figure B.2 shows an example of a contour plot. As mentioned above, the histogrid subtype is represented by a grid of 3D bars. Each bar can be colored, labeled and varied in height depending on the cell value. Figure B.3 illustrates a histogrid object.

All objects are dynamic in that they can be moved, altered and animated.

4.1.3 Template and Template Parser

As discussed in Section 4, the template contains basic object descriptions, initial visualization engine parameters, and the input and interface method to be used. The filter contains a parser that reads the template file and then initializes the filter accordingly. The template can be separated into three sections, object definitions, visualization parameters, and interface specification.

Object Definitions

The filter uses object definitions to create different graphical objects. The visualization engine supports two main types of graphical objects: solid polygon objects (including cubes, pyramids and spheres) and sheet objects (contour plots, histogrids). For the filter to create any of these objects, the corresponding base objects must be defined. The user defines a base object by specifying the desired base object type and a set of initial values for the object. Any number of graphical objects may be created from a base object. The created objects inherit all of the attributes from the base object with the exception of subtype, color, name, and initial position, which can be modified by the filter. Each type of base object has required initial values. These are explained in detail in section 4.4.1, which discusses the HDVS template.

An object definition for a sphere:

```
DefineObject
ObjectType     SOLID_OBJECT
ObjectSubtype  SPHERE
MoveCode       ACTUAL
Shading        GOURAUD_SHADING
Color          YELLOW
Name           Sun
Radius         80.9
AngleSize      30
Scale          1.0
Inity          0.0
Initx          0.0
CreateGrid     1
GridColor      3
EndDefine
```

This definition is for a solid object of the sphere type. Any type of solid object can be created from this base type. The user can create any number of objects from a base definition as long as they are of the same base type. For example, if one had a base definition of a cube solid object, from this one could create any type of solid object by giving it a different color, name, and initial position if desired. If a needed initial value is not supplied in the object definition, a default value is used. This occurs if a sphere is created from an object definition that does not specify a radius or angle size. To alleviate this problem, it is a good idea to define initial values necessary for all of the subtypes that will be created from a given object definition. If an initial value is not needed for a subtype, it is ignored.

Visualization Parameters

The template also contains basic initial settings for the visualizer. The user can define the lights, set the initial view position, and specify the colors and values for contour shading.

These settings are parsed and sent to the visualization engine upon startup of the filter. Explanations and illustrations of the various settings are presented in Section 4.4.1.

Interface Specification

The template file also defines the interface method. The user can specify if data are to be read from a file or piped directly to the filter. The user can also select one of many predefined data transformations or indicate that a Tcl script is to be used. Predefined data transformations and using Tcl are discussed in Sections 4.3.4 and 4.3.4 respectively.

4.1.4 Control

The control section of the filter is responsible for maintaining information on objects, performing data scaling, handling startup and initialization tasks, and providing a means for communication between the interfaces and the SPP. Current state information is kept for all objects that have been created. This information includes the object type, its movement status, and scaling status. The current position of each object is also maintained in the control section of the filter. For solid objects, the data are the x,y and z coordinates, as well as rotation values around the x,y, and z axes. Positions of sheet objects are kept as a list of all the grid points and their y values. Position information is kept for two reasons. First, by recording an object's current position, one can find the new location of an object simply by adding the incremental change to the current position. Secondly, since scaling is performed prior to sending a movement update message, the visualization engine will not posses the data values for the "real" position of an object. Because the data is recorded by the filter the data is not lost and can be retrieved from the filter by the visualization engine if needed.

The filter initialization responsibilities include memory management, SPP initialization, and interface initialization. The filter will pass control to either the Tcl interpreter or predefined utility function when initialization is complete, depending on what was specified in the template.

Most importantly, the filter provides an environment through which the chosen filter interface and the communication manager can communicate. This communication is achieved through a set of routines for creating and modifying objects, modifying the visualization engine, and handling incoming messages from the visualization engine.

4.2 Visualization Engine

4.2.1 General Description

The visualization engine is a 3D graphics engine that receives commands via an SPP interface. The engine is capable of rendering polygon-based objects as well as text and lines. The engine supports multiple light sources, variable and dynamic view positioning, and Gouraud shading. The engine requires no hardware support and is portable to most systems supporting the X Window System. The system utilizes the shared memory extension to X [11]. This allows a significant performance increase if the visualization engine is run on the same machine as the X server. In the future, OpenGL compatibility will be integrated into the 3D engine to take advantage of increasingly available graphics hardware. This will result in a dramatic increase in rendering performance for those machines with hardware support.

4.2.2 Interfacing to the Visualization Engine

The visualization engine provides an interface method via the SPP. All commands to create and modify objects are received by communication with the filter that originally created the object. The visualization engine also provides a control panel that enables the user to toggle among different viewing parameters. Other parameters of the visualization engine are set in the filter template or by the various interface routines.

4.2.3 Control Panel

The control panel for the visualization engine enables the user to specify different viewing options. The user can toggle between Gouraud shading and simple shading. The view type can be set to display polygon images, wire frame images, or a combination of both. Text can be toggled on and off. Animation speed can be adjusted. Contour shading can be enabled and disabled. Resolution of contour plots can be reduced or increased.

4.2.4 Features

The 3D engine supports many features, including multiple light sources, shading, and dynamic view positioning. There can be up to six light sources. A light source can be a point source or a directed "spot light" source. Objects can be shaded with Gouraud shading or flat shading. Gouraud shading produces a smooth look to polygon objects, while simple shading leaves the individual polygons distinguishable. Figure B.2 illustrates Gouraud shading where as B.4 is an example of simple shading. Light sources and shading are important aspects of visualization. They provide a more realistic 3D image that enables the user see depth and orientation of the view more easily. This permits the user to better assimilate the graphical data of the scene. The view position can be altered dynamically by using the mouse or the keyboard as well as by commands from the filter. This enables the user to navigate through the data, which is helpful in understanding large and complex data sets. Figures B.5 and B.6 illustrate variable view positions of the contour plot in B.2. Text may also be attached to any object and to any vertex of an object. Polygon-based objects may also be drawn as wire frame objects to improve animation speed. Figure B.7 provides an example of color scaling of a contour plot.

4.2.5 Control loop

The control loop for the visualization engine starts execution after a connection with the communication manager has been made and the **draw** command has been activated. This loop controls the rendering and message processing for the visualization engine.

Pseudo Code for the visualization engine control loop.

```
if redraw = True
     Display Buffer
Process Messages
Process X Events
     if event = quit exit program
   if have receive message or X Event
       Clear Buffer
      Perform Rendering
      redraw = True
  else
      redraw = False
```

- **Display Buffer**

  This control element outputs the buffer to the screen. The buffer contains the current rendered view.

- **Process Messages**

  This control element checks if any incoming messages have been received and, if so, will keep processing messages until it comes to a message with the more_to_come flag set to false.

- **Process X Events**

  This control element handles all of the X events that are currently in the event queue. Typical X events include resizing the window and activating any buttons on the control panel. If shared memory is being used, the control loop will block until the shared memory completion event is received from the display of the buffer. This prevents shared memory conflicts.

- **Clear Buffer**

  This control element clears the drawing buffer so a new scene may be rendered.

- **Perform Rendering**

  The rendering loop is executed if the visualization engine has received a message or X Event. If so, the rendering pipeline needs to be executed to redraw the buffer. The rendering pipeline acts on a list of basic graphical components, consisting of polygons, lines, and text bitmaps. Each time an active object is created, its base components are added to the generic object list. This list is then used in most of the rendering pipeline, which can be represented by the following pseudo-code loop:

```
Perform Lighting Calculations
Transform to user view
Build active list
Sort
   Process Active list
      Clip element of list
      Draw if still visible
```

The first two steps are performed on all of the objects. The first step determines the light intensity at each vertex or each polygon, depending on the shading type being employed. Next, all of the objects are transformed into the current view. The active list is built by going through each item in the generic object list and all items that are visible are added to the list. Visibility tests include backface removal and clipping in the z direction. This list is then sorted by z value. Finally each item in the active list is projected into screen space, clipped against the view window, and drawn if still visible.

4.3 Communication

The communication system consists of the communication manager program and the SPP message protocol. The communication manager controls the routing, resending and processing of messages, while the SPP is the underlying messaging protocol that all of the clients use to communicate.

4.3.1 Communication Manager

The communication manager can be considered the the hub of all inter-client communication. The communication manager handles all client connections as well as client message exchanges. In fact, any message that is sent from a filter client to a visualization client passes through the communication manager.

Connections

The HDVS employs a client-server connection model. The communication manager is the server and the various filter and visualization engines are the clients. Upon startup, the communication manager performs initialization and then periodically listens for connections on a specified TCP/IP port. Both filter and visualization clients can then connect to the communication manager at any time. When a client connects to the communication manager, it sends its type (filter or visualization engine) and is assigned an ID number. A client object is created, initialized, and added to the active client list in the communication manager. The client will include its ID number with all messages it sends. Upon client completion, the communication manager closes the connection. In addition to the normal updating that occurs upon client termination, filter clients require the communication manager to delete the filter's objects from all of the connected visualization engine clients.

Message Handling

The processing of messages is the primary responsibility of the communication manager. The communication manager determines which clients should receive messages and also performs any x-to-y mapping on the messages that needs to be done. Generally a message from a filter client will be sent to all of the visualization clients. This is because all visualization clients have the same global object list and ultimately display the same world view. Conversely, a message from a visualization client will only go to a filter client. For example, if an object is selected in the visualization engine, the message goes to the filter that created the object.

The communication manager also performs a translation on all messages. The visualization engine clients will have a list of all the currently active objects. Each object, however, belongs to only one filter. Therefore, the filter ID number for a graphical object is not the same as the ID number of the object in the visualization engine. The communication manager maps filter object numbers to visualization object numbers for messages originating from filter clients, and maps visualization engine object numbers to filter numbers for messages from visualization clients. The information needed for the object mapping is contained within each client in the form of a list. This list is modified each time an object is created or destroyed by a filter client.

The actions of the communication manager are for the most part, transparent as far as the clients are concerned. The filter and visualization engines send the same messages regardless of how many clients are connected. The communication manager supports a mechanism to enhance the functionality of the system by providing a means to add features to the message system such as synchronization.

4.3.2 Spinning Polygon Protocol (SPP)

The Spinning Polygon Protocol (SPP) is used in conjunction with the communication manager as the messaging system between the filter and the visualization engine. All communication between the two components must make use of the SPP routines available. The SPP contains message types that (a) manipulate graphical objects and (b) affect the visualization engine directly.

As can be seen from figure 4.3 the SPP can be separated into two logical levels. The bottom or transport level of the SPP uses TCP/IP and Berkeley sockets to send and receive messages. These low-level calls are encapsulated in a set of convenience functions. The top level, or interface layer, is used by the filter and visualization programs to send and receive different messages. The filter and visualization programs use an interface command set to interact with the interface layer. The command set include commands for creating objects and moving objects and is discussed in greater detail in Section 4.3.3. The communication manager communicates directly with the transport layer because it essentially resends messages with very little manipulation of the message itself. The double arrows indicate that all communication is bi-directional.

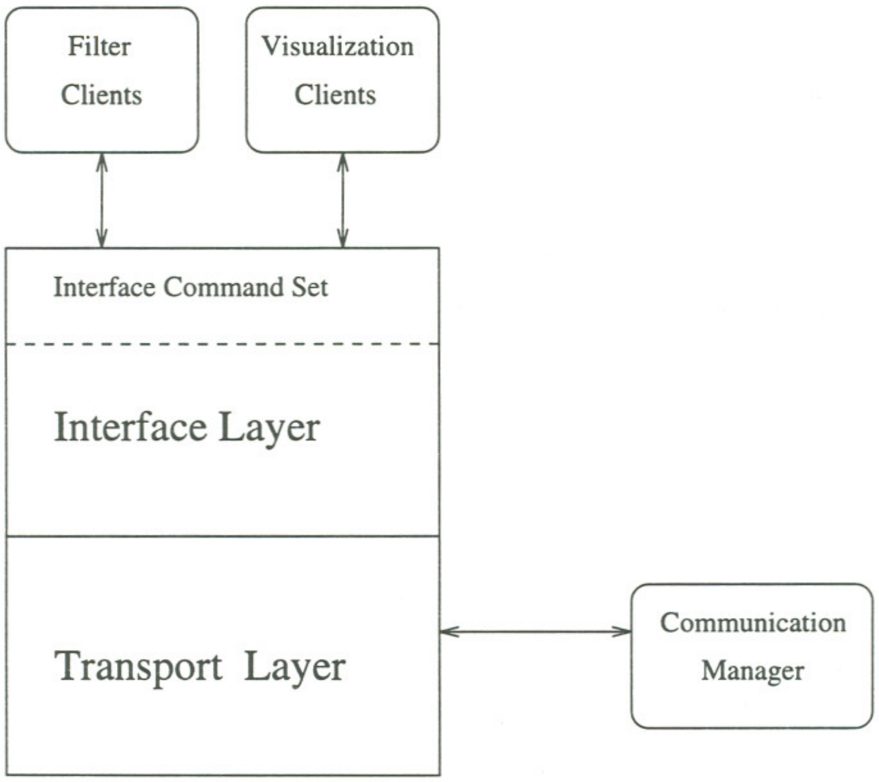

Figure 4.3: SPP Structure

The transport layer of the SPP contains the commands for initializing the communication manager server and the filter and visualization engine clients. There are also commands for establishing a connection between the client and server and sending and receiving messages. These commands were designed to be used by the interface level and are not meant to be accessed directly.

Transport Layer

The transport layer is used by both the communication manager and its clients. As will be seen in section 4.3.3, only the clients use the interface layer.

Message structure The transport level encapsulates messages as a message header followed by the message data. The message header contains the message type, the number of data objects it contains, the data length, and a flag that indicates if there are more messages to come.

The message header has the following structure:

```
SPP header_struct{
long message_type;
long number_of_objects;
long data_length;
long client_id;
long info;
long more_to_come;
data area;
}
```

The transport layer uses the message type during the reconstruction of an incoming message. Message reconstruction involves casting and setting the appropriate pointers to the data portion of the message. The receiving end uses the message_type and the number_of_objects fields to determine how the messages are to be processed. The data_length is necessary for both the sending and receiving of messages. The client_id is a unique number that specifies a specific client. The info field is used by the communication manager to give a connecting client its id number. Finally, the more-to-come flag indicates if more messages are expected.

The more-to-come flag is used only by the visualizer clients. The flag enables the receiving end to process multiple messages before proceeding to the rendering phase of the visualization engine's control loop. If the more_to_come flag of a message is set to true, the message will be processed and the message handler will block and wait for the next message. Otherwise, control immediately passes to the rendering phase and the scene is updated. If a group of incoming messages is processed before the screen is updated, a significant increase in performance can be achieved. This is due to the elimination of unnecessary scene rendering. Accordingly, many SPP commands involving object manipulation set the more_to_come flag to true to take advantage of this. A group of messages sent with the more-to-come flag equal to true must eventually be followed by another message to cause the screen to be updated.

Message Buffer When any module (filter, visualization or communication manager) is created and the SPP is initialized, a dynamic message buffer is created. All incoming and outgoing messages will be stored in the message buffer. The buffer contains the message header followed by the message data. The transport level routines that cause the building and sending of messages assume that the data have already been loaded into the data portion of the message buffer by the particular interface routine that was called. Having only one buffer does not pose any memory conflicts because the building and sending of a message, like the receiving and handling of a message, is an atomic event. Once a message is constructed, it must be sent before any other messages can be reconstructed. Likewise, once a message is reconstructed, it must be processed before any more messages can be sent or received.

Sending and Receiving Messages The SPP routines for sending and receiving messages are quite straightforward. Because the interface layer is responsible for copying the data into the message buffer, the transport-level **send** function needs only to determine the data length, to construct the message header, and to send the message. The receiving function reads the message, copies the message header and data into the message buffer, and sets the appropriate pointers so the data can be accessed by the appropriate message-processing routines.

4.3.3 Interface Layer

The interface layer is only used by the HDVS clients, since the communication manager is only concerned with processing and routing messages.

Creating and Sending Messages

The SPP contains a set of commands at the interface level that create and send different types of messages. Each interface function requires information on where the data are located and how many data items are to be sent. Each of these commands will copy the data to be sent to the message buffer and then call the transport-level routines discussed in section 4.3.2 to actually send the message.

The types of messages that can be created and sent can be separated into two logical types: messages that deal with object control and those that are used to control the visualization engine.

Object Control Messages

Messages used for object control can be further separated into (1) messages sent from the filter to the visualization engine and (2) messages sent from the visualization engine to the filter.

Messages from the Filter

- Create Objects

  This function will create one or more objects. It will update the appropriate object state information in the filter and send a list of objects to the visualization engine. The visualization engine then creates a graphical object for each object specified in the list.

- Delete Objects

  This function will delete one or more objects. It will update the object state information and send a list of the objects, referenced by number, to the visualization engine. The visualization engine then deletes each object in the list.

- Link Objects

  This function will draw a link between an object and a group of objects. It sends the link information to the visualization engine. The visualization engine then updates its object information with the new links. The links will be visually represented by colored lines between the linked objects.

- Change object Color

  This function will change the color of one or more objects. It will send the color information to the visualization engine. The visualization engine then changes the objects specified in the list to the given color.

- Alter Polygon

  This function will change the colors of a list of polygons. The polygons must belong to the same graphical object. The filter will save the original color of the polygon in its object state information to enable objects to be reset to their original color. The function will send a list of polygon-color pairs, enabling the new color for each polygon to be different. The visualization engine then updates the polygons specified in the list.

- Answer Resolution Request

  This function will change the resolution of the specified contour plot. The function updates the filter object state information and sends the new resolution information to the visualization engine. The visualization engine then changes the contour plot to the new resolution.

Messages from the Visualization Engine

- Request Resolution Change

  This function will request a resolution change for a contour plot. It will send the object number and the requested resolution to the filter. If possible, the filter will process the request, sending an Answer Resolution Request message to the visualization engine.

- Request Contour Change

  This function will request the contour attribute for a contour plot be enabled or disabled. It sends the object number and desired contour status to the filter. The filter then updates the object's state information to indicate the change in contour status.

Messages for Controlling the Visualization Engine These messages can also be further separated into (1) messages sent from the filter to the visualization engine and (2) messages sent from the visualization engine to the filter.

Messages from the Filter

- Create Lights

  This function will initialize a list of lights in the visualization engine. It sends a list of lights and their parameters to the visualization engine. The visualization engine will then update and turn on the specified lights.

- Create Text

  This function will attach a text string to a vertex of an object. It sends the text string, vertex, and object number to the visualization engine. The visualization engine then creates a text bitmap and attaches it to the specified object vertex.

- Change view

  This function will change the view position of the visualization engine. It sends the new coordinates and rotations of the view. The visualization engine will then update the view point.

- Reset

  This function is used to reset the filter and visualization engine. The reset can be hard or soft. A hard reset causes the connection between the filter and the visualization engine to be terminated. The visualization engine is reinitialized and will await a new connection from a new filter program. A soft reset causes the filter and the visualization engine to return to their initial state, deleting all objects, while retaining the connection. The function sends the type of reset and the visualization engine processes it accordingly.

- Update Message

  This function will cause the visualization engine to update the scene. It will send a Update message to the visualization engine. The update message is a dummy message with its more_to_come flag set to false. Upon receiving this message the visualization engine will exit the message processing loop and enter the rendering phase of the program.

Messages from the Visualization Engine to the Filter

- Select message

  This function indicates an object has been selected by the mouse in the visualization engine. It sends the object number and mouse button pressed to the filter. The filter then processes the callbacks that were registered for the indicated selection. Selection callbacks are only available in the C API library and are discussed in Section 4.4.5.

Processing Messages

Both the filter and the visualization engine have control loops that are executed to process incoming messages. All of the messages are designed to be used in one direction, either from the filter to the visualization engine or from the visualization engine to the filter. Therefore, any message used in the wrong direction will be ignored. An incoming message is reconstructed by the transport layer and then handled by message processing loops. If the message is an allowed type for the program that received it, the message is processed. The set of routines available at the interface level of the SPP are meant to provide all the necessary functionality for communication between the visualization engine and the filter. However, it is not difficult for new message types to be added if new functionality is needed. Moreover, the Multi Message type can be easily used for new message types that are not overly complex.

4.3.4 Interfacing to the Filter

The choice of an interface is quite important because it determines how the data are read, parsed, translated, and ultimately mapped into visualizer commands. The two interface methods available to users of the HDVS are a Tcl-scripting front end and a utility library. The two interface methods each have their own strengths and limitations with respect to ease of use, configurability, and performance.

Tcl Front End

The first method uses a Tcl front end and requires the user to write scripts in the Tcl language. This interface option's main strength is its configurability. The user has access to the commands contained in the Tcl scripting language plus a set of special-purpose commands to communicate with the filter and the visualization engine. Tcl scripts have most of the capabilities of a high-level computer language and can perform most required data parsing and mapping tasks. Using Tcl as the interface front end gives the user flexibility, yet eliminates the need to recompile parts of the HDVS. The interpretive nature of Tcl scripts makes using and modifying them quite easy. In fact, the Tcl front end can modify and create graphical objects on a command-by-command basis, providing a simple graphical editor.

Data parsing presents one of the major difficulties in visualizing heterogeneous types of data. The user must first determine what parts of the data are important and what translations should occur when mapping the data to the control of a graphical object or system parameter. The Tcl interface makes parsing and mapping data fairly simple with its built-in pattern matching capabilities.

Using the Tcl interface requires the completion of three tasks: altering the template, writing the script, and specifying the script to be executed. The script determines how the reading, parsing, and mapping of the data are to take place. The final step is the execution of the script. The major limitation of this interface method is the requirement that the user be knowledgeable enough about Tcl to write a script. Tcl is also is somewhat unwieldy for a scripting language.

Tcl commands for interfacing with the filter are closely related to the SPP interface-layer commands. In fact, most Tcl commands simply translate their data parameters into an acceptable format and then call the equivalent SPP interface command. The translation usually involves the conversion of a number represented by a string to the corresponding floating point representation. The Tcl command set provides an interface to the SPP as well as several convenience routines. Because the Tcl commands are so similar to the SPP the specific command syntax and description of these commands will be deferred until section 4.4.3, using the Tcl interface.

Utility Engine

The second interface option of the utilizes a utility engine. There are two options available with the utility interface: predefined utilities or utility creation using the utility library. The predefined utilities provide the simplest interface method available but are not configurable. In contrast, the utility-creation library is very configurable but requires the user to write the new utility. If a predefined utility is to be used it must be specified in the filter template.

The following predefined utilities are presently included.

Display Directory The UNIX command `ls -R` (recursive list) is used as input to the filter. The visual result of this utility is a 3D directory tree. The mouse can be used to isolate parts of the directory tree to improve comprehension of large trees. The mouse can also be used to execute programs by selecting the object representing the executable programs. Figure B.8 illustrates a full directory view of a sample directory tree. Different colors and shapes can be used to differentiate between directories, and types of files. Figure B.9 shows an isolated portion of the directory tree.

Display Top The UNIX command `top(1)` is used as input to the filter. The visual result of this utility is a dynamic 3D histogrid. Different values such as percent of CPU usage can be displayed. The value to be displayed is selected in the template. Figure B.10 is a visual representation of the `top(1)` command.

Normal Data Movement values for the created objects are used as input to the filter. The format and number of data elements depend on the type of the object. The utility assumes that the input will be a series of movement values of the correct type for each object created. Solid objects can be translated and rotated and require six data values: the x,y, and z position values and the x,y, and z rotation values. Sheet objects cannot be translated or rotated after creation so require only a list of y values. Contour plots require a y value for each grid point in the plot, while histogrids require a y value for each cell in the grid. The utility assumes that the input will be a series of movement values of the correct type for each object created. The movement values must correspond to the objects in the order they were created for each frame or movement iteration. For example, if three objects were created in the order [solid object, contour plot, solid object], then each iteration of movement data would consist of six values for the solid object: a set of y values for the contour plot, followed by six values for the third object. Because this option requires the filter to receive the correct movement type for each object in the order the objects are created, it is quite restrictive. Figure B.11 illustrates a fractal landscape generated with the Normal Data utility.

Normal Data Plus This utility is similar to the Normal Data utility but the objects to be moved are specified by an object number followed by the object data. A special flag is reserved to indicate that all movement data should be updated. This provides a slightly more flexible interface than the Normal Data method.

If the predefined utilities do not suffice and the Tcl front end proves to be unacceptable, new utilities can be written. A new set of functions must be created and registered to the filter. The functions must be written and C and will use the SPP interface to carry out HDVS commands. Creating new utility functions is explained in detail in Section 4.4.5.

4.4 Using the HDVS

4.4.1 Configuring the Template

The filter uses the template to construct a set of base objects, to set the interface type and to input method, and set various visualization engine parameters.

Keywords are written with the first letter of each word capitalized and no space between words. Data values can be numeric or predefined values, which are written in capital letters with each word separated by an underscore.

For example

```
Keyword  ObjectType
Data value  GOURAUD_SHADING or 1.2
```

Defining Base Objects

The following parameters can be set in an object definition.

```
DefineObject
 ObjectType         { SOLID_OBJECT  SHEET_OBJECT }
 MoveCode           { ACTUAL  DELTA }
 ObjectSubtype      { CUBE STAR PYRAMID SPHERE CONTOUR_PLOT HISTOGRID}
 Shading            { SIMPLE_SHADING  GOURAUD_SHADING }
 Color              { RED ORANGE_RED ORANGE YELLOW_ORANGE YELLOW }
                    { GREEN_YELLOW GREEN BLUE_GREEN BLUE }
 Name               any string


 Initx 0.0        # initial x position
 Inity 0.0        # initial y position
 Initz 0.0        # initial z position
 CreateGrid       { TRUE  FALSE }        #Create a wire frame for object
 GridColor        any color from above   #color for wire frame
```

Parameters Specifically for Solid Objects

```
 Radius 80.9      #radius of sphere
 AngleSize 30     #determines how many polygons make up sphere
 Scale 1.0        #scaling value
 Rotx 0.0         #initial rotation around x axis
 Roty 0.0         #initial rotation around y axis
 Rotz 0.0         #initial rotation around z axis
```

Parameters specifically for Sheet Objects

```
GridSize 20        #size of grids for sheet objects
Xdim 20            #number of squares in x dir
Zdim 20            #number of squares in y dir
```

```
EndDefine
```

Base objects are needed to create actual graphical objects. For each base object defined, one can create any number of objects that inherit all of the attributes of the base object except color, name, subtype, and initial position. To define a base object, the base type and specific initialization parameters must be specified. If a solid-type base object is defined, the Scale, Shading, and MoveCode are usually specified. If the subtype sphere is going to be used, the AngleSize and Radius values should also be set. If a sheet-type base object is defined, the Xdim, Zdim, Gridsize, and Shading fields should be set. Both base types can use the CreateGrid to specify a wire frame that can be toggled on or off. If a necessary value is left out of the object definition a default value will be used.

Defining the Interface

This section defines the interface method for the filter.

```
DataMethod       { PRE_DEFINED USER_DEFINED }

#if method PRE_DEFINED  pick on of the follow methods
MethodId     { NORMAL_DATA  DISPLAY_TOP  DISPLAY_DIR}

InputMethod   { FRM_FILE filename} or { InputMethod FRM_STDIN }
```

The DataMethod field indicates whether the filter should use the PRE_DEFINED or USER_DEFINED data parser. Specifying PRE_DEFINED indicates that a built in utility should be used. The available choices for predefined utilities are NORMAL_DATA, DISPLAY_TOP or DISPLAY_DIR. The specific functionality of these utilities is explained in detail in section REF_FORWARD. Specifying the DataMethod as USER_DEFINED indicates that a Tcl script will be used for data parsing. The InputMethod indicates whether the input data will be from standard input (FRM_STDIN) or from a file (FRM_FILE). If the input is from a file the filename follows the FRM_FILE directive.

Defining the Visualization Engine Parameters

Many parameters of the visualization engine may also be set in the template, including the number, type, and position of lights, the ranges and colors to be used for coloring of contour plots, and the initial view position.

```
Example Definition:
ViewDef Initx 200.0 Inity 120.0 Initz 500.0
ContourScale   RED -50.0  ORANGE_RED -35.0 ORANGE -20.0 YELLOW_ORANGE 0.0
               YELLOW 20.0 GREEN_YELLOW 35.0 GREEN 50.0
               BLUE_GREEN 80.0 BLUE 100.0
InitLights     LightNo 0 LightType NORMAL_LIGHT
               LightPosition -1.0 1.0 1.0
               LightNo 1 LightType SPOT_LIGHT
               LightPosition 200.0 50.0 -150.0
               LightDirection 0.0 -1.0 0.0
```

The ViewDef specifies the initial view point of the visualization engine. The direction of the initial view is always in the negative z direction.

The ContourScale determines the coloring of contour plots. The colors and corresponding ranges are specified in this section.

The InitLights specifies the initial lighting of the visualization engine. The type of light, direction and position are specified in this section.

In the following example, the template file displays a contour plot using the predefined utility for Normal Data.

```
# Object definition

TotalObjects 1

DefineObject
 ObjectType SHEET_OBJECT
 ObjectSubtype CONTOUR_PLOT
 MoveCode 2
 Shading GOURAUD_SHADING
 Color RED
 Name DataSet
 Scale 1.0
 GridSize 20
 Xdim 40
 Zdim 40
 Inity 0.0
 Initx -100.0
 CreateGrid 1
 GridColor 3
EndDefine

# Interface Method
DataMethod PRE_DEFINED MethodId NORMAL_DATA
InputMethod  FRM_STDIN
```

```
# Initial Visualization Engine Parameters
ViewDef Initx 200.0 Inity 120.0 Initz 500.0
ContourScale   RED -50.0  ORANGE_RED -35.0 ORANGE -20.0 YELLOW_ORANGE
               0.0 YELLOW 20.0 GREEN_YELLOW 35.0 GREEN 50.0
               BLUE_GREEN 80.0 BLUE 100.0
InitLights     LightNo 0 LightType NORMAL_LIGHT
               LightPosition -1.0 1.0 1.0
               LightNo 1 LightType SPOT_LIGHT
               LightPosition 200.0 50.0 -150.0
               LightDirection 0.0 -1.0 0.0
```

This template file defines a Gouraud shaded contour plot named DataSet. The predefined utility for NORMAL_DATA will be used for the interface method and the data is to be piped into the filter from standard input. The initial view position will be at the point (200.0,120.0,500). The contour scale will correspond to the values given and two lights will be enabled. The first light is a non-directional light source and the second light is a directional spot light.

4.4.2 Fundamental Objects File

This file contains descriptions on how to create the various solid object types supported. This information consists of information on the vertices, coordinates and how to construct the polygons. HDVS initially provides a file containing the information for creating cubes, pyramids and stars. Creating a new object type requires one to add the new object to the file.

4.4.3 Using the Tcl Interface

The Tcl front end to the filter enables the user to interact with the filter and subsequently the visualization engine, using the Tcl scripting language. This allows the user to create programs that can read and translate streams of data into a dynamic graphical representation. This method has a significant advantage over the other interface methods, primarily in the large degree of configurability available to the user. The user has the Tcl command set plus the added Tcl-filter commands. The user need not recompile anything, as Tcl is an interpreted language.

The Tcl-filter interface commands are primarily of two types, those that create and control graphical objects and those that alter parameters of the visualization engine. The list of commands available with an explanation of their functionality and the proper syntax is presented below.

Commands controlling graphical objects

Commands controlling graphical objects generally come in two forms. In one type, the user specifies the object or set of objects with their object numbers. The other method of referencing objects uses the object's name. Object numbers are assigned by the filter in the order objects are created. The name is specified upon creation of the object.

CreateObject {base_type subtype color initx inity initz name} This function creates one object from the given base type modified by the subtype value. The base type is an integer value that refers (in order) to the base object definitions in the template. The subtype specifies which subtype should be used. For example, if the base type is a solid object then the subtype could be for a cube, pyramid, star or sphere. The parameters initx, inity and initz refer to the initial coordinates used for object placement. The name is used to label the object and can also be used to refer to the object with other Tcl commands. All other characteristics that are needed for the creation of the object are used inherited from the base type definition.

DeleteObjectsByNum { objnum1 objnum2 ...}

DeleteOBjectsByName { objname1 objname2 ... } These commands will cause the specified list of objects to be deleted from the filter and the visualization engine. Currently, the object numbers are not reused and the object numbers will continue to advance as new objects are created.

LinkObjectsByNum { linkcolor objectnumfrom objnumto1 objnumto2 ...}

LinkObjectsByName{ linkcolor objectnamefrom objnameto1 objnameto2 ...}
These commands will create a visible link from the first object to each object in the object_to list. The link will be of the color specified. The links can be followed or traced in a limited manner in the visualization engine. For example, when viewing a directory tree a sub tree can be isolated for easier viewing.

ChangeObjectsColorByNum {color objnum1 objnum2 ...}

ChangeObjectsColorByNum {color objname1 objname2 ..} These commands will cause the given list of objects to change to the specified color.

ApplyMovementByNum {objnum numdatapoints listofdata}

ApplyMovemetMemByNum {objnum numdatapoints } These commands apply the given data to the specified object. The amount and type of data varies depending on the object. Solid objects use translations in the x, y, and z directions, as well as the rotations rotx, roty, and rotz around the object's origin. Movement for sheet objects requires a list of data values (y coordinates). Contour plots require a value for each grid point – a plot with xdim 20 and zdim 20 would require 21x21 or 441 points. Histogrids grids require one value for each cell of the plot.

The first movement command, ApplyMovementByNum, must have all the required data values for the movement as function parameters. This is useful for solid objects but rather unwieldy for sheet objects.

The second method assumes that all of the data values for the object have already been copied to an array. There are a set of memory commands that are useful in copying data to this array.

InitStaticMem {memsize} This command allocates an area in memory that can hold memsize number of floating point numbers. Currently, only one area may be initialized at a time. This area can be enlarged or decreased by calling this function again with the new size specified.

SetStaticMem {index value} This command sets the element in the array specified by index to the given value.

SetStaticMemRange {startindex range value1 value2 ..} This command will set the elements in the array starting with startindex and going until startindex+range. The number of values must be equal to or greater than range.

Once all the necessary values have been set, the call to ApplyMovementMemByNum may be performed.

Because the user may want to have many objects in the world move before updating the screen, calls to ApplyMovement.. will not have an effect until UpdateMovement is called.

UpdateMovement{} This command causes all of the objects that have been moved since the last UpdateMovement command to be updated in the visualization engine. If more than one ApplyMovement command has be called for one object, only the most recent one will take effect.

Update {} Filter Tcl commands usually result in a message being sent to the visualization engine. These messages may be acted on immediately or the visualization engine may wait for more related incoming messages. This is useful for letting the user create many objects, each with a separate command, and then allowing all the objects to be created before the screen is updated. This allows an increase in performance. Commands that are buffered in this way are

```
CreateObject
LinkObjects
ChangeObjectsColor
AttachText
```

To cause the visualization engine to proceed the user must send a Update command. This causes a message to be sent informing the visualization engine to stop waiting for messages. Thus the current scene will then be rendered.

As mentioned earlier, Movement commands have their own update command.

Visualization Engine Control

ChangeView {x y z rotx roty rotz} This command causes the view point and direction to be changed by the specified values. The view point is first moved to the given coordinates and then the rotations are performed.

Soft Reset This command causes a soft reset to occur. This causes all of the objects to be deleted in the filter and the visualization engine. Numbering of new objects will start at zero. The connection between the filter and the visualization engine will continue to exist.

ProcessMessage This command allows Tcl scripts to check for and process messages from the visualization engine. The function processes messages until the message queue is empty or until five messages have been processed.

SetLightSource { lightnum lighttype = NORMAL posx posy posz}

SetLightSource { lightnum lighttype = SPOT spotsize posx posy posz dirx diry dirz} This command allows one of four lights to be positioned and turned on. If the light type is set to be normal, a non-directional light is created and only the position of the light is needed. If the light type is a SPOT light then the spot size and spot direction are also required. If a light number is specified for a light that has already been set and the light is currently on, the new values will be used and the light will be updated. This is equivalent to moving the light.

TurnOffLight {lightnum} This command will cause the light specified by lightnum to be turned off. The light must be respecified with the SetLightSource command even if a light with the same parameters as the old light is desired.

AttachText {objectnum vertexnum text} This command will cause the text specified to be attached to the given vertex of the object. This text can then be viewed by using the text toggle button of the visualization engine. If the vertex is invalid the command will be ignored.

AttachTextHisto {objectnum cellnum text} This command is similar to the AttachText command but is used to attach text to a specific cell of a histogrid plot.

4.4.4 Using the Utilities

Predefine Utilities

Using the predefined utilities is the simplest and most straightforward interface method available for the HDVS. To use a predefined utility requires modification of only the filter template. This modification will generally include changing one or more base objects to the appropriate type, specifying which predefined utility is to be used, and selecting the input method for the data. As mentioned previously, the HDVS currently supports data input from files or from standard input.

Display Directory This utility displays a recursive directory tree. The following lines in the template must be modified as shown:

```
ObjectType  SOLID_OBJECT
Shading SIMPLE_SHADING
DataMethod PRE_DEFINED MethodId DISPLAY_DIR
```

The input method must be changed to either

```
InputMethod FRM_FILE filename
```

or

```
InputMethod FRM_STDIN
```

All other lines that are not applicable to solid objects are ignored.

To use the filter for input from standard input one would type `ls -R | filter`. The recursive directory will then be generated form the current directory and displayed graphically. The user can use the visualization engine to isolate subtrees by using the mouse interface. A program can also be executed by selecting the object with the mouse if the current directory is also the root directory of the tree.

Display Top This utility dynamically displays the `top(1)` command. The graphical representation consists of a 3D histogrid with the height of the bar representing the parameter specified in the template. The name of each process can be seen by selecting the "show text" button of the visualization engine.

The template must be changed in the following way:

```
ObjectType SHEET
ObjectSubType HISTO
Shading SIMPLE_SHADING
Xdim number
Zdim number
```

Xdim and Zdim should be large enough so all of the processes generated by top can be displayed.

```
DataMethod PRE_DEFINED MethodId DISPLAY_TOP
```

All other values can be set to whatever the user wants. All inapplicable values are ignored.

The command for animating top is `top | filter` (if the input option is standard input).

Normal Data This utility is used to animated any number and type of graphical objects. The only requirement is that the data for representing the movement of each object must appear in the order the objects were created. The data must also be of the correct type for the respective object. Solid data types have data in the form of six floating point numbers representing x, y, z translations and xrot, yrot, zrot rotations. Sheet data types will have a set of y values, the number of which is (xdim+1)x(zdim+1) for contour plots and xdim x zdim for histogrids. Because each object that currently exists must have movement data for each iteration of movement, this method is quite restrictive .

The template must include a object definition for each base type of object being used.

Normal Data Plus This is similar to Normal Data but the filter assumes each set of object movement data is preceded by the object number. This allows a more flexible method for moving objects as objects do not have to appear in any order or even at all. A flag number indicates that the movement data should be updated and the screen updated.

4.4.5 Creating New Utilities

New utilities can be created and used with the HDVS. One limitation of this option is the requirement that parts of the system be recompiled. Currently this option is not automated, and hence is not recommended for those unfamiliar with the HDVS.

Adding a new utility requires several tasks to be completed. First, the function(s) necessary for handling and translating the data must be written. This function should parse the data and then determine what actions (specifically what SPP calls) should be made. This routine would then have to be registered with the filter so it could be used as a predefined utility.

A typical routine would have the following loop:

```
While more data
  Get some data
  Parse data and make necessary translations
  Act on data using C interface to system (control graphical object or
  alter visualization engine)
  Process messages(includes incoming and outgoing messages)
End while
```

C API

The HDVS C API (Vislib) forms the basic mechanism for creating new utilities. The Vislib provides an interface that is very similar to that provided by the Tcl command set which was discussed in section 4.4.3. This similarity between the two interfaces exists because of the nature of the Tcl command set. All the Tcl commands are ultimately implemented by a corresponding Vislib C function. The HDVS Tcl commands essentially place the call data into the appropriate C data structures and then call the corresponding Vislib function. Therefore the functionality of the Vislib functions is the same as that of the HDVS Tcl command set. The actual C data structures used by Vislib and the set of function prototypes are listed in Appendix A.

Chapter 5

Discussion

5.1 Difficulties Along the Way

Many problems were encountered while building the HDVS. Some of the problems had easy solutions while others required a significant time investment to solve. Some issues were never adequately solved, as they were beyond the scope of this project. These issues can be separated in to three main categories, the visualization engine, the filter interface and the communication system.

5.1.1 Visualization Engine

The main obstacles encountered while developing the visualization engine dealt with functionality and performance. It became apparent early on that using the features available through X Windows would be inadequate for 3D animation. The PEX extension to X, which does support 3D rendering, did not meet performance requirements for animation. The low-level X library drawing primitives did not support 3D animation, and building a 3D engine based on these primitives would have been prohibitively slow. Some thought was given to using existing 3D engines but this idea was discarded for many reasons. The HDVS would require a very extensive interface to whatever graphical engine was employed. It was thought that the time investment in finding a suitable engine (if one even existed) and then learning how to interface with it could be better spent adapting a simple 3D engine the author had already built.

The author's simple engine was adapted to make use of the shared memory extension to X. This allowed a significant increase in performance by avoiding unnecessary memory copying. The engine was also modified to provide multiple light sources and Gouraud shading for polygon objects. By keeping the graphical objects somewhat simple, performance was streamlined to yield smooth animation for scenes consisting of 10,000 or fewer polygons.

5.1.2 Interface

Making a flexible and easy-to-use data input scheme was one of the problems faced when designing an interface for HDVS. A tradeoff had to be made between flexibility and user configuration. Because ultimately only the user knows what form the visualized data should take interaction at some level will always be necessary. The template, developed to address this issue, provides a simple method to configure the system. However, the template was not capable of completely solving the interface issue and so the utility engine and Tcl front end were added. These additions added two levels of functionality to HDVS. The utility engine enabled those with little or no computer experience to have access to data visualization techniques, and the Tcl front end provided a easily learned and flexible method for more extensive data visualization. In addition a C API was made available to enable more experienced users to write new utility programs.

5.1.3 Communication System

The communication system was created in response to several issues encountered while developing the system. Initially, the system was to be composed of a filter and a visualization engine. These two programs would communicate using TCP/IP sockets. To achieve this, the SPP was developed to enable messages to be passed between the two systems. The difficulty encountered in the implementation of SPP was creating a message system that could be extended to incorporate new message types. This flexibility was necessary because the functionality of the HDVS system often changed as it was developed, requiring new and different types of information to be exchanged between the two systems. Eventually the system was expanded to incorporate multiple filter and visualization programs. This added the complication of providing an environment where multiple filters could coexist in the same system. The communication manager was developed to address these issues.

The problems encountered while developing the system actually influenced how the final system was implemented. At each stage of development there were a new set of problems and the answers to these problems would take the system to the next stage in its maturation process. There are still issues to be addressed regarding the HDVS. These are discussed in Section 5.3.

5.2 System Goals

The goals of the ideal heterogeneous data visualization system were outlined in section 3. It is illustrative to see how successfully these goals were met.

5.2.1 Goals Fully Met

Multiple Data Inputs and Outputs The system is capable of integrating data generated from multiple computers. The system also enables visualization engines to be run on different machines while viewing the same data sets.

Distribution of Computation The system can be run in a fully distributed mode with the filters, visualization engines and communication manger able to be executed on different machines.

Communication Protocol The system provides a communication manger and extensible protocol (SPP) that support bidirectional message passing among the different clients of the system. The protocol is also extensible, to enable new message types to be created easily.

3D-Rendering Engine The system has a 3D-rendering engine that supports 3D surface plots, 3D histogrids and a variety of 3D base shapes. The system provides a means for the animation and display of graphical objects.

Flexible User Interface The system provides various methods to input and filter data. The system provides a utility engine, a Tcl front end and a C API to provide the user with multiple interface methods.

Phase Four System HDVS is a phase four system. It successfully separates the DGC and VC via the filter and visualization engine programs.

5.2.2 Goals Partially Met

Heterogeneous Data Types This was the only goal that was not fully met. The system does allow any type of data as input, but complex data forms require some form of parsing. This is unavoidable, as only the user fully understands how the data should be visualized. Ideally the system would provide a built-in scripting language specifically designed for parsing data. This would enable the user to configure the HDVS easily for any most any type of data.This is discussed more in section 5.3.

5.3 Future Work

Continued work on the HDVS will concentrate on two main areas: implementation and scope of use.

5.3.1 Implementation

The implementation of the 3D engine will be moved from C to C++, making it easier to add new types of graphical objects. New data-viewing constructs such as volume visualization will be added. In addition, the 3D engine will be improved to give better performance and will also utilize an OpenGL API in the rendering engine if it is available on the host computer.

The filter will be modified to provide a better support for the importing of data. A built-in interpreter that is specifically designed for easy configuration of data parsing will added. Furthermore, a control panel will also be added, enabling the user to dynamically modify data translation parameters such as scaling and translations. This feature will be particularly useful when integrating data from multiple sources. These changes will make the HDVS a system capable of meeting a wider variety of user needs.

5.3.2 The Holy Grail

The main goal of HDVS lies in its name: viewing heterogeneous types of data. This goal also implies the question, "How to maximize the viewable data types while minimizing the configuration requirements?". In some respects, these two ultimate goals are at odds with each other. One can see this by looking at the extreme cases. At one end of the spectrum lies a system capable of visualizing any type of data. This system would have unlimited flexibility, but would require the user to specify a translation for each type of data. At the other end, we have a system limited to a few basic data types that requires no intervention from the user whatsoever.

On the surface this seems to be a difficult problem. Looking at HDVS may give some insight on where to begin. HDVS is a Phase Four system, and as such has a logical separation between the DGC and VC. This separation makes it easier to see the weakness in data visualization systems. To truly be able to visualize heterogeneous data types, a system needs an intelligent, flexible filter to perform data parsing and mapping tasks. The key is not to concentrate on the latest rendering techniques and models, but on the filtering of the data. Because of the separation of DGC and VC, HDVS provides an environment suitable for development of the filter. The interaction between the two programs is minimal, enabling greater freedom in the future modification of the filtering system.

Bibliography

[1] S. Baum. *Software for Graphics and Data Analysis.* http://wwwocean.tamu.edu/~baum/ocean_graphics.html (February 1996).

[2] OpenGL Architecture Review Board. *OpenGL Reference Manual.* Addison Wesley, 1992.

[3] D.L. Brittain, J. Aller, and M. Wilson. Design of an end user visualization system. In A. Kaufman, editor, *Proceedings Visualization 1990*, pages 323–327. IEEE, ACM, IEEE Computing Society Press, October 1990.

[4] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, 1990.

[5] Center for the Computation and Visualization of Geometric Structures. *Geomview System.* http://www.geom.umn.edu/software/geomview/docs/geomview.html (May 1996).

[6] Greg Freiherr and Gary Pfitzer. The new breed. *Computer Graphics World*, 14(1):44–52, 1991.

[7] Richard S. Gallagher, editor. *Computer Visualization: Graphics Techniques for Scientific and Engineering Analysis.* CRC Press Inc., 1995.

[8] R. Hall. *Illumination and Colour in Computer Graphics.* Springer-Verlag, 1989.

[9] Dan Heller and Paula M. Ferguson. *Motif Programming Manual.* O'Reilly and Associates Inc, 1994.

[10] Advanced Visual Systems Inc. *AVS: Application Visualization System.* http://www.avs.com/product/avs.html (April 1994).

[11] Eric F. Johnson and Kevin Reichard. *Professional Graphics Programming in the X Window System.* MIS Press, 1993.

[12] Christopher Lampton. *Flights of Fantasy: Programming 3D Video Games in C++.* Waite Group Press, 1993.

[13] U. Lang, R. Lang, and R. Ruhle. Integration of visualization and scientific calculation in a software system. In *Siggraph '91: 18th International Conference on Computer Graphics and Interactive Techniques*, pages 123–145. SIGGRAPH, 1991.

[14] B.H. McCormick, T.A. DeFanti, and M.D. Brown. Visualization in scientific computing. *Computer Graphics*, 21(6):1-13, 1987.

[15] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley, 1994.

[16] D. Prawel, M. Brown, R. Kriz, and M. Vigil. How visualization applications drive tool selection: One product can't do it all. In L. Rosenblum G. M. Nielson, editor, *Proceedings Visualization 1991*, pages 345–347. IEEE Computing Society Press, October 1991.

[17] Valeria Quercia and Tim O'Reilly. *X Window System User's Guide.* O'Reilly and Associates Inc, 1993.

[18] George G. Robertson, Stuart K. Card, and Jack D. MacKinlay. Information visualization using 3d interactive animation. *Communications of the ACM*, 36(4):56-71, 1993.

[19] Will Schroeder, Ken Martin, and Bill Lorensen. *Vtk: The Visualization Toolkit.* http://www.cs.rpi.edu/~martink (May 1996).

[20] John Stasko. *POLKA Animation System.* Georgia Institute of Technology, http://www.cc.gatech.edu/gvu/softviz/parviz/polka.html (April 1996).

[21] Lloyd A. Treinish, David M. Butler, Georges G. Grinstein, and Steve T. Bryson. Grand challenge problems in visualization software. In *Siggraph '92: 19th International Conference on Computer Graphics and Interactive Techniques*, pages 233–238. Siggraph, 1992.

[22] Allan Tuchman, David Jablonowski, and George Cybenko. Run-time visualization of program data. In A Kaufman, editor, *Proceedings Visualization 1991*, pages 255–261. IEEE, IEEE Computing Society Press, October 1991.

[23] Edward R. Tufte. *The Visual Display of Quantitative information.* Graphics Press, 1983.

[24] Jeremy Walton. *Animation and Scientific Visualization*, Chapter 3, Get the Picture: New Directions in Data Visualization. Academic Press, 1993.

[25] Wavefront Technologies, *Wavefront: The Advanced Visualizer User Guide*, Wavefront Technologies Inc., 1991.

Appendix A

C API

Data types and function prototypes for the C API.

Data types:

```
/* data type used for creating objects */

typedef struct {
int object_number;
int object_type;
int sub_type;
int moveable;
int shading;
int color;
int scale;
int initx;
int inity;
int initz;
int rotx;
int roty;
int rotz;
float scalex;
float scaley;
float scalez;
int xdim;
int zdim;
int grid_size;
int radius;
int angle_size;
int create_grid;
int grid_color;
char name[MAX_NAME_LENGTH];
int text_color;
}create_object_type;



/* data type for altering polygon color */

typedef struct {
int object_num;
int polygon_num;
int new_color;
}alter_polygon_type;




/* data type for changing state information */

typedef struct {
int state_num;
union {
res_request_type res_request;
contour_request_type con_request;
}state_union;
}state_change_type;
```

```
/* data type for changing the resolution of a contour plot */

typedef struct {
int object_num;
int res_dir;
int interp_method;
}res_request_type;

/* data type for requesting contour coloring */

typedef struct {
int object_num;
int contour_status;
} contour_request_type;


/* data type for requesting a resolution change */

typedef struct {
int object_num;
int nxdim;
int nzdim;
int res_dir;
int grid_size;
}res_change_type;

/* data type for deleting an object */

typedef struct {
int object_num;
int status;
}delete_object_type;
```

```
/* data type for selecting an object */

typedef struct {
int object_num;
int button;
}select_object_type;

/* data type for modifying lights */

typedef struct {
int light_no;
int light_type;
int on;
float px,py,pz;
float dx,dy,dz;
}light_mess_type;

/* data type for changing the view position */

typedef struct {
float x,y,z;
float rotx,roty, rotz;
float type;
}view_mess_type;


/* data type for attaching text to a vertex */

typedef struct {
int object_num;
int vertex_num;
int y_offset;
char text[50];
}text_mess_type;


/* data type for reseting the visualization engine */

typedef struct {
int reason;
}reset_type;

/* data type for linking to objects together */

typedef struct {
int object_to;
int object_from;
int line_color;
}link_object_type;


/* catch all structure  for simple message requirements */

typedef struct {
int sub_type;
int object_num;
int temp1;
int temp2;
int temp3;
}multi_use_type;

MESSAGE FUNCTION PROTOTYPES:


void create_and_send_change_object_color(int num_objects, int color,
 int *object_list,int more_to_come);
```

```
void create_and_send_text_message(int num_objects,
                                  text_mess_type *temp_text);

void create_and_send_dummy_message();

void create_and_send_view(view_type *view_ptr);

void create_and_send_change_res(int object_num,res_change_type **res_ptr,
      int *num_altered,int newx, int newz,int res_dir);

void create_and_send_create_object(int cur_object_count,
   create_object_type *create_ptr,
                                   int num_objects,int more_to_come);

void create_and_send_link_object(filter_object_type *filt_ptr,
 int num_links,int object_from,
 int *link_array,
 int color,
 int more_to_come);

void create_and_send_delete_object(filter_object_type *filter_ptr,
   int *object_list,int  num_objects,
   int more_to_come);

void create_and_send_alter_cell(int object_num,int num_cells,
                                int *cell_list,
int *color_list);

void create_and_send_alter_polygon(filter_object_type *filter_ptr,
   int object_num,
   alter_polygon_type ** alt_ptr,
   movement_data *move_ptr,
   int num_colors,
   float *scaling,int *colors);
```

```
void create_and_send_light_message(int num_lights,
   light_source_type *light_defs);
```

Appendix B

Figures

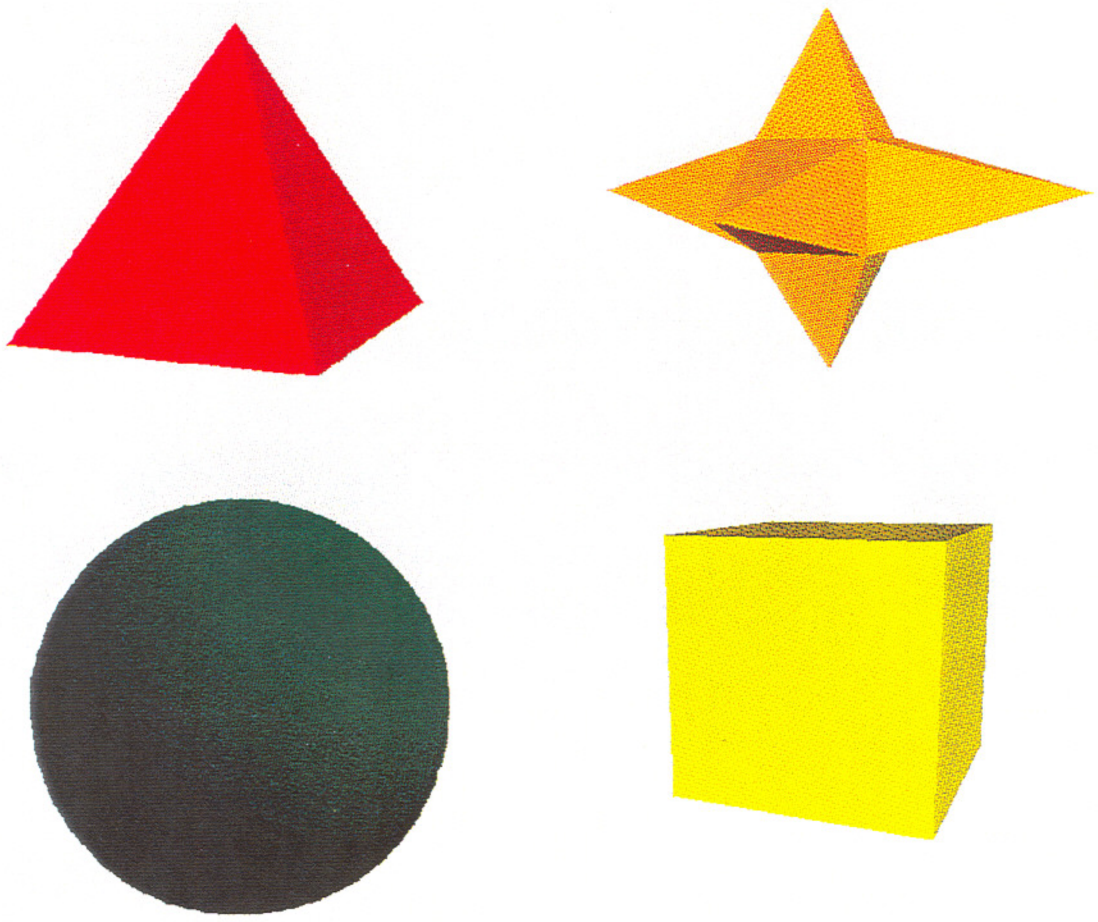

Figure B.1: Solid object primitives

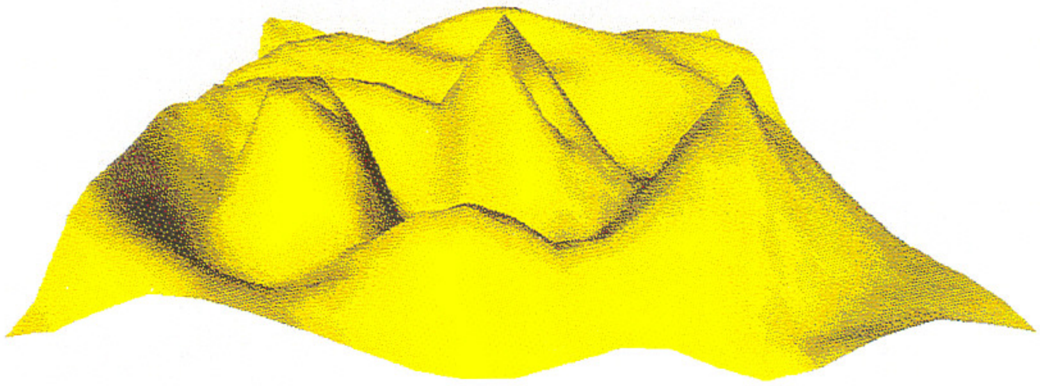

Figure B.2: Contour plot

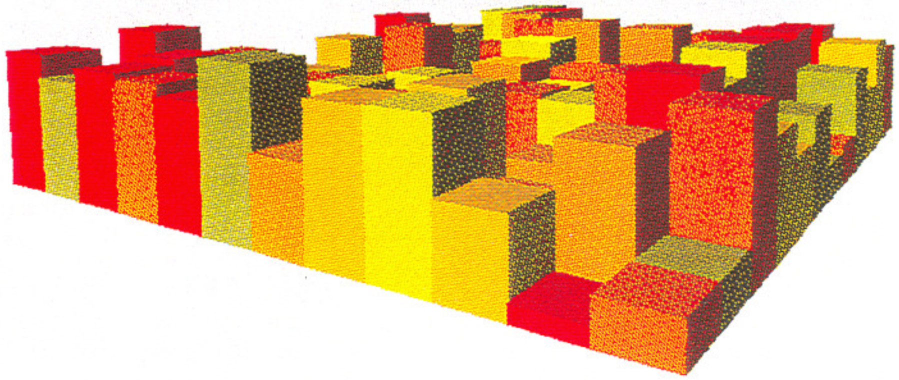

Figure B.3: Histogrid object

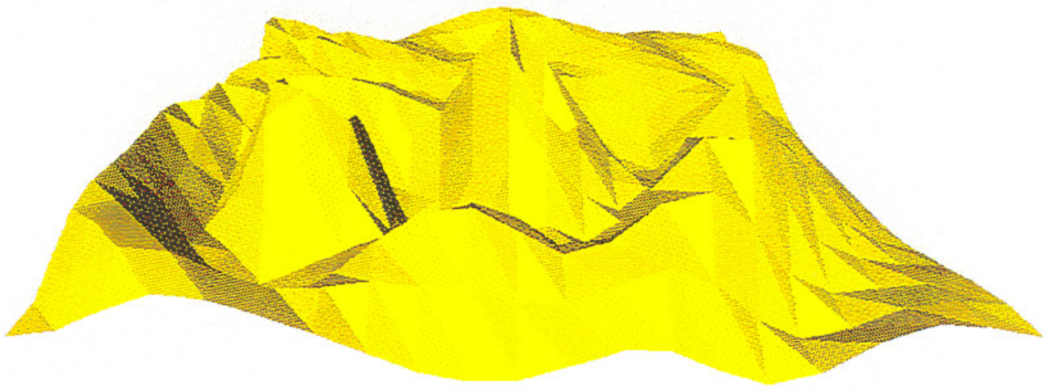

Figure B.4: Simple shading of contour plot

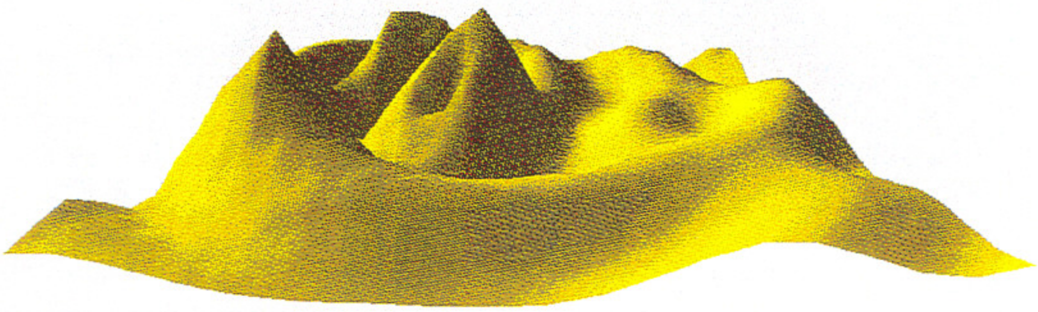

Figure B.5: Contour plot of B.2 with viewpoint rotated 90 deg

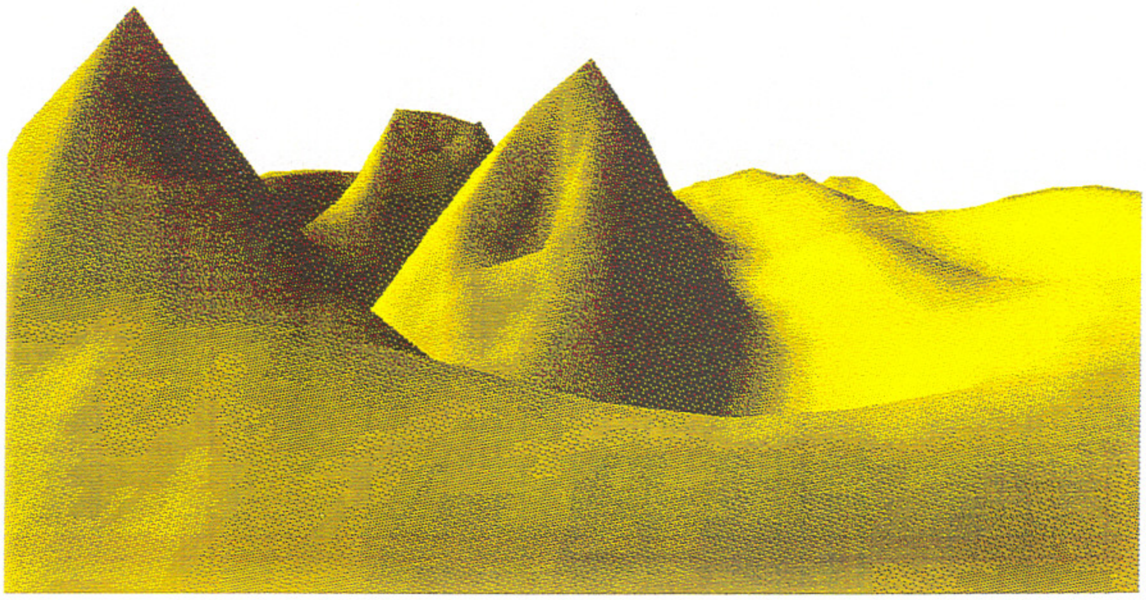

Figure B.6: Contour plot of B.2 with viewpoint closer to object

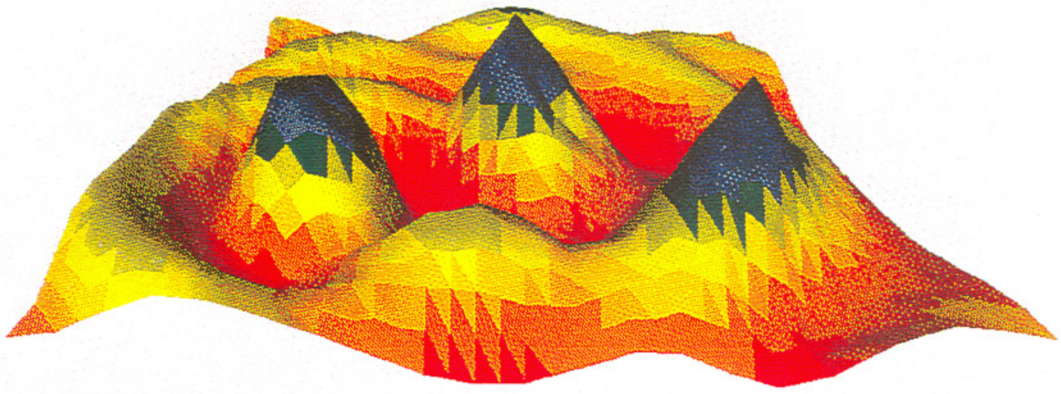

Figure B.7: Contour Plot with color scaling

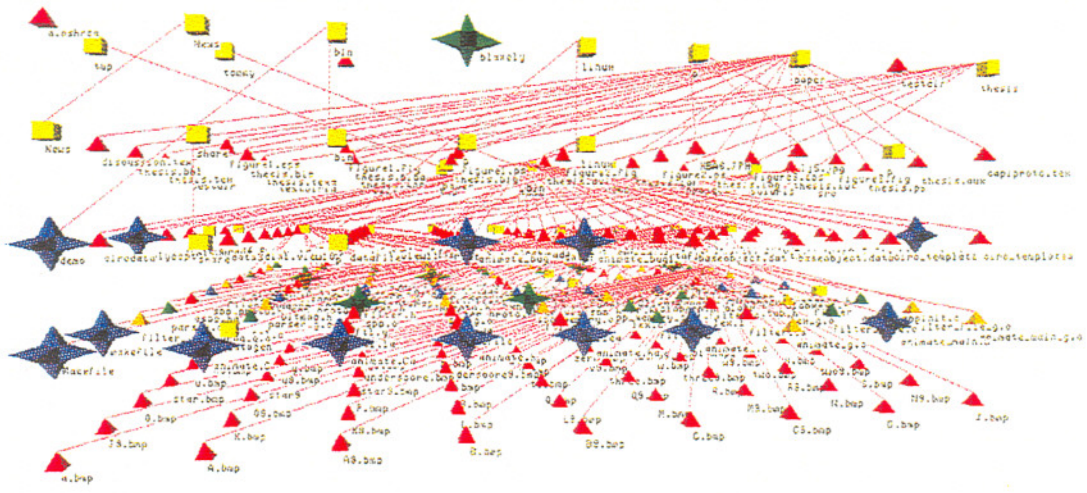
Figure B.8: Directory Utility: Full directory view

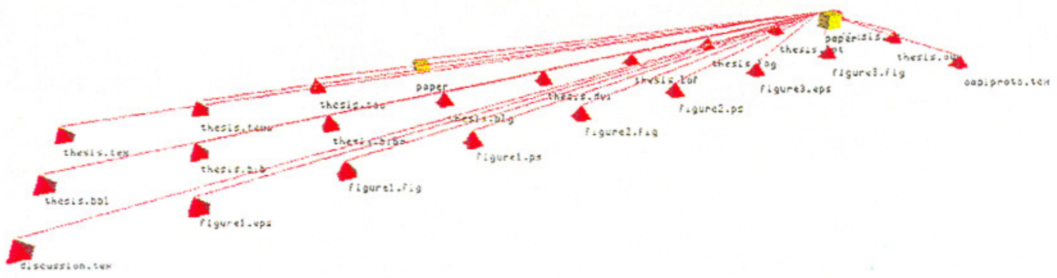
Figure B.9: Directory Utility: Isolated directory view

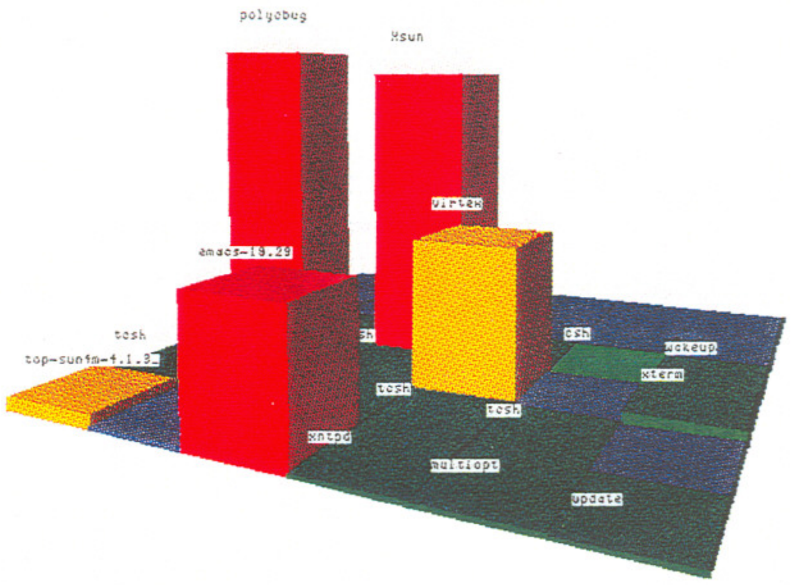

Figure B.10: Top Utility

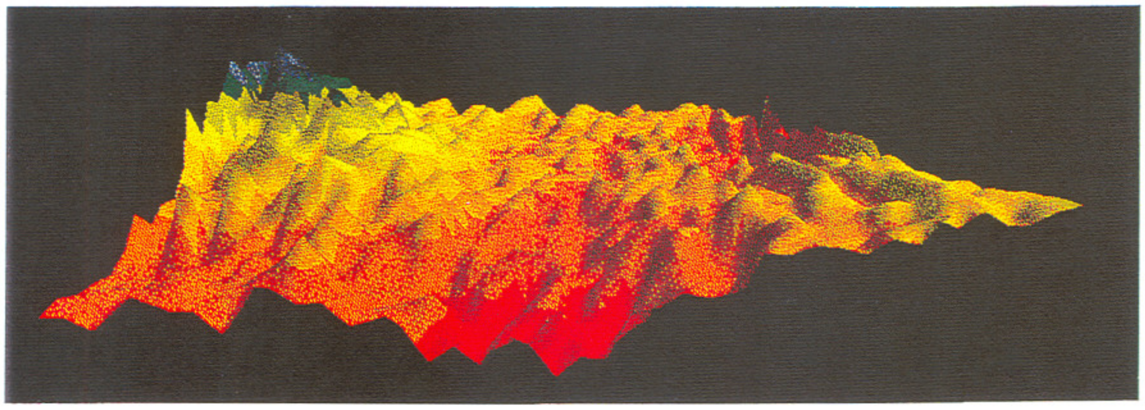

Figure B.11: Normal Data: Fractal landscape

Biographical Note

James Blakely was born in Bend, Oregon on May 10th, 1964. He graduated from Mountain View High School in 1982. As an undergraduate student, he attended Willamette University, Arizona State University, and finally graduated from Whitworth College with a BS in Physics and a minor in Mathematics in 1988. He went on to pursue an advanced degree in physics at the University of New Mexico. After several years of boredom, he left UNM in 1993 with an MS degree in Physics and transferred to OGI to pursue his true calling in computer science. In the summer of 1994 he designed the the 3D graphical display system in an advanced code browser for Imagix Corp. James enjoys most aspects of computer science, but has particular interest in computer graphics, data visualization, and the UNIX operating system. His hobbies include running, cycling, basketball, and contemplating the many combinations and varieties of puddin'.