NASA-CR-198007

FINAL
IN-45-CR
10 CIT-
63064
P. 130

A Final Report to

National Aeronautics and Space Administration (NASA)

prepared for

George C. Marshall Space Flight Center (MSFC)
Marshall Space Flight Center
Alabama 35812

for

Interactive Access and Management for Four-Dimensional Environmental Data Sets Using McIDAS

Grant # NAG8-828

University of Wisconsin Account # 144-BQ02

for the period of

1/1/90 thru 9/31/94

submitted by

William L. Hibbard
Gregory J. Tripoli

Space Science and Engineering Center
at the University of Wisconsin - Madison
1225 West Dayton Street
Madison, Wisconsin 53706

April 1995

N96-12989
Unclas
G3/45 0063064

(NASA-CR-198007) INTERACTIVE ACCESS AND MANAGEMENT FOR FOUR-DIMENSIONAL ENVIRONMENTAL DATA SETS USING McIDAS Final Report, 1 Jan. 1990 - 31 Sep. 1994 (Wisconsin Univ.) 130 p

I. INTRODUCTION

This grant has fundamentally changed the way that meteorologists look at the output of their atmospheric models, through the development and wide distribution of the Vis5D system. The Vis5D system is also gaining acceptance among oceanographers and atmospheric chemists. Vis5D gives these scientists an interactive three-dimensional movie of their very large data sets that they can use to understand physical mechanisms and to trace problems to their sources.

This grant has also helped to define the future direction of scientific visualization through the development of the VisAD system and its lattice data model. The VisAD system can be used to interactively steer and visualize scientific computations. A key element of this capability is the flexibility of the system's data model to adapt to a wide variety of scientific data, including the integration of several forms of scientific metadata.

This grant has also contributed to the development of the UW-NMS (University of Wisconsin Non-hydrostatic Modeling System), derived from the RAMS model.

II. WORK SUMMARY

We will briefly describe the accomplishments under this grant according to the outline of the Proposed Work Section of the grant proposal. The papers included in Appendix B provide a more detailed description of the accomplishments under this grant.

A. DEVELOPMENT OF FOUR-DIMENSIONAL ANALYSIS TECHNOLOGY

1. Data Management

We developed a five-dimensional grid file format, called v5d format, that serves as the input data format for the Vis5D system. This five-dimensional structure is a two-dimensional array of three-dimensional spatial grids, where the two-dimensional array is indexed by time and model output field. This structure is diagrammed in our paper Interactive Visualization of Earth and Space Science Computations, included in Appendix B. This file format can accommodate:

1. A mix of three-dimensional fields and two-dimensional fields (i.e., fields with a single vertical level such as surface pressure).
2. A variety of map projections.
3. Missing data indicators.
4. Variable data compression.

Because atmospheric and ocean models produce data in a wide variety of different file formats, we have provided a set of well-documented and easy-to-use tools to help users convert data from their native format to v5d format. Users have written converters

for standard formats such as HDF, GRIB and GRADS, and we include these converters with the ftp distribution of Vis5D.

We provide tools to help users import their own topographical data into Vis5D. We also provide simple formats for users to import image data into Vis5D for overlay on the topographical map - this may be used to visually compare satellite images with model output.

The VisAD system has very sophisticated internal data management, but does provide this as a file format. Rather, VisAD includes built-in functions for importing a variety of data formats from McIDAS, and also allows users to write their own data import functions in either C or Fortran.

The VisAD system tightly integrates visualization with computation, providing support for computational steering and visual analysis of high-level science algorithms. In order to do this, the VisAD data model is extensible, allowing users to define data types appropriate to their applications. This data model users lattice theory as the foundation for integrating a variety of scientific metadata into its data management. For example, satellite navigation and missing data indicators fit very naturally into this data model. Detailed descriptions of this data management are available in several of the papers included in Appendix B.

2. Data Analysis

It is important for scientific visualization systems to de-couple data analysis from data management and visualization, so that Earth scientists and system designers can each concentrate on their area of expertise. We have taken this attitude in the Vis5D and VisAD systems. Vis5D supports user-written data analysis in two ways:

1. Through simple type-in formulas that define new fields as functions of existing fields. For example, users may calculate wind speed by typing "SPD=SQRT(U*U+V*V)".
2. By writing Fortran programs that are dynamically linked with Vis5D to compute new fields as more complex functions of existing fields.

The VisAD system includes a programming language in order to support the close integration of visualization with computation. This provides a very flexible way for users to write data analysis algorithms. This programming language can dynamically link with functions written by users in either C or Fortran, which is useful for large computations and for supporting users' existing data analysis software.

3. Visualizing Data

Both Vis5D and VisAD provide interactive, animated, three-dimensional visualization. That is, images are rendered in a fraction of a second so that users can interactively rotate them in three dimensions and interactively control images in other ways.

The Vis5D system renders scalar volume data by:

1. Volume rendering (i.e., semi-transparent colored "fog"). Users can interactively adjust the mapping from data values to colors and transparencies.
2. Iso-surfaces. Users can interactively control the iso-levels of surfaces.
3. Horizontal and vertical 2-D plane slices filled with iso-lines. Users can slide these planes through the volume using the mouse, and can interactively control the spacing between contour lines.
4. Horizontal and vertical 2-D plane pseudo-colored slices. Users can slide these planes through the volume using the mouse, and can interactively adjust the mapping from data values to colors.

The Vis5D system renders vector volume data (i.e., wind and current velocities) by:

1. Trajectory ribbons. Users can dynamically generate these from a starting point defined using a 3-D cursor. They are integrated both forward and backward in time.
2. Horizontal and vertical 2-D plane slices filled with wind arrows. Users can slide these planes through the volume using the mouse, and can interactively adjust the spacing and length scaling of vectors.

Users can interactively select various combinations of fields and rendering techniques, as well as controlling animation and three-dimensional viewpoint (i.e., rotation, panning and zooming).

In both the Vis5D and VisAD systems, quantitative information can always be extracted from visual information through cursor locations, contour labels, color scale legends and data probes.

4. Interactive Workstation

The tremendous advancement of technology has ended the era of the custom designed meteorological workstation. That is, custom designs should now be confined to software. We have adopted the Unix workstation as the platform for the Vis5D system, including SGI, IBM, HP, DEC and SUN. The initial version of VisAD only runs on SGI workstations because it is less mature than Vis5D and because it places greater demands on graphics performance. However, it will ported to other Unix workstations. Both systems use the X, GL and OpenGL libraries for graphics. We will port these systems to

personal computers as there graphics performance improves and as software standards emerge in the personal computer market.

5. System Integration

The Vis5D system is designed for a single purpose, visualizing the output of atmospheric and ocean models, and this specialization permits a high degree of integration in its user interface and overall structure.

The VisAD system is designed for the broader purpose of interactively visualizing and steering scientific computations, but achieves coherence based on its simple but flexible models of scientific data, computations and displays. In each of these areas, the system gives users a few simple techniques that they can combine to design complex data structures, programs and displays.

B. ANALYSIS OF REMOTE SENSED DATA

The VisAD system is a flexible tool for analyzing remote sensed data, as demonstrated by the sample programs included in the system's ftp distribution. Its data model can adapt to the wide variety of sampling geometries and data characteristics of different remote sensing instruments. Its programming language allows the expression of a wide variety of data analysis algorithms. Its display architecture can be used to look at the same data in many different ways.

The VisAD system also provides the capability to compare data from different remote sensing instruments, and to compare such data with model output data, in geographically aligned Earth frames of reference.

The Vis5D system also provides a capability to compare satellite image data with model output data.

C. ANALYSIS OF MODEL DATA

The Vis5D system is focused on the task of visualizing model data. It also allows users to define data analysis algorithms, either by simple type-in formulas or by writing Fortran programs.

The VisAD programming language provides an extremely flexible capability for defining analysis algorithms for model output. It also provides the capability for interactive steering of models. This is demonstrated by an application of VisAD to Bob Aune's two-dimensional shallow water model. For example, the user can interactively adjust the time interval between time steps. If the interval is set over a critical value, the visualization reveals high frequency waves due to the development of numerical instability.

Another interactive control allows the user to apply a spatial filter and damp out the waves. This demo is included with the VisAD ftp distribution as the shallow.v program. This simple demo is primarily useful as a teaching tool, but it does illustrate the possibility for interactive steering of models.

III. ON-LINE INFORMATION

The results of the SSEC 4-D Visualization Project are available via the World Wide Web at the URL:

http://www.ssec.wisc.edu/~billh/vis.html

The UW-NMS model is used to make daily U. S. forecasts in Vis5D format, and these are available via the World Wide Web at the URL:

http://java.meteor.wisc.edu/vis5d-oper.html

The Vis5D and VisAD systems are also available by anonymous ftp. The Vis5D system is available as follows:

```
% ftp iris.ssec.wisc.edu
    or
% ftp 144.92.108.63

login: anonymous
password: myname@location

ftp> cd pub/vis5d
ftp> ascii
ftp> get README
ftp> bye
```

See section 2 of the README file for complete installation instructions.

The Vis5D ftp distribution includes:

1. Complete source code and makefiles.
2. Documentation and porting guide.
3. Sample data sets.
4. Support for importing data from various formats.
5. Executable files for SGI and IBM also available.

The VisAD system is available as follows:

```
% ftp iris.ssec.wisc.edu
     or
% ftp 144.92.108.63

login: anonymous
password: myname@location

ftp> cd pub/visad
ftp> ascii
ftp> get README
ftp> bye
```

See section 2 of the README file for complete installation instructions.

The VisAD ftp distribution includes:

1. Complete source code and makefiles.
2. Documentation and on-line help facility.
3. A large number of demo programs and data sets.
4. Built-in functions for reading McIDAS data files.
5. Support for importing data from other formats.
5. Support for linking with software written in C and Fortran.
6. Executable files for SGI are also available.

IV. APPENDICES

Appendix A. List of Publications

Appendix B. Copies of Selected Publications

APPENDIX A

This is a list of the papers written and videos produced under this grant. Copies of selected papers are included in Appendix B.

Hibbard, W., and D. Santek, 1990; Cold fronts moving across the north Atlantic. SIGGRAPH Video Rev., No. 61.

Hibbard, W., and D. Santek, 1990; The VIS-5D system for easy interactive visualization. Visualization '90, San Francisco, IEEE. 28-35.

Santek, D., T. Whittaker, J. Young, and W. Hibbard, 1991; The implementation plan for McIDAS-AIX. Preprints, Conf. Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology. New Orleans, American Meteorology Society. 177-179.

Hibbard, W., D. Santek, and G. Tripoli, 1991; Interactive atmospheric data access via high speed networks. Computer Networks and ISDN Systems, 22, 103-109.

Hibbard, W., 1991; Access - end user (scientist) view and environment subgroup. Part of SIGGRAPH '90 workshop report, data structure and access software for scientific visualization. Edited by Lloyd A. Treinish. Computer Graphics 25(2), 104-118.

Hibbard, W., and C. Dyer, 1991; Automated display of geometric data types. Univ. of Wisc. Comp. Sci. Dept. Tech. Report #1015.

Hibbard, W., and B. Paul, 1991; El Nino Satellite Observations and Downburst Simulation. SIGGRAPH Video Rev., No. 74.

Aune, R., G. Callan, and W. Hibbard, 1991; A 4D visualization of a 4D assimilation system. AMS conference, Denver, Oct. 14-18.

Tripoli, G. J., 1992; An explicit three-dimensional nonhydrostatic numerical simulation of a tropical cyclone. Meteorology and Atmospheric Physics 49, 229-254.

Hibbard, W., 1992; A highly parallel approach for satellite archive processing. Preprints, Conf. Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology. Atlanta, American Meteorology Society. 82-83.

Hibbard, W., C. Dyer and B. Paul, 1992; A development environment for data analysis algorithms. Preprints, Conf. Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology. Atlanta, American Meteorology Society. 101-107.

Hibbard, W., and B. Paul, 1992; Energy generation by controlled thunderstorm. SIGGRAPH Video Rev., No. 82.

Hibbard, W., and B. Paul, 1992; Distributed visualization at the Space Science and Engineering Center. Notes for SiGGRAPH course 7, Distributed Scientific Visualization on High-Performance Networks, 4.1-4.22.

Hibbard, W., 1992; Systems issues for the 120 Terabyte GOES archive. Proc., Managing Terabyte Databases in the 90s and beyond. NOAA and the MITRE Corp. 24-25.

Hibbard, W., C. Dyer and B. Paul, 1992; Display of scientific data structures for algorithm visualization. Visualization '92, Boston, IEEE, 139-146.

Hibbard, W., C. Dyer and B. Paul, 1992; Using VIS-AD to visualize a cloud discrimination algorithm. Video proceedings of Visualization '92, Boston, IEEE.

Rhyne, T., M. Bolstad, P. Rheingans, L. Petterson, W. Shackleford, M. Botts, E. Pepke, K. Johnson, W. Hibbard, C. Dyer. B. Paul, and L. Treinish, 1992; Visualization requirements in the Atmospheric and Environmental Sciences. Visualization '92, Boston, IEEE, 428-435.

Hibbard, W., C. Dyer and B. Paul, 1992; Graphical representations of scientific data. Proceedings, Workshop on Two and Three Dimensional Spatial Data: Representation and Standards, University of Western Australia.

Hibbard, W., W. Lagerroos, D Wade and N. Troxel-Hoehn, 1993; Design for and experience with the McIDAS GOES inventory. Preprints, Conf. Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology. Anaheim, American Meteorology Society. 140-143.

Hibbard, W., C. Dyer and B. Paul, 1993; VIS-AD data management. Preprints, Conf. Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology. Anaheim, American Meteorology Society. 158-161.

Paul, B., A. Battailoa and W. Hibbard, 1993; Progress with VIS-5D / distributed VIS-5D. Preprints, Conf. Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology. Anaheim, American Meteorology Society. 162-164.

Hibbard, W., B. Paul, C. Dyer and A. Battaiola, 1993; Interactive visualization techniques for large environmental data sets. Notes for SiGGRAPH course 71, Visualizing Planet Earth, 3.1-3.28.

Hibbard, W., and B. Paul, 1993; Interactive visualization of the computations of air quality models. Proceedings, Regional Photochemical Measurements & Modeling Studies. San Diego, Air & Waste Management Association.

Hibbard, W. L., B. E. Paul, D. A. Santek, C. R. Dyer, A. L. Battaiola, and M-F. Voidrot-Martinez, 1994; Interactive visualization of Earth and space science computations. IEEE Computer 27(7), 65-72.

Hibbard, W., C. Dyer and B. Paul, 1994; A lattice model for data display. IEEE Visualization '94, 310-317.

Hibbard, W., C. Dyer and B. Paul, 1994; The VIS-AD data model: integrating metadata and polymorphic display with a scientific programming language. In *Database Issues for Data Viualization*, Lecture Notes in Computer Science number 871. Edited by J. P. Lee and G. G. Grinstein. Springer-Verlag. 37-68.

Bergeron, R., W. Cody, W. Hibbard, D. Kao, K. Miceli, L. Treinish and S. Walther, 1994; Database issues for data visualization: developing a data model. In *Database Issues for Data Visualization*, Lecture Notes in Computer Science number 871. Edited by J. P. Lee and G. G. Grinstein. Springer-Verlag. 3-15.

Hibbard, W., and B. Paul, 1994; Classifying and modeling data in the physical and natural sciences. Notes for SiGGRAPH course 27, Visualizing and Examining Large Scientific datasets: a Focus on the Physical and Natural Sciences, I-1 to I-27.

Hibbard, W., and B. Paul, 1994; Hurricane Gilbert. SIGGRAPH Video Rev., No. 105.

Hibbard, W., and B. Paul, 1994; Real-time volume rendering of downbursts. SIGGRAPH Video Rev., No. 105.

Hibbard, W., and B. Paul, 1994; Visualizing atmospheric chemistry and physics with VIS-5D. Proceedings, International Specialty Conference on Computing in Environmental Management. Raleigh, NC. Air and Waste Management Association. 46-50.

Hibbard, W., C. Dyer and B. Paul, 1995; Interactivity and the dimensionality of data displays. IFIP WG5.10 Workshop on Perceptual Issues in Visualization. Edited by G. Grinstein and H. Levkowitz. Springer-Verlag.

APPENDIX B

These are copies of selected papers written under this grant.

reprinted from
COMPUTER

N15

Interactive Visualization of Earth and Space Science Computations

William L. Hibbard, Brian E. Paul, David A. Santek, and Charles R. Dyer, University of Wisconsin-Madison

André L. Battaiola, Instituto Nacional de Pesquisas Espaciais

Marie-Françoise Voidrot-Martinez, Service Centrale d'Exploitation de la Météorologie

Scientists often view computer algorithms as risk-filled black boxes. These visualization packages help scientists see the internal workings of their algorithms and thus understand their computations.

Computers have become essential tools for scientists simulating and observing nature. Simulations are formulated as mathematical models but are implemented as computer algorithms to simulate complex events. Observations are also analyzed and understood in terms of mathematical models, but the number of these observations usually dictates that we automate analyses with computer algorithms.

In spite of their essential role, computers are also barriers to scientific understanding.[1] Unlike hand calculations, automated computations are invisible and, because of the enormous numbers of individual operations in automated computations, the relation between an algorithm's input and output is often not intuitive. This problem is illustrated by the behavior of meteorologists responsible for forecasting weather. Even in this age of computers, many meteorologists manually plot weather observations on maps, then draw isolines of temperature, pressure, and other fields by hand (special pads of maps are printed for just this purpose). Similarly, radiologists use computers to collect medical data but are notoriously reluctant to apply image-processing algorithms to that data. To these scientists with life-and-death responsibilities, computer algorithms are black boxes that increase rather than reduce risk.

The barrier between scientists and their computations can be bridged by techniques that make the internal workings of algorithms visible and that allow scientists to experiment with their computations. Here we describe two interactive systems developed at the University of Wisconsin-Madison Space Science and Engineering Center (SSEC) that provide these capabilities to Earth and space scientists.

Visualizing Earth simulations

Numerical models of the Earth's atmosphere and oceans form one important class of scientific algorithms. The history files produced by these models are traces of their computations, and our Vis-5D system[2] is widely used by scientists for interactively visualizing these history files. This system takes its name from the fact that model his-

0018-9162/94/ $4.00 © 1994 IEEE

PRECEDING PAGE BLANK NOT FILMED

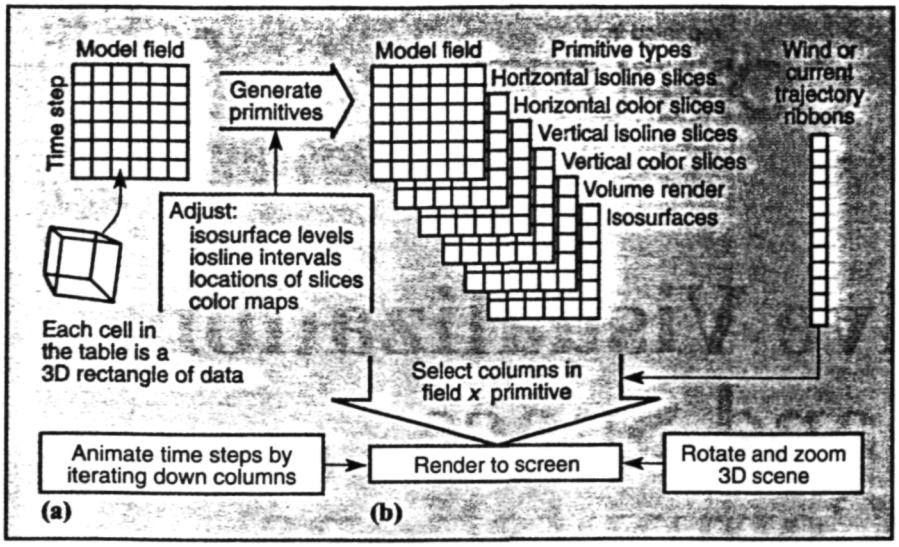

Figure 1. Vis-5D transforms simulations of the Earth's atmosphere and oceans into an interactive graphical environment: (a) array of 3D grids indexed by time step and field; (b) array of graphics primitives indexed by time step, field, and primitive type.

tory files are 5D rectangles of data, organized as 2D arrays of 3D spatial grids. The 2D arrays are indexed by time and by model field (for example, temperature, pressure, salinity, wind or current velocity, and so on).

Figure 1 shows the pipeline for rendering this data into 3D animations under the user's interactive control. The system transforms data grids into graphical primitives that consist of 3D vectors and polygons. (On large workstations, we also use an efficient interactive volume-rendering technique.[3]) The rendering of graphical primitives creates a virtual Earth environment behind the workstation screen. Users can reach into this virtual environment with a mouse to move slices through the data grids, place seed points for wind trajectories, and rotate and zoom their view. In Figure 2, the window on the right contains the virtual Earth environment. The array of icons on the left allows users to select combinations of fields and rendering techniques and to control animation, isolevels, trajectories, color maps, and so on.

Modern workstations can respond to these controls within the time of an animation step (usually between 1/30 and 1/5 second), giving users the sense of interacting with a small virtual atmosphere or ocean. To explore the 3D geometry of their fields, as well as cause-and-effect relationships between different fields, users should be able to rotate images and change the combinations of fields displayed without interrupting the smooth animation of model dynamics. Thus, we do not synchronize animation with the computation of graphical primitives; instead, we store primitives in intermediate tables indexed by time and field.

The size of a model history file is the product of five numbers and can be quite large. For example, a data set spanning 100 latitudes by 100 longitudes by 20 vertical levels by 100 time steps by 10 model fields contains 200 million grid points. To maximize data set size, we compress grid data and derived graphics by scaling them linearly to one- or two-byte integers. To preserve fidelity, we use different scaling factors for each horizontal slice of each 3D grid. With compression, we can store one grid point, plus derived graphics, in 2.5 bytes of virtual memory. For history files that are too large for workstations, the system splits into a graphics client on a workstation and a data server on a supercomputer connected via network.[4]

Sometimes users need to see derived quantities, such as the vorticity or divergence of air flow, to understand the physics of a simulation. Users can write C and Fortran functions for deriving new diagnostic fields and invoke them during a visualization session (they are dynamically linked with Vis-5D via sockets). To maximize data fidelity, these calculations use floating-point grid values in disk files rather than compressed values.

To illustrate how Vis-5D works, Figure 2 shows a snapshot of a numerical experiment performed by Gregory Tripoli and Peter Pokrandt of the University of Wisconsin-Madison using their UW-NMS (Nonhydrostatic Modeling System) weather model and visualized using Vis-5D. They are modeling a novel idea proposed by William Gray of Colorado State University for generating energy by creating a permanent rainstorm over a hydroelectric generator. The white object is a balloon 7 kilometers high in the shape of a squat chimney that floats in the air above a patch of tropical ocean. The purpose of the numerical experiment is to verify that once air starts rising in the chimney, the motion will be self-sustaining and create a perpetual rainstorm. The vertical color slice shows the distribution of heat (as well as the flow of heat when model dynamics are animated); the yellow streamers show the corresponding flow of air up through the chimney; and the blue-green isosurface shows the precipitated cloud ice (a cloud water isosurface would obscure the view down the chimney, so it has been toggled off for this snapshot). The simulation takes many hours to run, even on the largest computers, so the virtual time of the visualization is not in lock step with the model's computations. Rather, model output accumulates in a history file, and users are free to move around in simulated time, searching for problems. Once problems are found, users trace their root causes by working back through time and by comparing different model fields.

Michael McCann and Matthew Koebbe of the Naval Postgraduate School (NPS) applied Vis-5D to visualize the ocean simulation shown in Figure 3. This is a view from the north, looking at a region of the Pacific Ocean straddling the equator, including ocean bottom topography and a volume rendering of ocean current speed. Ocean models produce history files similar to atmosphere models, although the NPS model is remarkable for its high resolution and challenges the capacity of our visualization system.

Visualizing a broader class of computations

While Vis-5D is effective for visualizing simulations of the atmosphere and oceans, scientists also design and use a much broader class of algorithms that requires more general visualization techniques. We developed the Vis-AD (Visualization for Algorithm Development) system to meet this need.[5] Whereas Vis-

ORIGINAL PAGE IS
OF POOR QUALITY

5D runs as a postprocess to simulations. Vis-AD serves as the execution environment for scientists' algorithms, supporting a greater variety of visual experiments with algorithms. Where Vis-5D assumes that data are organized as a five-dimensional rectangle and that 3D graphical space always represents 3D physical space, Vis-AD lets scientists define their own data organizations and abstract graphical spaces to support a broad class of algorithms. The Vis-AD system combines

(1) A *data model* that includes complex data types defined in terms of tuples and functional relations. The data model integrates several forms of metadata based on a conceptual model of computer data objects as finite approximations to mathematical objects.
(2) A *computational model* based on a high-level interpreted programming language that supports distributed computing and can link to user-written functions in C and Fortran.
(3) A *display model* based on interactive, animated 3D voxel volumes. A novel technique lets scientists control how their data is displayed without placing a substantial burden of graphics knowledge on them.
(4) A *graphical user interface* that is highly interactive and gives scientists an integrated view of data, computation, and display.

The system functions like an interactive debugger with high-level data management and visualization. While a

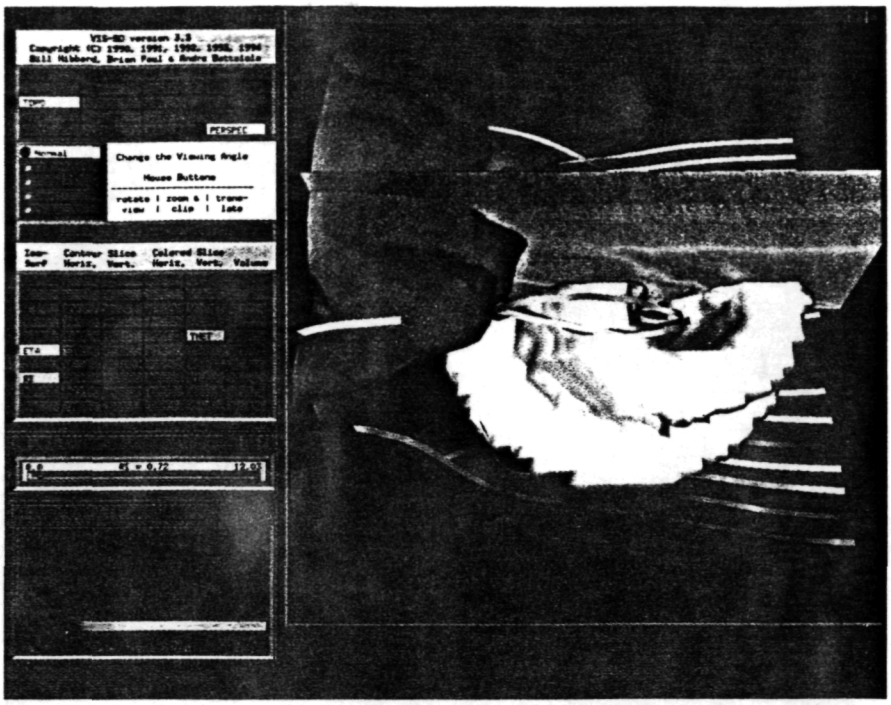

Figure 2. Simulation of William Gray's novel idea to generate energy by creating a permanent rainstorm over a hydroelectric generator.

Figure 3. Volume rendering of current speed from a simulation of the Pacific Ocean. The model's high resolution lets users see currents and eddies.

System availability

Vis-5D is available at no charge by anonymous ftp from iris.ssec.wisc.edu (144.92.108.63) in the pub/vis5d directory. The README file contains complete instructions for retrieving and installing the software. The system includes source code and documentation.

Simon Baas and Hans de Jong of Rijks Universiteit, Leiden, modified the Vis-5D source code so that on machines without special graphics hardware it can run under the X Window System (that is, without the GL library for 3D graphics) and, thus, on a wide class of workstations. We enhanced their work and include it as an option with our standard ftp distribution. We are also adding support to Vis-5D for various non-Cartesian map projections and vertical coordinate systems.

Although there is great interest in data format standards, the scientific community is only starting to adopt them. Thus, we have found that the most important element for making our system usable has been a set of data import utilities. These include template programs (with versions in both C and Fortran) to help users convert their history files into a form that our software can read. Users can modify these template programs to read their own history file formats.

Vis-AD is also available by anonymous ftp from iris.ssec.wisc.edu. It is located in the pub/visad directory. Again, see the README file in that directory for complete information.

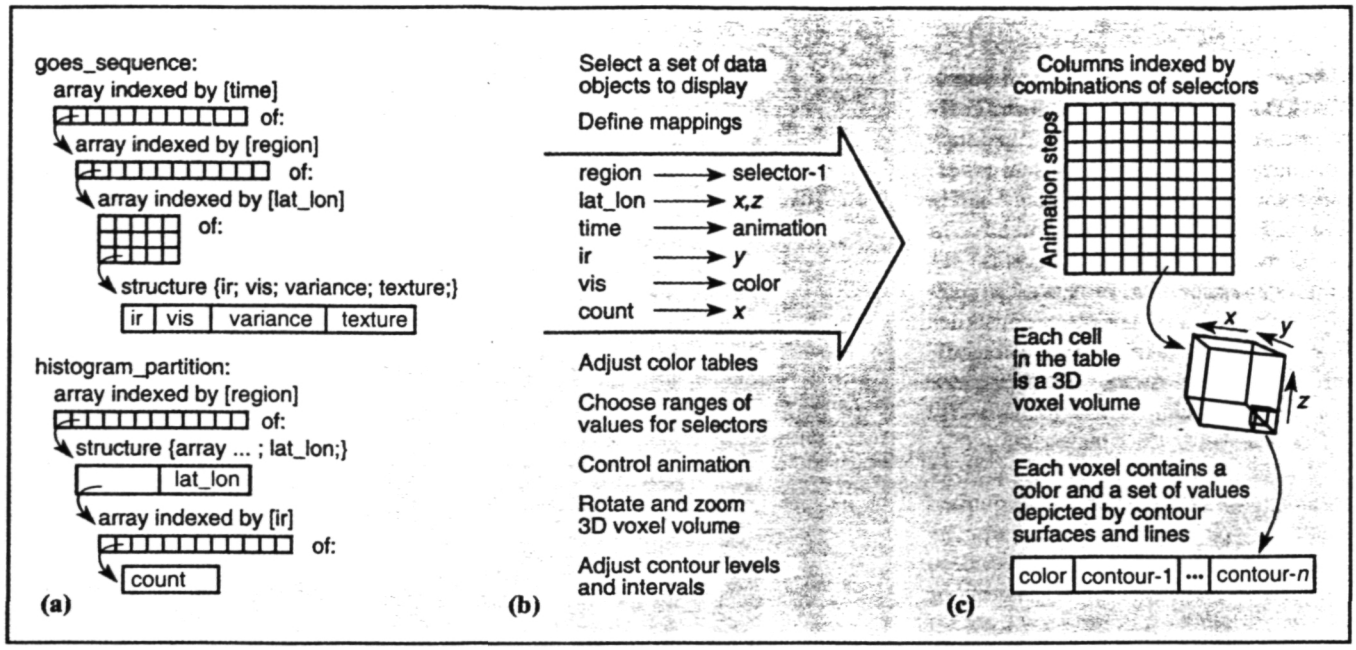

Figure 4. The Vis-AD data and display models: (a) examples of data types that users can define in the data model; (b) user interface for controlling scalar mappings that control how data are depicted in the display model; (c) a diagram of the voxel-based display model.

debugger prints values of variables and arrays to help users track down low-level program bugs, Vis-AD generates visualizations of complex data objects to help scientists understand the high-level behavior of their algorithms. Coupled with a graphical interface for steering computations, these visualizations enable a wide variety of visual experiments with algorithms.

Designing data types for scientific algorithms. To scientists designing algorithms, the data model appears as a set of three rules for designing data types appropriate to their algorithms. Scientists can

(1) Define a scalar type for a primitive variable. Scalar types may be real variables like time or *ir* (an infrared radiance); they may be pairs or triples of real numbers like lat_lon (a pair of real numbers for the latitude and longitude of an Earth location); they may be integers like count (a frequency count used in a histogram); or they may be text strings like satellite_id.
(2) Define a data type as a tuple of values of other types (this is like a structure in the C programming language).
(3) Define an array type as a finite sampling of a functional relation from one type (the domain of the function — this must be a scalar type) to another type (the range of the function — this can be a complex type). An array data object is a finite set of objects of the range type indexed by values of the domain type.

Arrays and tuples can be combined in hierarchies to build complex data types. The left-hand column of Figure 4 shows how the rules can be applied to define two data types appropriate for a cloud discrimination algorithm. A data object of the goes_sequence type is a time sequence of satellite images, each partitioned into a set of rectangular regions. The image in each region is an array of pixels indexed by lat_lon values (the latitudes and longitudes of the pixels' Earth locations). Each pixel is a tuple containing *is* (visible) and *ir* (infrared) radiances, as well as variance and texture values computed by the algorithm from ir radiances. A data object of the histogram_partition type is an array of histograms computed by the algorithm, one in each image region, specifying frequency counts for the ir radiances of pixels in the region. Calculation of histograms is an important step in the cloud discrimination algorithm, and displays of these histograms are very useful for tracking down problems with the algorithm.

The center column shows how users can control the displays of complex data objects by mapping scalar types to the components of the voxel-based display model diagrammed in the right-hand column. That is, users define mappings from scalar types to the x, y, and z coordinates of voxels in the display, to the colors of voxels, to animation step number, and so on. Because complex data types are ultimately defined in terms of scalar types, the system can derive depictions for complex data types from the mappings defined for their scalar components.

Figure 5 shows a data object of type goes_sequence displayed according to four different frames of reference. Its top right window shows the data object displayed as a colored terrain, as defined by the examples of scalar mappings in the center column of Figure 4. In the top left window, both ir (red) and vis (blue-green) radiances are mapped to color. In the bottom right window, ir is mapped to selector (only pixels whose ir radiances fall in the selected range are visible), and time is mapped to the vertical axis, producing a stack of four images. In the bottom left window ir, vis, and variance are mapped to the three axes and texture is mapped to color, producing a colored 3D scatter diagram (lat_lon is not mapped).

These displays are highly interactive. Users can rotate and zoom displays using the mouse, animate them, interactively adjust the mapping of scalars to color using a color-map icon, and change the subsets of data objects selected for display using slider icons. The voxel-based display model fits naturally with volume rendering techniques,[6] and as graphics speeds improve we will extend the display model to include transparency

and reflectivity values at each voxel. We will also add vector values at each voxel to provide a model for flow-rendering techniques.

Figure 6 illustrates the system's overall user interface via its application to a simple bubble-sort algorithm. The window on the left is used to edit the text of the sort program. This program is written in an interpreted language (the syntax for user-defined data types is part of this language). Scientists' programs can call functions written in C or Fortran, including those running remotely across a network. Users can start and stop their programs, set breakpoints by clicking on program lines, and execute single steps. They can also connect program values to graphical icons for interactively steering their computations. The dark horizontal bar across the program window indicates the current line of execution, and the short dark strings are the names of data objects selected for display. Users select data objects by clicking on their names, and their depictions appear in the window on the right. The scalar mappings that define a display frame of reference are edited in the small text window at the top of the screen. The system can display data in several different frames of reference simultaneously (Figures 5 and 9 show multiple frames of reference).

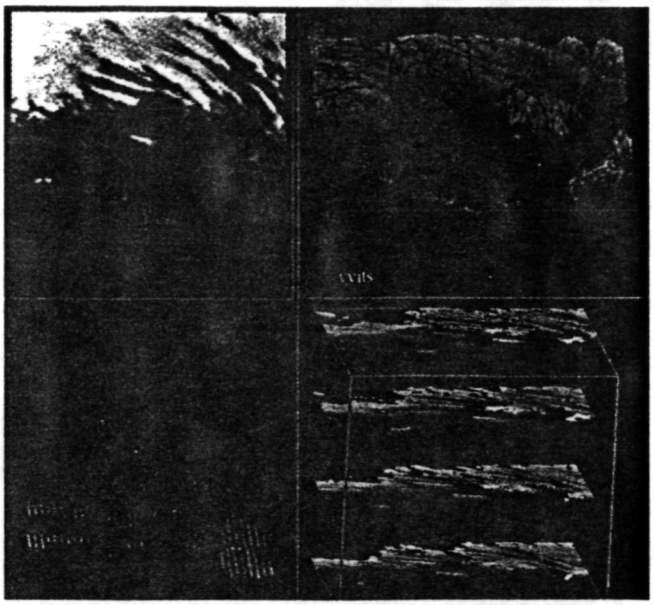

Figure 5. A time sequence of satellite images displayed in four different frames of reference.

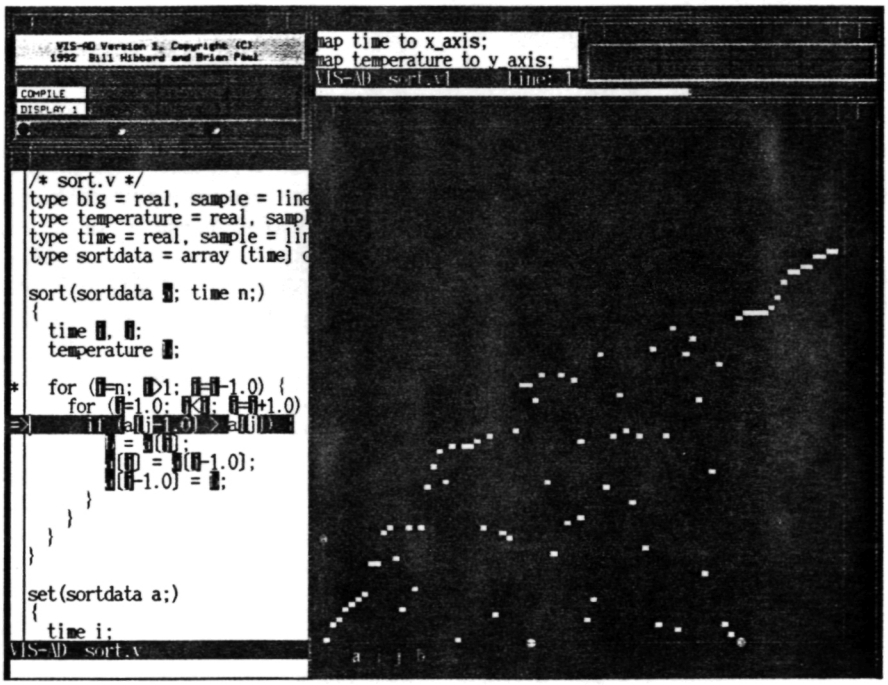

Figure 6. Visualizing the computations of a bubble-sort algorithm.

The data object being sorted in Figure 6 is an array of temperatures indexed by time. We have mapped time to the horizontal axis and temperature to the vertical axis, so the array is displayed as a graph (the set of white points) of temperature versus time. The bubble-sort algorithm is organized as two nested loops. The index of the outer loop has type time and is displayed as a small green sphere on the lower horizontal axis (note that the white points to the right of the green sphere are sorted). The index of the inner loop also has type time and is displayed as a small red sphere; it marks the horizontal position of the current maximum value bubbling up through the array. The small blue sphere on the left-hand vertical axis depicts an object of type temperature used as a temporary variable for swapping array values.

Integrating metadata into the data model. Mathematical models define infinite-precision real numbers and functions with infinite domains, whereas computer data objects contain finite amounts of information and must therefore be approximations of the mathematical objects they represent. Several forms of scientific metadata serve to specify how computer data objects approximate mathematical objects, and we have integrated these into our data model. For example, missing data codes (used for fallible sensor systems) can be viewed as approximations that carry no information. Any value or subobject in a Vis-AD data object can be set to the missing value. Scientists often use arrays for finite samplings of continuous functions, as, for example, satellite image arrays are finite samplings of continuous radiance fields. Sampling metadata, such as those that assign Earth locations to pixels and real radiances to coded (for example, 8-bit) pixel values, quantify how arrays approximate functions and are integrated with Vis-AD array data objects.

The integration of metadata into our data model has practical consequences for the semantics of computation and display. For example, we define a data type goes_image as an array of ir radiances indexed by lat_lon values. Arrays of this

ORIGINAL PAGE IS
OF POOR QUALITY

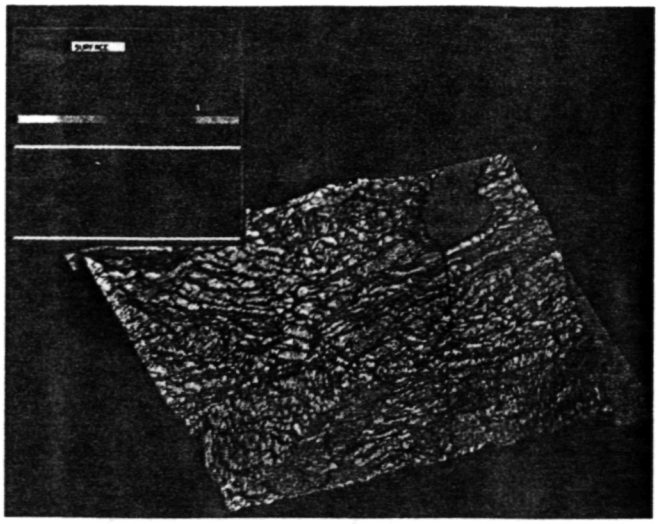

Figure 7. Percentage of cumulus clouds derived from satellite data, mapped onto a Midwest topography.

data type are indexed by pairs of real numbers rather than integers. If goes_west is a data object of type goes_image and *loc* is a data object of type lat_lon, the system evaluates the expression goes_west[loc] by picking the sample of goes_west nearest to loc. If loc falls outside the region of the Earth covered by goes_west pixels, goes_west[loc] evaluates to the missing value. If goes_east is another data object of type goes_image generated by a satellite with a different Earth perspective, then the expression

goes_west – goes_east

is evaluated by resampling goes_east to the samples of goes_west (that is, by warping the goes_east image) before subtracting radiances. In Earth regions where the goes_west and goes_east images do not overlap, their difference is set to missing values. Thus, metadata about map projections and missing data contributes to the semantics of computations.

Metadata similarly contributes to display semantics. If we have selected both goes_east and goes_west for display, the system uses the sampling of their indices to coregister these two images in a common Earth frame of reference. The samplings of 2D and 3D array indices need not be Cartesian. For example, the sampling of lat_lon may define virtually any map projection. Thus, we can display data in non-Cartesian coordinate systems.

Visualizing analyses of satellite observations. A pair of Geostationary Operational Environmental Satellites (GOES) located at eastern and western stations over the US generate one 1,024 × 1,024 image every 4 seconds. NASA's Earth Observing System, as planned, will generate about five 1,024 × 1,024 images per second. These data volumes are too large to be understood by direct visualization. Thus, the proper role of visualization for satellite observations is helping scientists to develop algorithms for automating their analysis.

Robert Rabin et al. of the National Severe Storms Laboratory, working at the University of Wisconsin-Madison, have developed algorithms for analyzing cumulus clouds in GOES images.[7] These algorithms identify which pixels are part of cumulus clouds and calculate a seasonal percentage of cumulus cloud cover as a function of time of day and location. The results of this computation are called a cloud census. We designed a census_image data type as an array of pixels indexed by lat_lon, where each pixel is a tuple containing a cumulus_percent and a topography value (elevation of the Earth's surface above sea level). The cloud census is stored in an object of the census_sequence type, defined as an array of census_image data objects indexed by time. Figure 7 is a census_sequence data object displayed in a frame of reference defined by mapping lat_lon to the x-z plane, mapping topography to the y axis, mapping cumulus_percent to color, and mapping time to animation. (Note Lake Michigan in the upper right corner of the image.) The color map icon in the upper left corner shows that we have chosen yellow for low cumulus percentages and blue for higher percentages. This display shows a clear correlation between cumulus percentage and topography, and when animated helps us to understand how cumulus clouds develop during the day. Since ignorance of the mechanics of cloud formation is a major cause of uncertainty in efforts to predict the climatic consequences of the increase in greenhouse gases, such understanding may have important long-term consequences.

Visualizing analyses of astrophysical observations. Because of the flexibility of its data and display models, Vis-AD is not limited to image processing applications. Figure 8 was generated from an algorithm for processing observations from an astrophysics mission. The Diffuse X-ray Spectrometer flew on the space shuttle in January 1993 and recorded several million events, each potentially an observation of an X ray emanating from interstellar gas.[8] However, most of the recorded events are spurious, so Wilton Sanders and Richard Edgar of the University of Wisconsin-Madison needed to develop an algorithm for identifying valid events. For this algorithm, we defined the xray_event data type as a tuple containing scalars for an event's time, wavelength, longitude, pulse_height, position_bin, goodness_of_fit, occulted_flag, and many other fields. We also defined a data type event_list as an array of xray_event tuples indexed by event_number. The figure shows a data object of the event_list type, displayed in a frame of reference defined by mapping longitude, wavelength, and time to the three axes, by mapping pulse_height to color, and by mapping position_bin and goodness_of_fit to selector. Each X-ray event is displayed as a colored dot. Slider icons in the upper right corner are used to select ranges of values for position_bin and goodness_of_fit, so that only those events whose field values fall in the selected ranges are displayed. This provides an easy way to experiment with event selection criteria.

To ferret out the mechanisms that produced spurious events, we defined many different frames of reference to see correlations among various sets of event fields. We also displayed the distribution of events as functions of various fields in the form of 1D and 2D histograms. Our ability to change the display mappings of scalars as easily as we could rotate images was a key to successfully understanding the sources of spurious events.

Visualizing computations for education. Figure 9 was generated from a simple simulation of a 2D cell of atmosphere. The dynamics of this cell are governed

ORIGINAL PAGE IS
OF POOR QUALITY

by a system of three differential equations developed by E.N. Lorenz[9] to study turbulence. Roland Stull chose to use this 2D simulation in his course on atmospheric turbulence at the University of Wisconsin-Madison. The right window shows wind streamlines (isolines of the "stream function") and temperatures (warm air is red and cool air is blue) in the 2D cell of atmosphere. The lower left window shows the solution to Lorenz's equations as a path through a 3D phase space, revealing the two lobes of the familiar Lorenz attractor. The upper left window shows this same path in two phase-space dimensions versus time, illustrating the apparently random (that is, chaotic) temporal distribution of alternations between the two-phase-space lobes. The state of the 2D atmosphere in the right window corresponds to a single blue point overlaid on the red phase-space path in the lower left window. As the simulation algorithm runs, these displays of changing data objects animate the relation between the changing 2D atmosphere and the blue point moving along the phase space path, showing that the two lobes of the Lorenz attractor in phase space correspond to clockwise and counterclockwise rotation in the 2D cell of atmosphere.

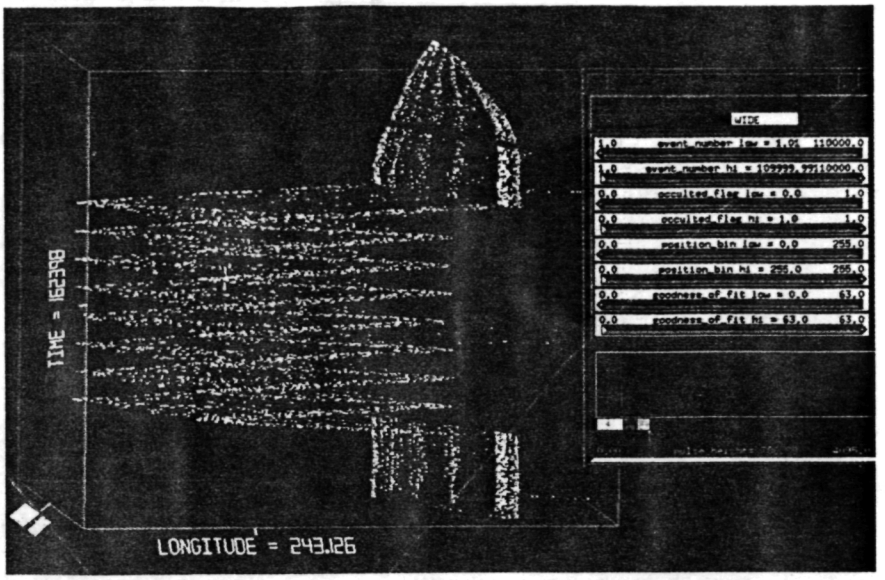

Figure 8. X-ray events from a 1993 Diffuse X-ray Spectrometer flight on the space shuttle.

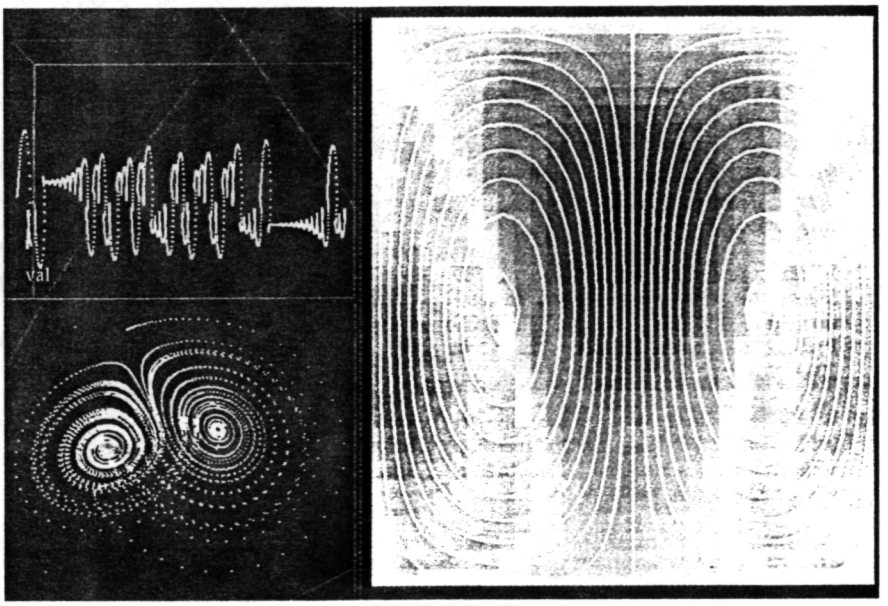

Figure 9. Three views of the chaotic dynamics of the Lorenz equations.

Comparisons with other techniques

The dataflow technique — represented by AVS (Application Visualization System), Iris Explorer, and Data Explorer — gives users the flexibility to design their own rendering pipelines as networks of basic modules. Although we recognize the value of this approach, we designed Vis-5D with a fixed rendering pipeline (diagrammed in Figure 1), which we felt could meet the needs of atmosphere and ocean modelers without asking them to design a module network. In fact, Vis-5D denies many choices to its users (for example, shading model parameters, and colors and locations of light sources) to keep its user interface simple.

Because it interprets arrays as finite samplings of functional relations, the Vis-AD data model is similar to the data models of Data Explorer and SuperGlue, which are based on fiber bundles. However, not all data models based on fiber bundles support complex hierarchies of tuples and functional relations, as the Vis-AD data model does. Vis-AD's scalar mappings define display functions that can be applied to any data type. This is similar to the polymorphic display functions defined in object-oriented systems like SuperGlue and Visage. However, users of object-oriented systems define display functions in a programming language, whereas users of Vis-AD define display functions by sets of scalar mappings. Just as the dataflow systems define a user interface for controlling data display based on the abstraction of the rendering pipeline, the Vis-AD system defines a user interface for controlling data display based on the abstraction of mappings from scalars to display scalars.

Interactive visualization techniques are making a difference in the work of scientists who have the means and who make the effort to use them. We have exploited special assumptions about data organization to make it easy for scientists to apply Vis-5D to their data. The result is a system that is widely used by atmosphere and ocean modelers.

Scientists have needs that do not fit the

ORIGINAL PAGE IS
OF POOR QUALITY

by a system of three differential equations developed by E.N. Lorenz[9] to study turbulence. Roland Stull chose to use this 2D simulation in his course on atmospheric turbulence at the University of Wisconsin-Madison. The right window shows wind streamlines (isolines of the "stream function") and temperatures (warm air is red and cool air is blue) in the 2D cell of atmosphere. The lower left window shows the solution to Lorenz's equations as a path through a 3D phase space, revealing the two lobes of the familiar Lorenz attractor. The upper left window shows this same path in two phase-space dimensions versus time, illustrating the apparently random (that is, chaotic) temporal distribution of alternations between the two-phase-space lobes. The state of the 2D atmosphere in the right window corresponds to a single blue point overlaid on the red phase-space path in the lower left window. As the simulation algorithm runs, these displays of changing data objects animate the relation between the changing 2D atmosphere and the blue point moving along the phase space path, showing that the two lobes of the Lorenz attractor in phase space correspond to clockwise and counterclockwise rotation in the 2D cell of atmosphere.

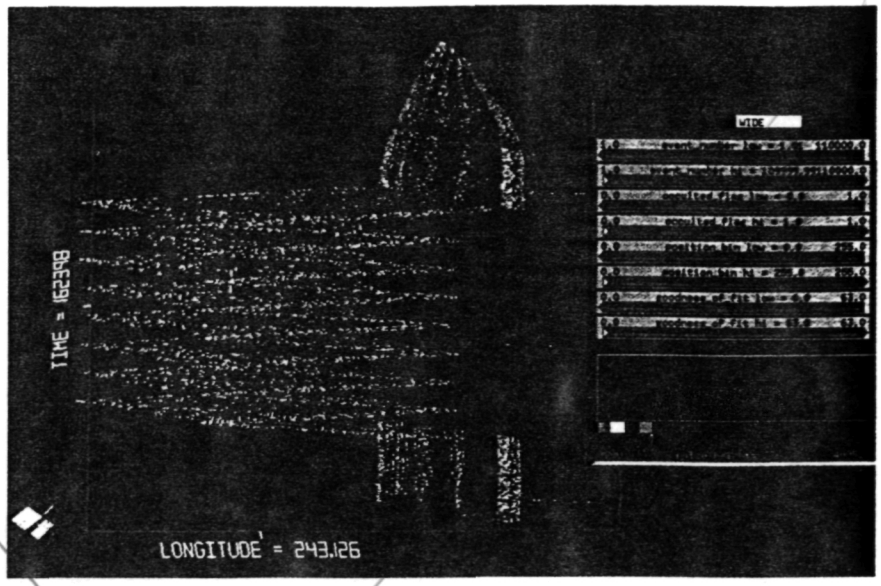

Figure 8. X-ray events from a 1993 Diffuse X-ray Spectrometer flight on the space shuttle.

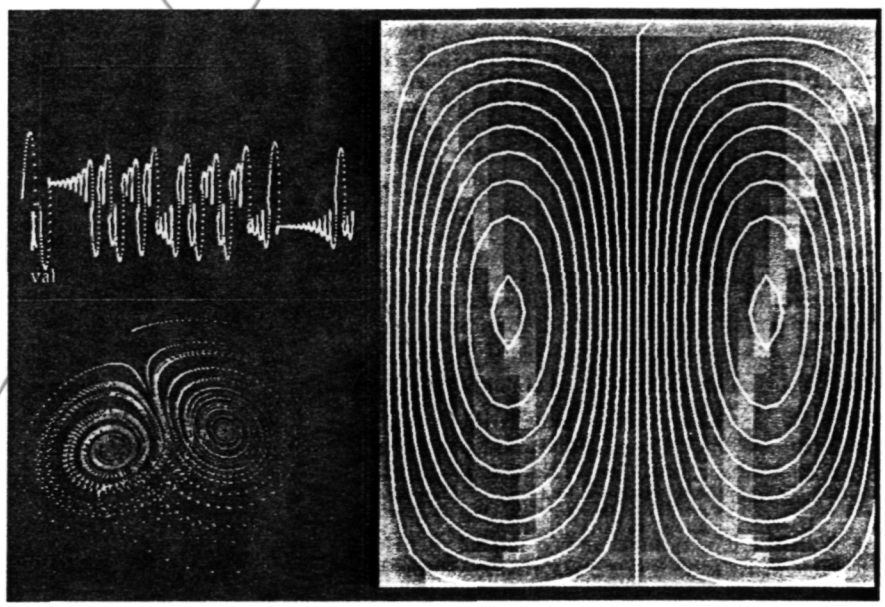

Figure 9. Three views of the chaotic dynamics of the Lorenz equations.

Comparisons with other techniques

The dataflow technique — represented by AVS (Application Visualization System), Iris Explorer, and Data Explorer — gives users the flexibility to design their own rendering pipelines as networks of basic modules. Although we recognize the value of this approach, we designed Vis-5D with a fixed rendering pipeline (diagrammed in Figure 1), which we felt could meet the needs of atmosphere and ocean modelers without asking them to design a module network. In fact, Vis-5D denies many choices to its users (for example, shading model parameters, and colors and locations of light sources) to keep its user interface simple.

Because it interprets arrays as finite samplings of functional relations, the Vis-AD data model is similar to the data models of Data Explorer and SuperGlue, which are based on fiber bundles. However, not all data models based on fiber bundles support complex hierarchies of tuples and functional relations, as the Vis-AD data model does. Vis-AD's scalar mappings define display functions that can be applied to any data type. This is similar to the polymorphic display functions defined in object-oriented systems like SuperGlue and Visage. However, users of object-oriented systems define display functions in a programming language, whereas users of Vis-AD define display functions by sets of scalar mappings. Just as the dataflow systems define a user interface for controlling data display based on the abstraction of the rendering pipeline, the Vis-AD system defines a user interface for controlling data display based on the abstraction of mappings from scalars to display scalars.

Interactive visualization techniques are making a difference in the work of scientists who have the means and who make the effort to use them. We have exploited special assumptions about data organization to make it easy for scientists to apply Vis-5D to their data. The result is a system that is widely used by atmosphere and ocean modelers.

Scientists have needs that do not fit the

special assumptions of Vis-5D, so we developed the Vis-AD system by generalizing some of the concepts of Vis-5D. Because of its flexibility, this system confronts its users with complex choices. However, we have organized these choices in a consistent framework of data, display, and computational models. Vis-AD has demonstrated its utility to scientists working with its developers. When we complete its documentation and online help functions, we are confident that it will be useful to a wide community of scientists. ■

Acknowledgments

This work was supported by NASA Grant NAG8-828, and by the National Science Foundation and the Advanced Research Projects Agency under Cooperative Agreement NCR-8919038 with the Corporation for National Research Initiatives.

References

1. B. McCormick, T. DeFanti, and M. Brown, "Visualization in Scientific Computing," *Computer Graphics*, Vol. 21, No. 6, Nov. 1987.

2. W. Hibbard and D. Santek, "The VIS-5D System for Easy Interactive Visualization," *Proc. Visualization 90*, IEEE CS Press, Los Alamitos, Calif., Order No. 2083, 1990, pp. 28-35.

3. W. Hibbard and D. Santek, "Interactivity is the Key," *Proc. Chapel Hill Workshop Volume Visualization*, Univ. of North Carolina, Chapel Hill, 1989, pp. 39-43.

4. W. Hibbard, D. Santek, and G. Tripoli, "Interactive Atmospheric Data Access Via High-Speed Networks," *Computer Networks and ISDN Systems*, Vol. 22, No. 2, Sept. 1991, pp. 103-109.

5. W. Hibbard, C. Dyer, and B. Paul, "Display of Scientific Data Structures for Algorithm Visualization," *Proc. Visualization 92*, IEEE CS Press, Los Alamitos, Calif., Order No. 3090-02, 1992, pp. 139-146.

6. A. Kaufman, D. Cohen, and R. Yagel, "Volume Graphics," *Computer*, Vol. 26, No. 7, July 1993, pp. 51-64.

7. R.M. Rabin et al., "Observed Effects of Landscape Variability on Convective Clouds," *Bull. Am. Meteorological Soc.*, Vol. 71, No. 3, Mar. 1990, pp. 272-280.

8. W.T. Sanders et al., "Preliminary Results from the Diffuse X-ray Spectrometer," *EUV, X-ray, and Gamma-ray Instrumentation for Astronomy IV*, Soc. Photooptical Instrumentation Engineers, Vol. 2,006, July 1993, pp. 221-232.

9. E.N. Lorenz, "The Mechanics of Vacillation," *J. Atmospheric Science*, Vol. 20, Sept. 1963, pp. 448-464.

William L. Hibbard is a researcher at the Space Science and Engineering Center of the University of Wisconsin-Madison. He is the principal investigator under the NASA grant that supported the development of the Vis-5D and Vis-AD systems and has been an investigator of the Blanca Gigabit Testbed. Hibbard has been a member of the program committees for the IEEE Visualization Conferences from 1990 to 1994.

He received a BA in mathematics in 1970 and an MS in computer science in 1974, both from the University of Wisconsin-Madison.

Brian E. Paul is a computer scientist at the Space Science and Engineering Center. His research interests include computer graphics and programming languages.

He received his BS in computer science at the University of Wisconsin-Oshkosh and is pursuing his master's degree at UW-Madison. He is a member of the ACM.

David A. Santek is a team leader in the scientific applications area at the Space Science and Engineering Center. His research interests include satellite data analysis, image analysis, and computer graphics.

He received a BS in atmospheric and oceanic science from the University of Michigan in 1975 and an MS in meteorology from the University of Wisconsin in 1978.

Charles R. Dyer is a professor in the Department of Computer Sciences at the University of Wisconsin-Madison. His research interests include computer vision, robotics, and visualization.

He received his BS degree from Stanford, his MS from the University of California at Los Angeles, and his PhD from the University of Maryland in 1979. He is on the Advisory Board of *IEEE Transactions on Pattern Analysis and Machine Intelligence* and is program cochair of the 1996 IEEE Conference on Computer Vision and Pattern Recognition.

André Luis Battaiola is a researcher at the Instituto Nacional de Pesquisas Espaciais (National Institute of Space Research) in Brazil. His research interests are computer graphics and scientific visualization.

He received his BS in physics and an MS and a PhD in electrical engineering from the University of Sao Paulo, Brazil, in 1982, 1987, and 1992, respectively.

Marie-Françoise Voidrot-Martinez is a computer scientist at Meteo-France. Her research interests include interactive computer graphics and user-interface design. She received a degree in meteorology from the French National Meteorological School in 1986 and a master's degree in computer science from the school of Centrale Paris in 1989.

Readers can contact William Hibbard at the Space Science Engineering Center, University of Wisconsin-Madison, 1225 W. Dayton St., Madison, WI 53706; e-mail whibbard@macc.wisc.edu.

Copyright © 1994 The Institute of Electrical and Electronics Engineers, Inc.
Reprinted with permission from COMPUTER,

Meteorol. Atmos. Phys. 49, 229–254 (1992)

Meteorology and Atmospheric Physics

© Springer-Verlag 1992
Printed in Austria

Department of Meteorology, University of Wisconsin-Madison, Madison, Wisconsin, U.S.A.

An Explicit Three-Dimensional Nonhydrostatic Numerical Simulation of a Tropical Cyclone

G. J. Tripoli

With 12 Figures

Received January 31, 1992
Revised June 28, 1992

Summary

A nonhydrostatic numerical simulation of a tropical cyclone is performed with explicit representation of cumulus on a meso-β scale grid and for a brief period on a meso-γ scale grid. Individual cumulus plumes are represented by a combination of explicit resolution and a 1.5 level closure predicting turbulent kinetic energy (TKE).

The results demonstrate a number of expected and unexpected important scale interaction processes. Within the central core of the developing cyclone, meso-β convective regions grow and breakdown into propagating inertia-gravity waves throughout the lifecycle of the cyclone. In the early stages, the amplitude of pressure fluctuations associated with the meso-β scale convection exceed the central pressure of the cyclone and strongly modulate its intensity. With each meso-β scale pulsation, the cyclone core increases in strength, measured by the central pressure deficit. The increasingly strong inertial frequency of the storm core acts to increasingly trap the convection induced heating within the core by balancing the tangential wind against the low central pressure, before the meso-β scale convection breaks down and sends the warmth away as a propagating wave. Eventually, the slow manifold's amplitude exceeds the amplitude of the meso-β scale oscillations and a stable eye region is formed. As inertial instability increases, increasingly high thermal warmth can be protected in the core, allowing persistent subsidence to form and to clear out the cyclone eye.

On the outside of the eye wall, strong inertial stability gradients in the troposphere cause convective warming to split the inflow to the eye wall and spawn outwardly propagating inertia gravity waves. These waves carry away all of the heating forced by convection that is not inertially trapped by the eye wall and act as a moderating influence on storm intensity.

Inertia gravity waves are also spawned in the stratosphere at the top of the eye wall by the revolution of asymmetric cumulus structures. In all instances, the tropospheric waves are coupled to the propagating stratospheric waves which both move at $35\,ms^{-1}$, although there are many instances where the stratospheric waves seem to have no tropospheric counterpart. Hence the anvil top forcing and low level breakdown are linked.

The outwardly propagating inertia gravity waves act to initiate outer bands of convection. This initiation is with the assistance of low level boundary layer variations of density related to previous convection and to virga falling from the anvil which moistens and destabilizes the mid levels of θ_e minimum. The convection initiated by these waves does not move substantially outward with the wave, although may appear to develop outward discontinuously.

1. Introduction

The notion of CISK (Conditional Instability of the Second Kind) was first described by Charney and Elliasen (1964) as a way to explain the growth of a hurricane from a weak tropical depression. The underlying principle is that an ensemble of deep precipitating cumulus result in deep tropospheric heating which lowers the surface pressure within the storm, increasing the tangential winds of the balanced vortex. The friction layer induces radial inflow which, in turn, forces the convergence of warm moist marine layer flow into the storm to fuel the convection. The critical link in this chain of events was the relationship between

PRECEDING PAGE BLANK NOT FILMED

the surface convergence and the heating function which is the role played by deep cumulus convection.

Early attempts to model this interaction by Ooyama (1964) were thwarted by the existence of nonlinear instability. Charney and Elliasen (1964) were successful in achieving vortex growth because they introduced the concept of *cumulus parameterization* which attempted to represent the effects of cumulus without explicitly resolving cumulus circulations. Their success together with Ooyama's experiences led to massive observational and parameterization development efforts in the community to accurately define the heating function hoping that tropical cyclone genesis could be accurately predicted. Parameterization theory spread to other areas of meteorological prediction and theoretical analysis where cumulus were deemed important, such as general circulation theory (Arakawa and Schubert, 1974), wave CISK theory (Lindzen, 1974; Raymond, 1975; Raymond, 1976), mesoscale convective systems (Fritsch and Chappel, 1980; Kreitzberg and Perky, 1976) and so on. In all applications, the use of cumulus parameterization schemes to represent scale interaction has resulted in little or no increase in the knowledge of the processes they were designed to represent. As model resolution increases to resolve the meso-β scale, the underlying cumulus parameterization assumption of scale separation breaks down.

The development of three dimensional cumulus/mesoscale models in the mid 1970s (Klemp and Wilhelmson, 1978; Cotton and Tripoli, 1978; Clark, 1977) led to massive breakthroughs in the 1980s in the understanding of cumulus scale overturning and gravity wave behavior. At the same time, scientists were beginning to recognize important aspects of the CISK process beyond the effects of Ekman pumping which were highly dependent upon the structures of the small circulations within the cumulus ensemble. For instance, gravity waves were proposed as a major component of spiral bandedness in tropical cyclones (Kurihara, 1976; Willoughby, 1977; Willoughby, 1978; Willoughby, 1979), the degree to which inertia gravity waves were spawned by deep convection was shown to strongly modulate the cyclone growth rate (Schubert and Hack, 1982), and the existence of asymmetric wave-like features in the cyclone circulation were shown to significantly affect radial momentum transport (Holland, 1983).

In order to build a scale interaction three-dimensional model of a tropical cyclone, all thermodynamical and dynamical processes must be represented simultaneously. Hydrostatic, axi-symmetric, and cumulus parameterized models are inadequate for this purpose because of the built-in biases and restrictions inherent in their design.

Since Rosenthal (1978) demonstrated that Ooyama's (1964) concern of nonlinear instability was somewhat overstated and that explicit convection models on the tropical cyclone scale were possible, there has been renewed interest in explicitly representing latent heat release rather than parameterizing it. With model resolutions allowed under present computing constraints in three dimensions, it is still not quite possible to realistically resolve the true scale of cumulus plumes. So the debate becomes one of whether the parameterized approach, with assumed subgrid scale cumulus ensembles with clouds of assumed simplistic structures, is more realistic or whether explicitly predicted convective overturning on the scale of mesoscale convective systems rather than individual plumes, is more realistic in a model of the entire cyclone.

In a relatively short while, that debate will become irrelevant because plausible model resolutions will dramatically increase so that cumulus can be explicitly resolved realistically. In this paper a nonhydrostatic tropical cyclone simulation will be presented which has sufficient grid resolution to explicitly resolve convective overturning on the full meso-β scale, and for a shorter period of simulation, a portion of the meso-γ scale near the eye wall. Unresolved convection was represented by a 1.5 level turbulence closure scheme.

The purpose of this paper is to explore the types of internal scale interactions which arise in a fully three-dimensional non-hydrostatic model of a tropical cyclone when cumulus are explicitly represented. As a consequence, it is desirable to keep the interaction with the initial state and the initial environment as simple as possible so the results can be easily understood. It is elected to simulate the growth of an axi-symmetric modified Rankine vortex perturbation in an initial undisturbed marine environment with thermodynamic structure typical for tropical cyclone development and without mean wind.

The next section will describe the numerical model employed briefly and will be followed by a section discussing the experimental design. Next the results will be presented for a 56 hour simulation period. This will be followed by a section which attempts to tie the results together to form a coherent picture of the scale analysis process as it can be modeled today. Finally, in section 6, conclusions concerning the implications of these results will be drawn.

2. Numerical Model

The numerical model used was the Tripoli (1992) nonhydrostatic mesoscale model. The model was based on the non-Boussinesq quasi-compressible dynamical equations. Model thermodynamics were calculated by integrating the enthalpy conserving θ_{il} along with the total water and several ice and liquid water precipitating hydrometeor specific humidities. The thermodynamic system was closed under the assumption of zero supersaturation over liquid which also defined diagnostic relationships for vapor and cloud water. For these simulations, it was assumed that the entire grid lies over a body of water and so the model's terrain following coordinate system was not implemented. A spherical horizontal coordinate transformation was implemented for this simulation.

This numerical experiment utilized no cumulus parameterization despite its meso-β scale horizontal resolution over the first 50 hours of simulation. In this way wave scale interactions generated by the convective motions could be simulated explicitly, although at times these motions could significantly depart from realism due to deficiencies of the resolution. Some of the limitations of this approach were softened by utilizing a 1.5 level turbulence closure, where turbulent kinetic energy (TKE) was predicted using a modified form of the closure developed by Redelsperger and Someria (1981, 1982), and subsequently used to form the vertical down-gradient mixing terms. Some modifications were included to represent the effects of saturation, ice phase, and precipitation loading on turbulence generation. A more complete explanation of the closure is given in Appendix A.

The 1.5 level closure enabled the mixing effects of cumulus to be transported by the explicitly resolved motions. Upscale transport, such as that caused by subgrid scale motions transporting subgrid scale fluctuations, and transilient turbulence (Stull, 1989) cannot be represented by the TKE closure used here. Nevertheless, it has been found that the simpleTKE closure in conjunction with partially resolved cumulus updrafts created realistic vertically oriented plumes of high TKE which significantly enhance explicitly predicted vertical transport locally. Also, horizontal advection of the TKE acted to spread the effects of cumulus mixing downstream, and in particular, around the eye wall. This can be significant since the time scale of turbulent dissipation was approched by the inertial frequency as the storm increases in strength. This will be shown in the discussion of results below.

The surface fluxes of moisture, sensible, and radiative heat were calculated assuming an ocean of uniform temperature 301 K. The surface layer parameterization was based on the Louis (1979) surface layer, with surface roughness specified as a function of wave height (Delsol et al., 1971).

Horizontal mixing was based on a deformation based 2nd order eddy viscosity closure described by Tripoli and Cotton (1982) with a background fourth order diffusion as described by Klemp and Wilhelmson (1978). The 1.5 level closure was deemed inappropriate for the horizontal because the issues controlling turbulence were more closely tied to numerical as well as physical enstrophy cascade rather than subgrid scale dry and moist convective processes which control vertical diffusion.

Radiative transfer in a cloudy atmosphere was predicted with the radiation parameterization developed by Chen and Cotton (1983). A modified form of the Cotton et al. (1986) (hereafter referred to as CEA) explicit microphysics prediction scheme was employed. The scheme predicts rain, graupel, pristine crystals, and snow crystals. Major modifications made were to:

(1) Divide the original pristine ice category into a snow and pristine category. The modified pristine category was assumed to be composed of newly nucleated hexagonal plate crystals of uniform mass 1.5×10^{-12} kg. This size was determined in separate axisymmetric simulations of a tropical cyclone with explicitly predicted pristine crystal sizes. The new snow category was assumed to follow a Marshall–Palmer

distribution with slope and intercept derived from an explicitly predicted number concentration per unit mass, similar to the number concentration prediction of CEA. The snow was assumed to have grown from its nucleation size and to be somewhat rimed.

(2) Aggregated crystals were assumed to be part of the snow category. Aggregation of snow and pristine crystals was represented by appropriate transfers of mass to snow from pristine crystals and appropriate adjustments of the predicted snow concentration. The characteristic snow diameter was limited to not exceed 0.33×10^{-2} m, which was consistent with the Cotton et al. (1986) aggregate distribution slope.

The CEA model originally grouped both nucleated and new crystals together. Since a constant size distribution had been assumed, massive nucleation at cold temperatures would drastically alter the average crystal size and would remove all memory of the growth that some of the larger crystals had been through. Here, growth processes were assumed to convert only a given number of crystals at the specified mass to the snow category, which itself was logarithmically distributed. Hence, new and mature populations of crystals would continue to exist where massive nucleation occurred. This was especially important to the simulations of cirrus anvils.

(3) Graupel was represented with a constant slope Marshall–Palmer distribution, with a characteristic diameter of 2×10^{-4} m. This value was derived from explicit axi-symmetric simulations with predicted graupel mixing ratio and concentration where it was found to be characteristic of the graupel size predicted.

The numerical infra-structure of the model was based on a hybrid leap frog (dynamics) and forward (thermodynamics) time integration on advection with a semi-implicit time split approach to representing the high frequency pseudo-sound wave terms. An enstrophy conserving leap frog advection scheme was used for the integration of wind and pressure while a forward 6th order Crowley was used for the integration of the thermodynamical quantities.

A two-way multiply nested Arakawa "C" grid system was employed, using a form of the meshing technique employed by Tremback et al. (1990) and Clark and Farley (1984) modified for a hybrid time differencing scheme and to accommodate a fourth order smoothing operator in the grid center. The grid nesting scheme was also modified to allow any nest to move along a specified trajectory or to move with the surface pressure minimum.

As shown by Tripoli (1992), the model's mean mass field must be specified externally because of the quasi-compressible closure produces a pressure solution unique only to within a constant. This constant was determined by requiring the mean surface pressure across the outer domain to remain constant by making an adjustment to the mean exner function over the entire domain at every large grid time step. The adjustment was made to the exner function so that local hydrostatic balance was unaffected by the change. In addition to this adjustment, the outward and inward changes of mass flux predicted by the external boundary's radiation condition were forced to balance by reducing excessive inflow (or outflow) tendencies on a percentage basis.

3. Experiment Design

The prescribed initial horizontally homogeneous basic thermodynamic structure was taken from the temperature and humidity profiles observed at Kingston, Jamaica, 36 hours before the passage of Hurricane Gilbert. Initial mean winds were assumed to be zero.

The grid locations, spacings, lengths, timesteps and implementation period appear in Table 1. The outer grid was centered along the expected path of the modeled storm, while the medium grid was centered over the location of the initial perturbation.

The initial mean state was perturbed by a modified Rankine vortex described in Appendix B. The vortex was defined by its relative vorticity field which was set to decrease from 2×10^{-4} near the center, to slightly negative outside the center and then to increase to zero again. The vorticity field is adjusted so that the total area weighted negative vorticity exactly balances the positive vorticity at a radius of 800 km. Hence no net vorticity was added to the flow. The stream function field was formed from the relaxation of the vorticity field which was in turn used to calculate the wind field.

Table 1. *Two-Way Nested Grid Setup*

Grid	1	2	3	4
Period of activation	0–56 hr	0–56 hr	8–56 hr	50–56 hr
# Central latitude	17.5 N	16.5 N	16.5 N	18.5 N
# Central longitude	71.0 W	68.0 W	68.0 W	69.7 W
# Zonal boxes	64	60	64	60
Zonal spacing	60 km	20 km	10 km	3.3 km
Zonal domain length	3840 km	1200 km	640 km	198 km
# Meridional boxes	64	60	64	60
Meridional spacing	60 km	20 km	10 km	3.3 km
Meridional domain length	3840 km	1200 km	640 km	198 km
# Vertical boxes	42	42	42	42
Vertical spacing	400–800 m	400–800 m	400–800 m	400–800 m
Domain height	26 km	26 km	26 km	26 km
Large timestep	120 s	40 s	20 s	6.33 s
Small timestep	40 s	13.3 s	6.67 s	2.11 s

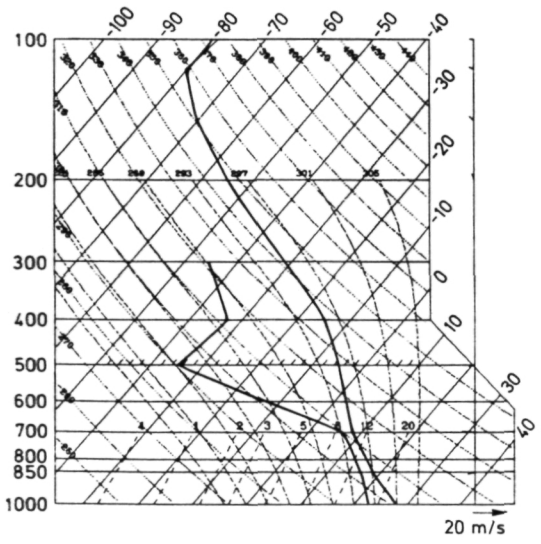

Fig. 1. Initial sounding observed at Kingston, Jamaica at 0000 UTC 11 September, 1988

The perturbation produced a maximum tangential wind of 11 ms^{-1} at the surface at a radial distance of 150 km from the vortex center.

Corresponding virtual potential temperature and pressure fields were then calculated using the nonlinear balance equation. The outer grid perturbation pressure field was first found using Neuman boundary conditions and then normalized to the observed sounding at the location of Kingston, Jamaica (Fig. 1). The potential temperature field was determined by assuming hydrostatic balance. The medium mesh was dynamically balanced using Dirchlet boundary conditions on pressure determined from the integration of the outer grid.

The third mesh was not implemented until 8 hours of integration in order to conserve computer resources during the period when the model was beginning to build an Ekman Layer. The fourth grid, with a 3.3 km horizontal resolution was implemented for a 6 hour period at the mature stage of the simulated tropical cyclone. The limited time of its use was both because of the high cost of its use (8 hours of Cray-ymp for 1 hour of simulation), and because the limited horizontal extent of the grid required that the eye wall be contained within a box 200 km on a side.

4. Results

4.1 Evolution of Structure

4.1.1 Initial Rapid Growth Phase – Transition to Tropical Storm

The simulation was initiated at 0000 LST. Because the initial modified Rankine vortex was specified to be nondivergent, the vortex must have developed a divergent wind component by frictional processes to initiate convective activity. In this section, the modeled transition from a balanced[1] and nondivergent vortex, to a developing tropical cyclone

[1] "Balanced" refers to the state where the wind and mass fields are linked to each other so that the time tendency for horizontal divergence vanishes, as dictated by the nonlinear balance equation.

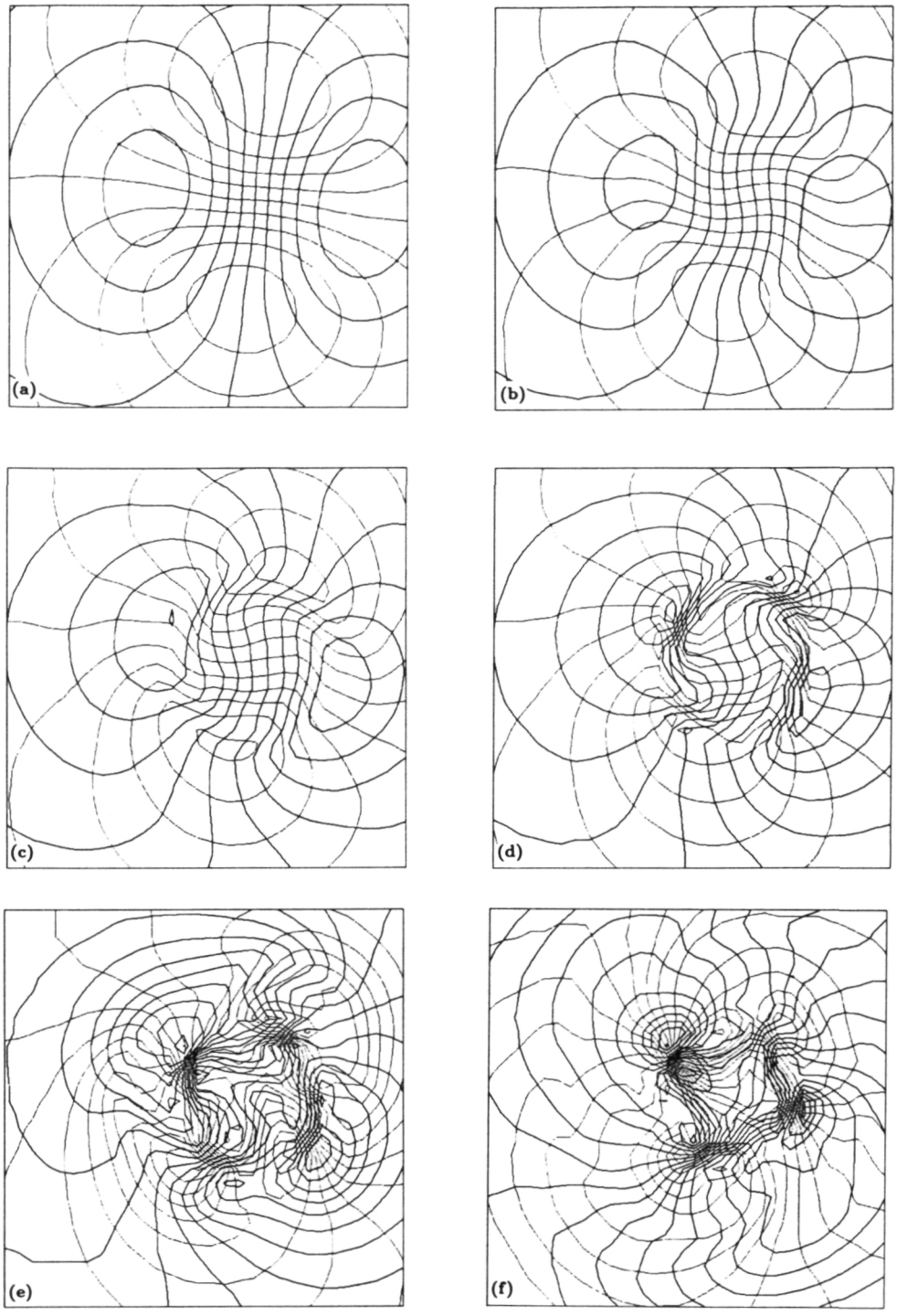

Fig. 2. Contours of zonal (u) and meridional (v) wind components at intervals of 2 ms^{-1} over a region 600 km across centered near the vortex core and viewed from the second mesh. Simulation time is (a) 5 hr, (b) 7 hr, (c) 8 hr, (d) 9 hr, (e) 10 hr and f) 11 hr of simulation

is described. Admittedly, the modeled transition was strongly dependent on the initial structure of the Rankine vortex, and such transitions in a real developing cyclone were likely to be strongly dependent on the nature of the initial perturbing disturbance and large scale forcing. It was however instructive to look at the modeled process of transition in order to gain some insight into the nature of the explicit interaction between a weakly inertially stable vortex and relatively strong, and

at times dominant, deep convection processes in three dimensions.

During the first 5 hours of simulation, surface friction gradually weakened the initial tangential wind in the lowest layer causing it to turn slightly inward. Figure 2a depicts the variation of the zonal and meridional wind components (u and v respectively) at 5 hours. The intersecting u and v contours formed a grid at the center of rotation. The u and v grid direction, if defined to be the direction of the central v component contour pointing toward negative u values, indicated the relative degrees of convergent and rotational flow. The initial u and v grid direction, with only a cyclonic rotational wind, pointed directly north. By five hours, a convergence was evident by the slight veering of the u and v grid direction.

By 7 hours, it was found that the surface u and v grid in the inner vortex had begun to back, indicative of the development of divergence in the vortex center. This resulted from a maximizing of surface convergence at a radius of 250–300 km which was roughly equivalent to the radius of peak velocity specified initially for the Rankine vortex. This forced a shallow upward circulation at the radius of maximum convergence, which subsequently forced mass inward, above the surface layer toward the vortex center. That, in turn, caused the surface pressure in the center to rise slightly, reducing the horizontal pressure gradient at the center. As a result, the flow inside the convergence ring became unbalanced and began to diverge outward, strengthening the convergence.

As the convergence sharpened, a circular ring of condensate formed from air being forced upward to the lowest condensation level (LCL). Peak vertical motions grew very slowly at this time before the moisture reached the level of free convection (LFC). By 8 hours, Fig. 2c showed an amplified convergence ring. As a result of inward angular momentum transport, peak winds had increased slightly and regained most of the speed lost to friction earlier.

Shortly after 9 hours (Fig. 2d) the forced cloud ring reached the LFC and the vertical velocities began to dramatically increase from magnitudes of centimeters per second to meters per second. The deepening convection warmed the air at mid to upper levels, and resulted in a more rapid lowering of the surface pressure which greatly enhanced the surface convergence into the ring and hence the vertical growth rate of the ring as well.

In general, because convection favors the smallest scales in both the radial and tangential directions, any asymmetric perturbation will likely initiate a tangential variation of the convection. In this otherwise symmetric initial perturbation, the only asymmetry, besides the beta effect, was the tangential variation of the finite differencing truncation error. This had a distinctly tangential wave number four variation due to the rectangular grid geometry. This very small tangential perturbation grew rapidly and by 10 hours (Fig. 2e), a distinct pattern of four dominant convective centers appeared, with some weaker meridional variation of intensity superimposed from the beta effect.

The role of convection in forcing surface convergence became dominant over the frictional effects by this time. As a result, the surface velocity fields converged into these convective systems to the point that the flow in the major vortex center actually became anticyclonic while the four convective centers each developed closed circulations of their own. In fact, at 10 hours, the surface lows beneath the convective systems reached a perturbation pressure (perturbation from the horizontal mean) of -5.5 hPa while the central pressure of the parent vortex pressure was -3.5 mb. This was 0.5 hPa greater than the initial surface pressure. Hence, the initial modified Rankine vortex had broken up into four separate meso-β scale convective systems.

Simulated precipitation formed within the individual convective systems and, as with any unbalanced convective system, maturity was reached and the system progressed into a breakdown phase (Tripoli and Cotton, 1989). This resulted in the formation of deep internal waves radiating from the center of the collapse. Phase speeds were calculated from animated sequences of the vertical motion field and found to be about $35\,\mathrm{ms}^{-1}$. The collapse also resulted in the rise of the surface pressure beneath the convective centers. Convergence, forced at upper levels within the parent vortex center, resulted in a strong surface pressure decrease to -11 hPa by 11 hours (Fig. 2f). Also evident were outward propagating waves in the u and v fields indicated by the wave like structures in the contours away from the center. Three dimensional visualization of these waves

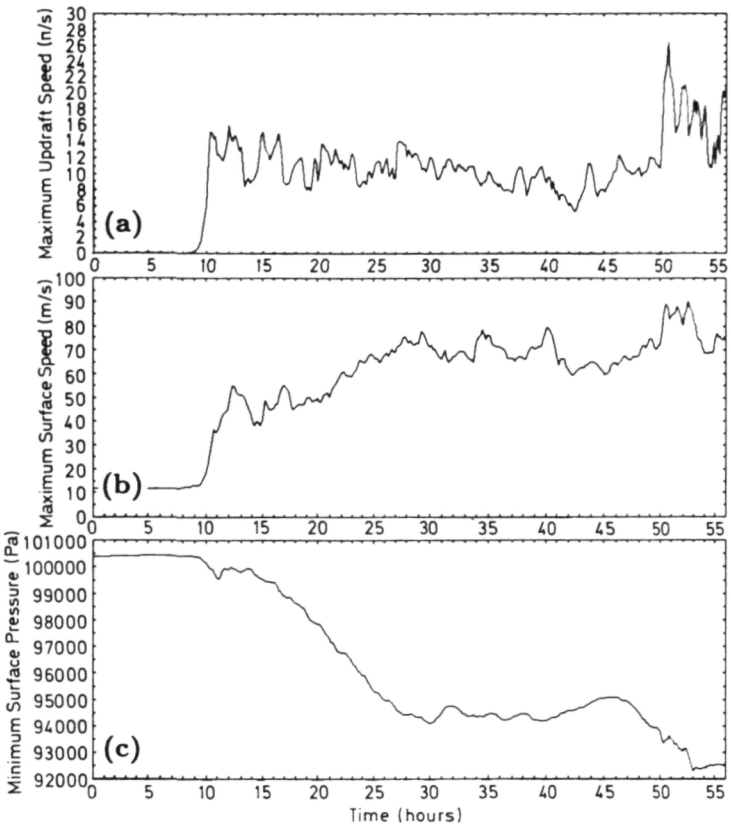

Fig. 3. Graphs of (a) peak updraft, (b) peak wind speed and (c) minimum surface pressure over the storm integration volume of the finest grid activated as a function of time from 0 to 56 hr

(not shown for this particular time) showed these waves to be spiral bands of vertical motion, not associated with deep convection outside the inner convergence ring. The bands sloped upward and outward, indicative of upward and outward propagation (Durran, 1981).

It must be recognized that the precise number and sizes of the mesoscale convective systems formed were probably greatly affected by the initiating mechanism from which they were created. In this simulation it was the tangential variation resulting from numerical truncation that caused the four cell convective system structure. The model grid resolution of the simulation was also an important factor, but not thought to be the dominant one since the meso-β scale systems were over 15 grid lengths across and so not near the smallest scale possible. It is most reasonable to expect that in a more realistic scenario, the convection would have developed a strong tangential variation as it did here, although the size and number of systems would have been dependent on variations in initial conditions. It is also likely that given no radial variation initially, there is an optimal size and number of convective systems that would form. Nevertheless, these results suggested that even minor departures from axisymmetry would take precedence over any tendency for an optimal size in determining the scale of the initial convective systems forming in the vortex. It is perhaps unsettling to this study that there is no reason to believe that this would not have a significant impact on the evolution of the cyclone.

After the 11 hour integration time, another period of meso-β scale growth ensued. Here, the low pressure again became most intense beneath three primary convective systems, reaching -7.5 hPa between the most intense system, compared to -6.5 hPa beneath the parent vortex central pressure. Note, however, that the parent vortex had closed the gap between its intensity and that of the meso-β scale convective systems of the first growth period. This was a result of the growth of the slow manifold[2] in response to long-term inertial adjustment to the convective heating.

[2] The slow manifold here reverses to the balanced part of the system, which evolves with inertial frequencies as opposed to fast manifold modes which evolve with frequencies related to the Brunt Vaisala frequency.

The growth and decay cycle of mesoscale convective system (MCS) activity surrounding the cyclone core continued for several hours. During this time, the amplitude of the slow manifold increased in strength and eventually reached the point where it could maintain inflow over and above the modulated inflow and outflow of the surrrounding convective systems. This occurred between 16 and 18 hours of simulation. This pulsating development process was also shown by Tripoli and Cotton (1989) to be an important component of the organization of orogenic mesoscale convective systems and could be viewed as the realization of the geostrophic adjustment process described theoretically by Schubert and Hack (1982).

During this period of growth which began at 9 hours of simulation, a rapid acceleration of the tangential wind occurred. Figure 3 shows the evolution of peak vertical motion, peak horizontal wind gust, and lowest surface pressure. The peak wind gust was somewhat less related to the tangential wind during the early stages because of the strong divergent wind in the early stages of storm development. Sustained tangential winds were found to be near $15\,\mathrm{ms}^{-1}$ at 11 hours and $20\,\mathrm{ms}^{-1}$ at 13 hours and $22\,\mathrm{ms}^{-1}$ at 14 hours. Hence, the tropical storm phase was reached around 14 hours of simulation, approximately 5 hours after the first free convection. The surface low at the center of the storm vortex was -8.5 hPa perturbation pressure of 1002 mb. Since the 14 hour time was actually in a peak MCS phase, the surface pressure was actually 0.5 hPa higher in the center than it was at 13 hours. Nevertheless, the 14 hour time marked the first point where the surface pressure remained equal to or lower than the MCS pressure lows surrounding the vortex center during the MCS phase.

Figure 4 shows the microphysical, TKE, θ'_V, vertical motion (viewed from below to see the low-level vertical motion), θ_e and potential vorticity at 14 hours. The Ertel potential vorticity (Z) used is defined as:

$$Z = \rho\eta\cdot\nabla\theta, \tag{1}$$

where ρ is air density, η is the absolute vorticity vector and θ is the potential temperature. The view is from the 2nd grid nest in order to attain a storm scale perspective.

Note that at that time, spiral bands were already apparent in the cloud water field. Precipitation was evident beneath the developing storm. At that time, a cirrus anvil was in the initial stages of formation. There was a false eye within the condensate field resulting more from the absence of convective towers than from forced subsidence. This can be seen from the lack of a warm core in the $\theta_v l$.

The microphysics of the convective regions were dominated by pristine crystals at 12–15 km, snow at 5–12 km, graupel at 5–7 km AGL, and rain below the melting level. Primary precipitation growth occurred by the Bergeron-Findeisen process followed by riming within the supercooled liquid water to produce graupel. The graupel then melted and formed rain. The snow, composed primarily of aggregates and rimed crystals reached average water contents in excess of 0.6×10^{-6} $\mathrm{g\,m}^{-3}$ at 10 km AGL (above ground level). Graupel reached contents near $2 \times 10^{-6}\,\mathrm{g\,m}^{-3}$ and melted entirely before reaching 3 km AGL. The massive melting between 3 and 5 km AGL was found to have a major impact on the entire storm evolution and will be discussed in more detail below. Rain reached contents near $5 \times 10^{-6}\,\mathrm{g\,m}^{-3}$ from collision coalescence growth. This produced peak rainfall rates of over 9 cm per hour at the surface.

Figure 4 also shows that the TKE field formed vertically erect plumes of vertical mixing in the vicinity of the deep convective towers. The TKE reached peak magnitudes of $10–20\,\mathrm{m}^2\mathrm{s}^{-2}$, showing that considerable vertical mixing assisted the explicit vertical transport. Nevertheless, peak vertical velocities within explicit convective motions were reaching $6–12\,\mathrm{ms}^{-1}$ at that time. This was large considering the meso-β scale of the fine mesh.

Vertically propagating internal waves within the stratosphere were evident in the θ'_v field. Animated visualization of the development showed that they emanated from overshooting updrafts and were spawned in a spiral with an angular frequency at the center equal to the angular frequency of the cyclone. This frequency was related to the combined lifecycle of the elevated cloudtop and the time it took to pass a point while circling the storm core. The outward propagation phase speed was $35\,\mathrm{ms}^{-1}$, calculated by clocking the wave front through a 40 minute animated propagation sequence. The vertical wavelength of

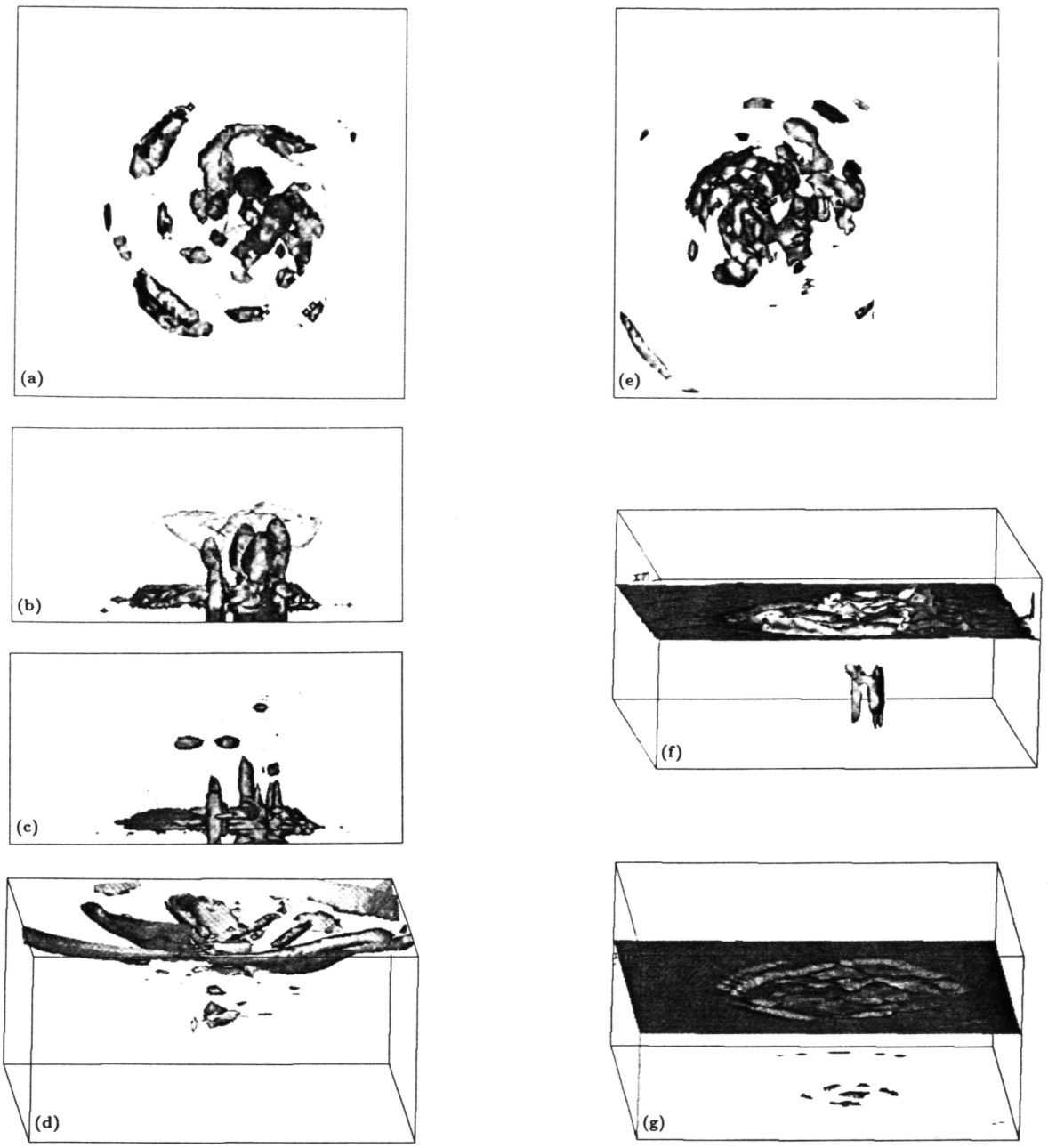

Fig. 4. The simulated three-dimensional storm viewed over the entire second grid (region is 1200 km across) and at 14 hr showing: (a) and overhead view of the 0.01×10^{-6} g m^{-3} condensate surfaces with north pointing up (the rain and graupel surfaces are darkely shaded, the cloud water and snow surfaces are lightly shaded, and the pristine crystal surface is partially transparent), (b) the view of the condensate surfaces described in a) from the south, (c) the view of the 2 m^2 s^2 TKE surface from the south, (d) an oblique view of the 5.1 K θ_v' surface from the west, (e) oblique view of the 0.3 ms^{-1} (lightly shaded) and -0.3 ms^{-1} (darkly shaded) vertical motion surface from below with north facing up (note east is to the left here) (f) oblique view of the 0.5 PVU surface from the south, and (g) an oblique view of the 355 k θ_e surface from the south

the internal waves was about 9 km while the horizontal wave length was approximately 150 km. Defining the linear gravity phase speed of:

$$c_p = \frac{N}{l} \tag{2}$$

where N is the Brunt Vaisala frequency of the stratosphere calculated to be 2.4×10^{-2} and l is the vertical wavenumber given by:

$$l = \frac{2\pi}{L_z} \tag{3}$$

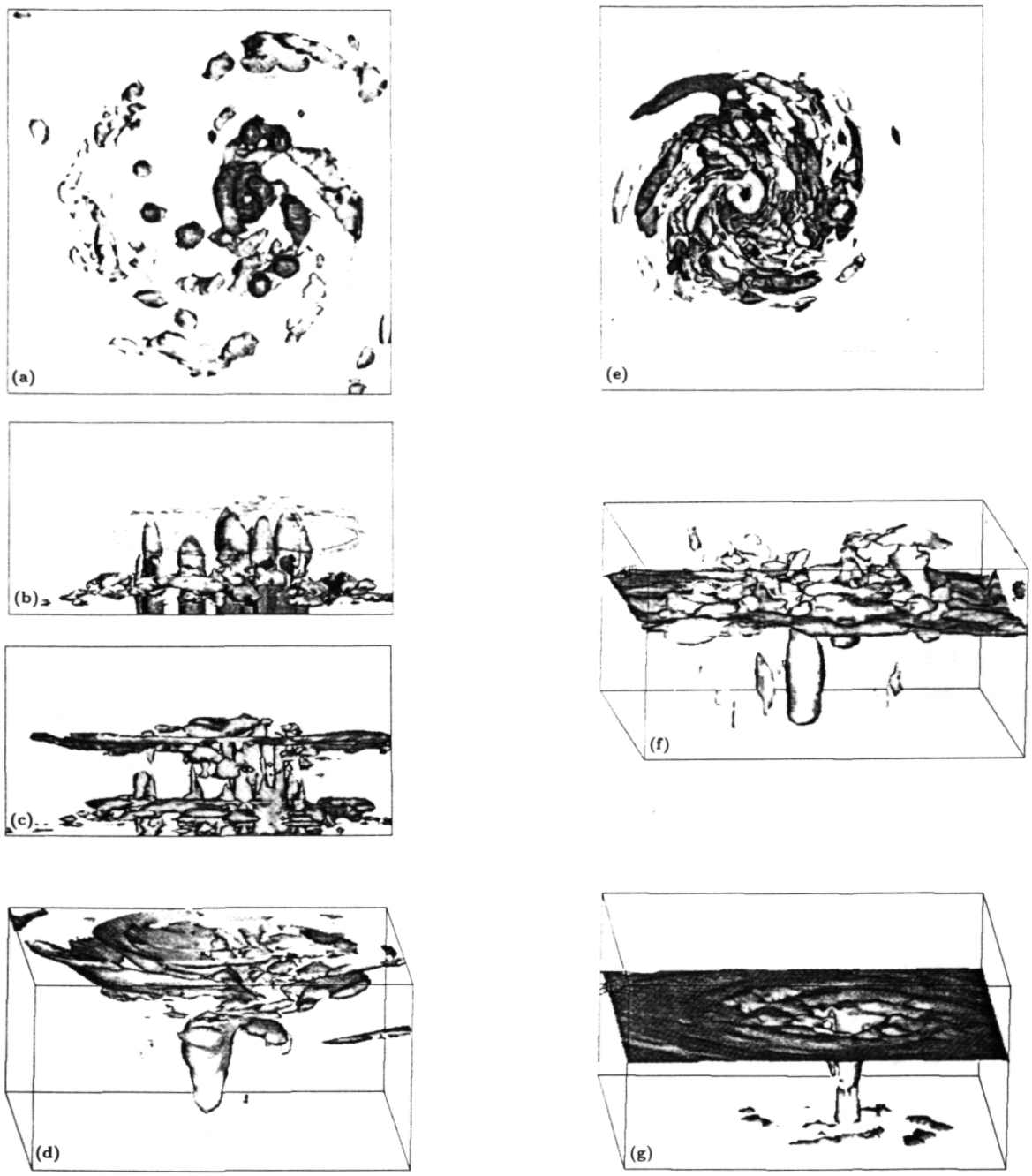

Fig. 5. Same as Fig. 4, except for 26 hr of integration

and L_z is the vertical wavelength, then $c_p = 34.6$ ms^{-1}, which is precisely the simulated phase speed.

The vertical motion field, viewed from below, showed the geometry of both the vertical motion associated with the MCSs and that associated with the gravity waves at all vertical levels. Some spiral bands were visible aloft in this figure, associated with the propagating stratospheric waves. Deep vertical plumes of vertical motion in excess of 1 ms^{-1}, not depicted in this figure existed in association with the MCS activity.

The potential vorticity field showed a small mid-tropospheric maximum had begun to develop in association with diabatic sources within the convection at 14 hours. Finally, the θ_e field showed growth of the 355 K surface, near the ocean surface, but not yet a consistent vertical plume of

the 355 K surface on the scale of the 20 km averaged grid. The peak θ_e at the surface roughly coincided with the strongest surface winds and hence the strongest moisture fluxes resulting from the bulk surface flux formulation.

4.1.2 Development of Inertially Stable Vortex Center – Transition to Hurricane

Figure 5 depicts the storm structure at 26 hours as the first rapidly deepening phase (which began just after 14 hours of simulation) was concluding. The local time was 0200 LST. By that time the surface pressure had decreased to 950 mb, for a net drop of 52 hPa in the 12 hr period since the 14 hour time discussed earlier. Peak horizontal winds had increased to average speeds approaching 70 ms^{-1}, clearly reaching the hurricane intensity range.

The condensate field showed that the convective activity had greatly spread outward from the eye wall in a banded structure. Low elevation liquid only stratiform cloudiness was associated with much of the banded cloudiness, showing that the clouds were forced by the bands rather than the clouds forcing the bands.

The vertical motion field exhibited a good eye wall within the troposphere with a series of spiral vertical motion bands radiating outward from the center at all levels. The spiral bands of vertical motion were most apparent within the stratosphere and had amplitudes of 0.25 to 0.5 ms^{-1}. Over approximately 30% of the region, however, the stratospheric bands extended downward into the troposphere, where a second vertical motion maximum was found approximately one vertical wavelength below the stratospheric maximum. The spiral bands were associated with a temperature perturbation of 5–10 K amplitude in the stratosphere and less than 2 K in the troposphere.

Over much of the region, the bands extending into the lower troposphere were coincident with the condensate discussed above. Animated visualization of the condensate field and vertical velocity field demonstrated that the condensate field was not moving outward with the waves. Instead, the waves tended to organize and amplify existing cloudiness as they propagated through the cloudiness. The clouds within the bands, tended to be short lived when compared to the propagating bands. This made individual groupings of clouds impossible to track for more than a couple of hours. Over the period they did exist, however, and they seemed to spiral slightly inward, with the low level flow. In this sense, they behaved similarly to the mesoscale convective systems at earlier times.

The outlying regions of cloudiness were found to have a significant impact on the surface layer θ_e field, creating relatively cold pockets. It was found that these cold pockets grew strongest from the melting of graupel at the melting level. In fact, in at least one case, a θ_e minimum was formed at the surface from melting above, which was less than the environmental minimum. These $theta_e$ minima proved to be long-lived and subsequently worked their way into the eye wall. Moreover, the stronger maximum seemed to initiate new convection when intersecting with outward propagating waves. As these density currents became caught up in the vortex, they tended to elongate in the horizontally sheared flow and become band-like themselves.

The surface beneath regions of convection within the eye wall was found to be associated with a relative θ_e minimum itself for the reasons discussed above for the outer bands. This produced a mild density current and a local pressure increase at the surface. This undoubtedly enhanced the convergence into the eye wall. Moreover these density currents together with the density currents merging from the outer bands enhanced the asymmetry of the eye wall, sometimes resulting in deep convection over only half of the eye wall. In fact, it was rare to have a completely closed pattern of heavy surface rain beneath the eye wall.

As these density currents moved around the eye wall, they seemingly split off the outer edges of the surface pressure gradient field, giving rise to the outwardly propagating bands which were distinctly visible in animated sequences of surface pressure. Since the outward propagating bands were also closely connected with the stratospheric forcing, the split in surface pressure gradient occurring at the formation of the band was inherently coupled to the stratospheric formation associated with the moving elevated cloudtop.

The predicted TKE structure showed that there was a continued existence of deep mixing processes in support of the crudely resolved explicit deep convection. The TKE within the eye wall had begun to form a closed cylinder in the lower portion of the eye wall, with individual plume

structures remaining aloft. This was indicative of the emphasis of storm scale spiraling eye wall convection in the lower levels. This resulted from increasing inertial stability which suppressed the development of divergent wind in response to the heating in favor of a rotational flow. Note also the extensive anvil level canopy of high TKE produced as a result of both shear production from the outflow layer and byoyant production from radiative cooling at the anvil top.

The θ_v' field showed clear evidence of a warm core by 26 hours, forced by subsidence from the stratosphere. The development of the warm core was associated with the cessation of the growth and breakdown process of MCS activity in the eye wall discussed earlier. Prior to the development of sufficient inertial stability of the vortex, the subsidence warming forced by individual convective plumes resulted in the breakdown of the convection in the eye wall due to the destruction of low level pressure gradients supporting the MCS inflow. As the storm scale vortex increased in inertial stability, inertial frequencies increased to the point where the tangential wind of the vortex could adjust to the warming so that divergent flow from the eye wall into the eye would not form and act to destroy eye wall convection. Hence by 18 hours of simulation, the individual plumes ceased their periodic breakdown in the eye wall, and the growth of a persistent warm core commenced.

This was shown clearly by Fig. 6 which depicts a series of trajectories at the early stage of storm development centered around 14 hours and the maturing stage centered around 26 hours. Note that in the early stage, the updraft trajectories spiraled into the center at the surface and then moved nearly straight up in deep convection plumes. At the later time (Fig. 6b), the trajectories spiraled upward in the lowest 5 km and then moved vertically upward. As the storm increased in intensity toward 50 hours (Fig. 6c) the spiral became deeper with little or no explicit vertical movement suggestive of vertical convection. Instead of spiral convection dominated, which was indicative of the dominance of the inertial frequency over the convective and gravity wave frequencies.

The continued growth of the inertially stable vortex was also evident by the formation of a strong potential vorticity maximum of up to 2 PVU at the storm core. A strongly varying potential vorticity field was induced in the lower stratosphere

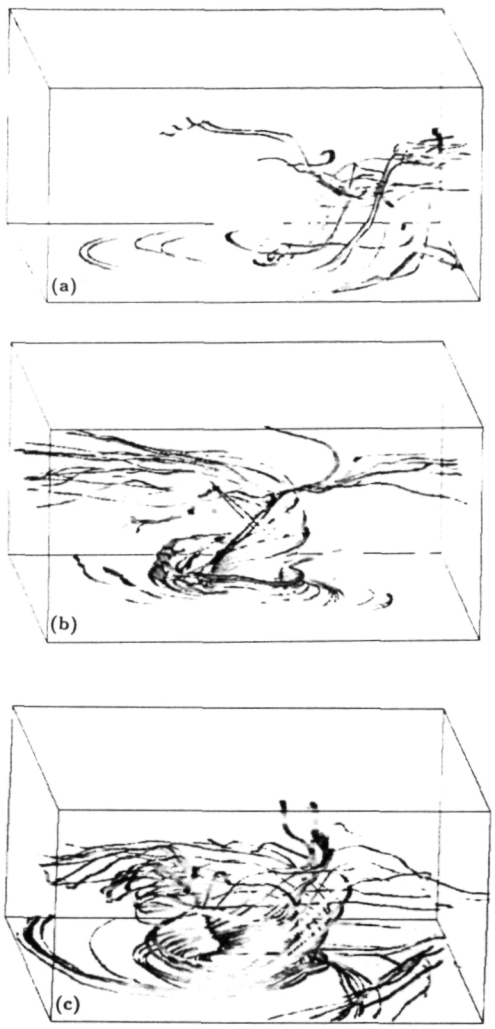

Fig. 6. Oblique view of air trajectories through storm core, viewed from the south centered at (a) 14 hr, (b) 38 hr and (c) 48 hr of integration time

by the penetration of convection from below. The potential vorticity structure at the tropopause was characterized by a core of high positive potential vorticity surrounded by regions of negative potential vorticity induced by convective momentum transport from below.

At this time, the 355 θ_e surface had formed a vertical conduit within the storm core and from the surface to the stratosphere. The increased surface values of θ_e were distinctly banded, and associated with the banded structure of the cloudiness and vertical motion field.

The increased inertial stabilization of the eye wall was clearly evident in the evolution of the surface pressure and wind vector fields displayed in Figs. 7 and 8. At 14 hours of simulation,

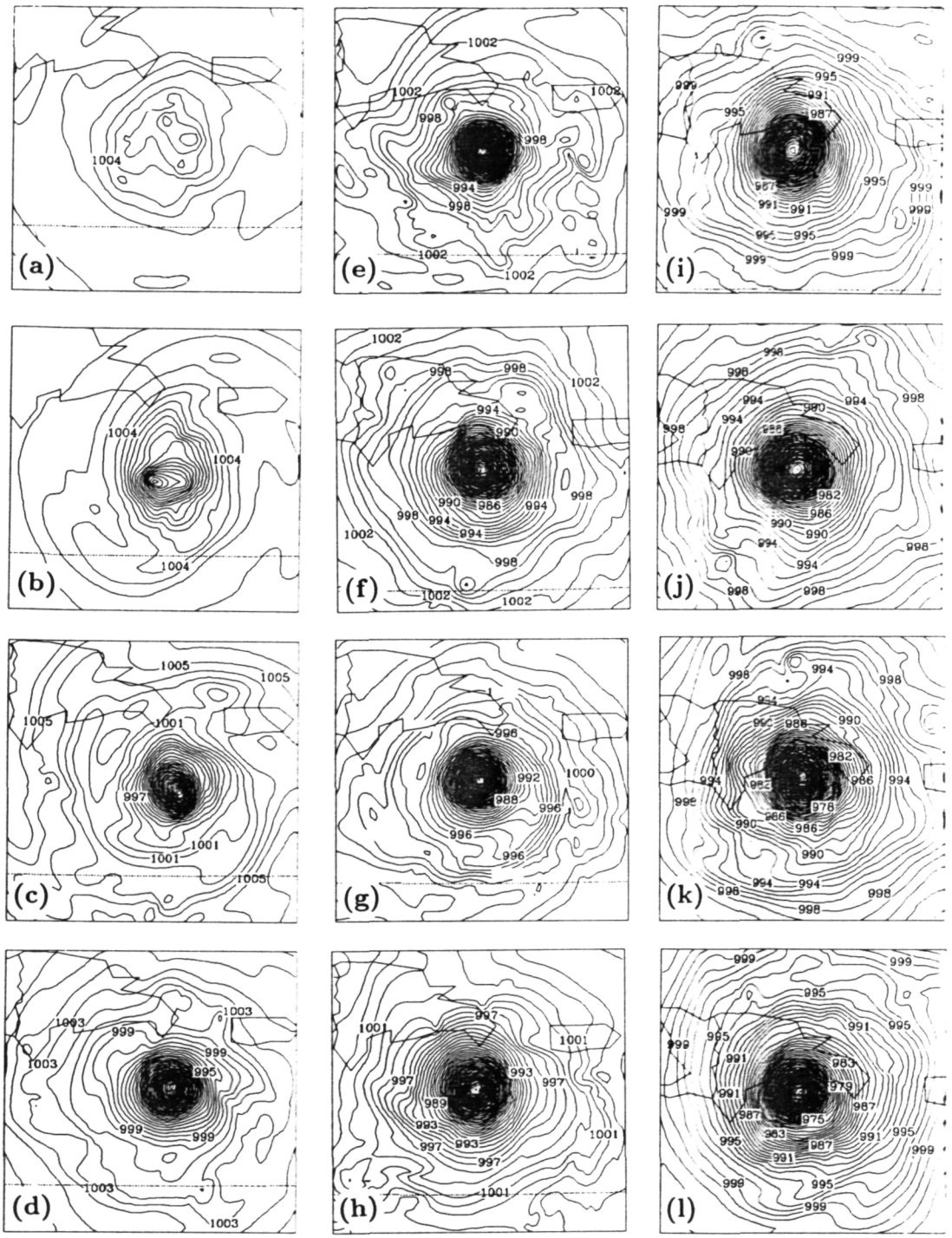

Fig. 7. Surface pressure field contoured at intervals of 1 hPa every four hours beginning at 14 hr of integration with (a) and ending at 54 hr with (k). The pressure at 56 hr (l) is shown

multiple pressure centers were found, while the velocity field was characterized by a series of minor vortices within the primary circulation. By 18 hours a dominant storm core low pressure had emerged and by 22 hours a nearly axi-symmetric low pressure core was evident. At the same time a convergent eye wall flow became more visibly consolidated into a circular convergence band and by 26 hours it became reduced in size to approximately 150 km in diameter.

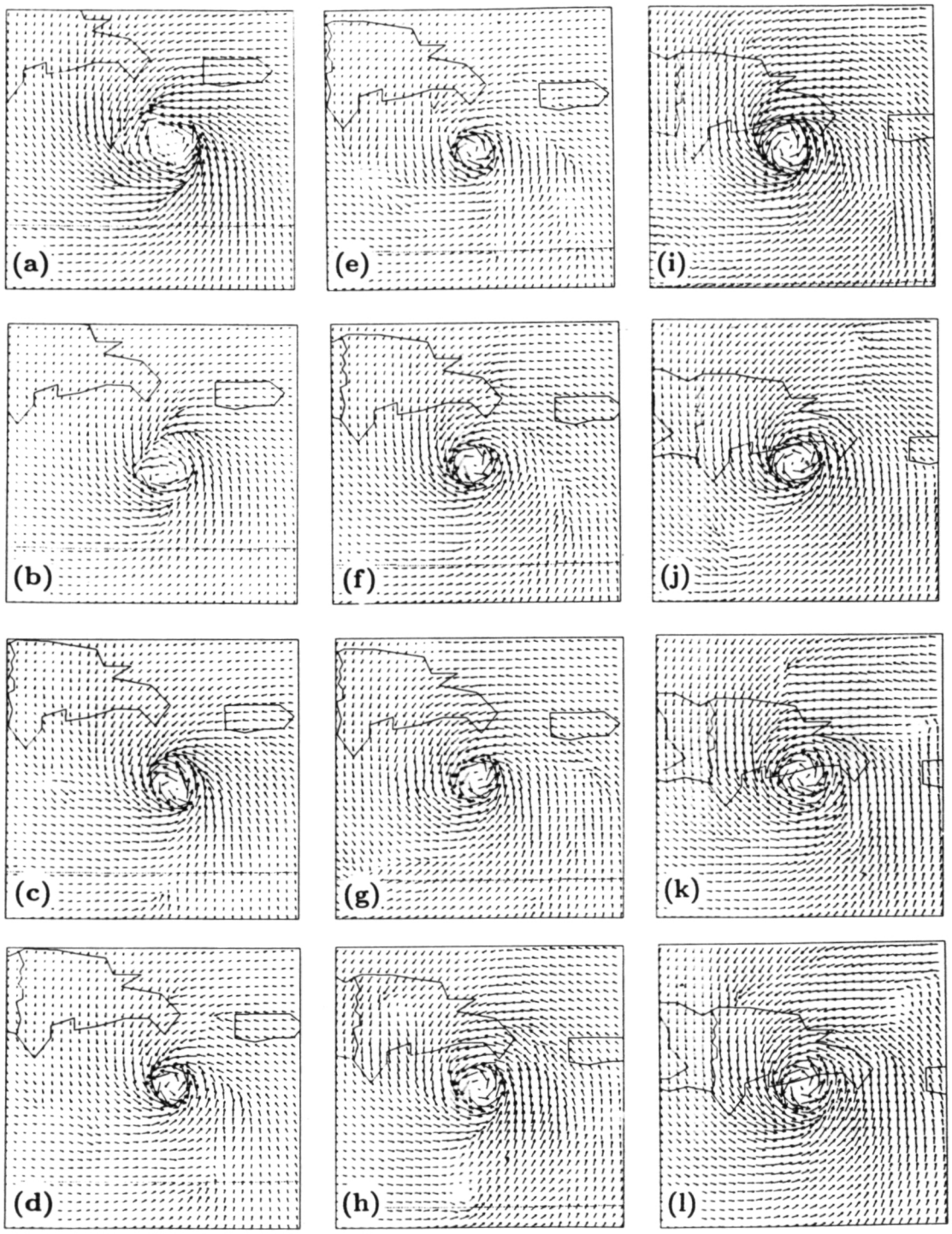

Fig. 8. Same as Fig. 7 except wind vectors are shown instead of pressure at surface level

Outside the storm core, the tangential variation of the pressure structure remained. It was associated with the movement of gravity waves and with regions of convection where density currents were formed. With each major wave node emanating from the core, the rate of pressure fall was momentarily reduced, suggesting that the gravity wave activity was weakening the warm core, and hence reducing the rate at which the cyclone could deepen. Thus, the convection heating was balanced inertially within the core, but continued to form wave activity outside the core where inertial

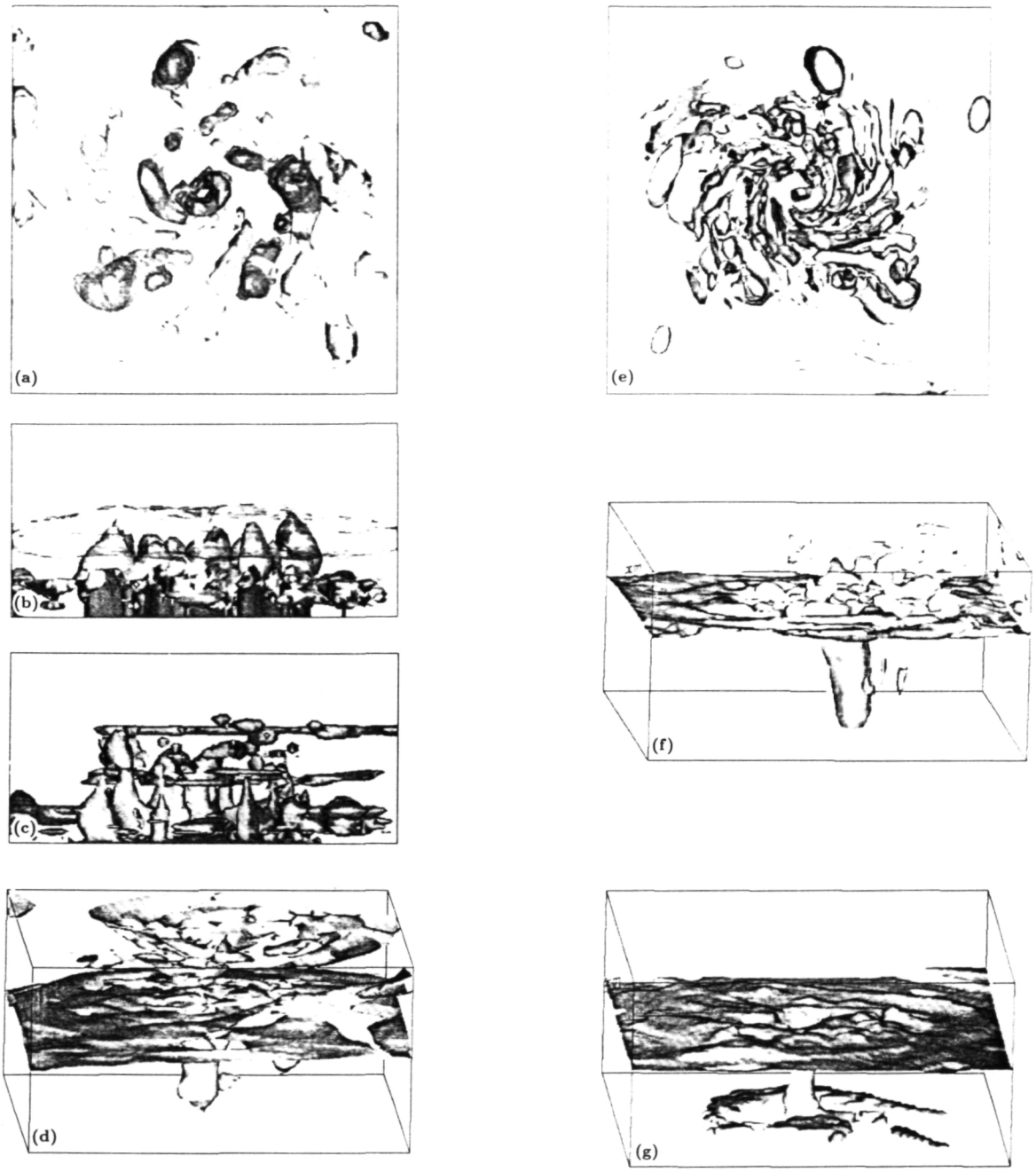

Fig. 9. Same as Fig. 4, except for 38 hr of integration

stability was less. This is again consistent with adjustment theory posed by Schubert and Hack (1982) (Schubert and Hack, 1982).

4.1.3 Inertially Stable Vortex Attains Quasi-Steady Balanced State at 38 Hours – Hurricane Stage

The three dimensional storm structure at 38 hours (1400 LST) is depicted in Fig. 9. At this time the minimum surface pressure had fallen to 942 mb, or another 8 mb since the 26 hour time. In fact, all of the pressure fall was before 30 hours when the sun rose above the horizon. Overall, the pressure evolution was much less dramatic than the previous 12 hours and, the times of maximum pressure fall, roughly coincided with darkness.

At the 38 hour analysis time the cirrus anvil, composed of pristine crystals with contents of about 0.5 gm^{-3}, had expanded beyond the second

grid. The vertical motion continued to maintain a circular eye wall, however, the banded structure outside the eye wall had become somewhat fragmented. This seemed to be a result of the convection in the outer bands creating gravity waves of its own, interfering with the more coherent waves emanating from the eye wall.

Systematic warming of the anvil top in the afternoon hours was evident in the θ'_v field. It was also evidenced by a weaker TKE maximum at the upper surface of the anvil and the more shallow penetration of the TKE field due to increased stability aloft . The lesser vertical penetration of convection was also evident in the potential vorticity field, which seemed to produce less excitation of stratosphere turbulence than earlier.

Significant outlying convective activity remained, as evidenced by the condensate fields. Comparison of this numerical simulation with a host of previous similar simulations, suggested that much of the outlying convection was supported by the anvil itself. It has been found that the amount of outlying convection was quite sensitive to the amount of water carried out from the center in the anvil. The precise reason why has not been investigated at this time, however, it is speculated that the precipitation may have moistened and destabilized the low to mid levels of the troposphere, releasing the thermal "cap" imposed on outlying convection created by forced subsidence. This enabled the spiral inertia-gravity waves emanating from the center to initiate convection more readily. In fact, in one simulation similar to the one described in this paper except that there was no radiation and there was a slightly different setting on pristine crystal size, the anvil size was dramatically reduced and almost all of the outlying convection was eliminated. This resulted in long term dramatically coherent stratospheric bands over the entire 50 hours of simulation. It is likely that result was at least partially a result of the radiation difference and remains a topic for future research.

Associated with inertial stabilization at 38 hours was the substantial weakening of peak convective updrafts to 6–10 ms^{-1}. This was likely due at least partially to the inertial stabilization of the vortex, although outlying convective updrafts would not be expected to decrease for that reason. Another probable reason was the diurnal stabilization of the upper troposphere in response to daytime warming, which has already been suspected to have weakened the TKE and potential vorticity fields. The fact that the peak updraft and pressure falls increased again after the next sunset, seem to support this hypothesis.

4.1.4 Explosive Deepening Resumes After Sunset

Shortly after sunset, which was at 42 hours of simulation (1800 LST), strong deepening of the storm resumed and the central pressure fell from 952 mb at 45 hours to 922 hPa at 53 hours for a total fall of 85 hPa from the initial vortex specification.

The simulation at 50 hours was depicted in Fig. 10. At this time a massive "tree trunk" of 355 K θ_e rose upward within the eye wall. Convection was more limited to the eye wall and inner bands.

TKE field again began to penetrate into the lower stratosphere as they did at the 28 hour analysis time, although there was an absence of strong TKE along the upper surface of the cirrus anvil, suggesting increased stability over the previous night. It is expected that this may have been a result of the accumulated warming of the region during the day, the effects of which remained.

The warm core remained strong, but was connected to strong warming spread along the base of the anvil as a result of longwave absorption of upwelling radiation from low levels. The potential vorticity maximum continued to be maintained also at 50 hours, although greater perturbation of the stratospheric potential vorticity field again commenced in response to the deepened convection.

The eye wall diameter remained at 120 km, with an eye diameter of about 30 km. Peak tangential winds increased in response to the new growth to 80 ms^{-1} by 50 hours of integration.

4.1.5 Extended Simulation with Meso-γ Scale Resolution

Because the simulated eye wall had stabilized at a mean diameter less than 200 km, it was deemed deemed feasible to add a fourth grid with 3.3 km horizontal resolution to better resolve eye wall processes. The grid was added at 50 hours, and the integration was extended to 56 hours. The vertical motion field shown in Fig. 10 shows that the peak vertical velocities responded immediately to the increase resolution but then tended back

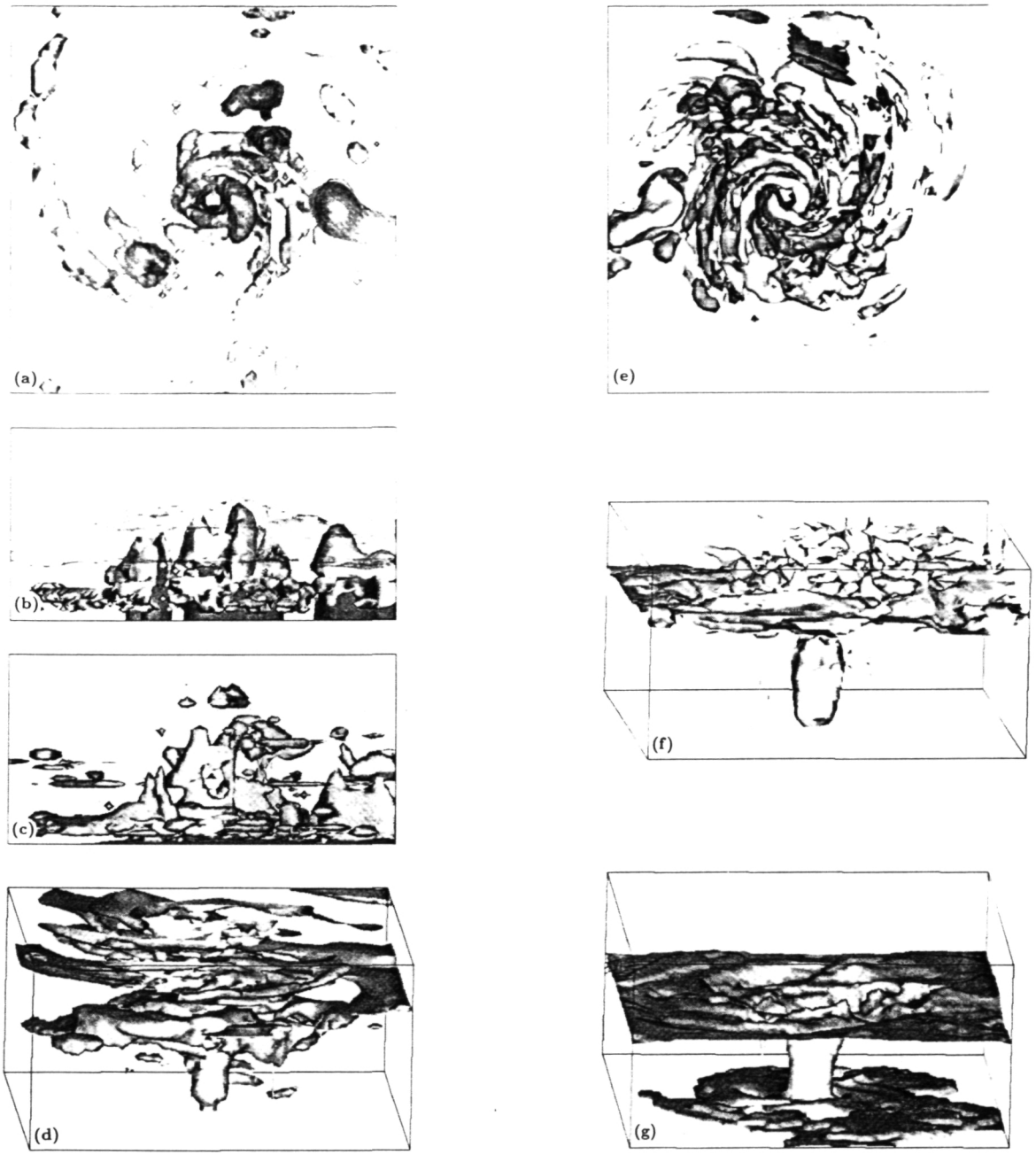

Fig. 10. Same as Fig. 4, except for 50 hr of integration

downward. This probably resulted from a short term imbalance of the core resulting from less overall horizontal dissipation in the balance as a result of less numerically related smoothing. The peak surface wind speed also momentarily gusted to over 85 ms^{-1}, but then weakened to previous levels. Surprisingly, the minimum surface pressure continued on the same trend downward it began at 45 hours and then stabilized at between 922 and 925 mb.

Figure 11 shows the predicted three-dimensional fields after 56 hours. Overall, they did not look substantially different than they did with the 10 km resolution when interpolated to the second grid. Some finer structure of the inner bands and within the TKE field was apparent.

To better depict the structure of the inner core, selected fields of the simulated eye wall within the 4th grid were displayed with full resolution. Figure 12 depicts the cloud, TKE, w, potential

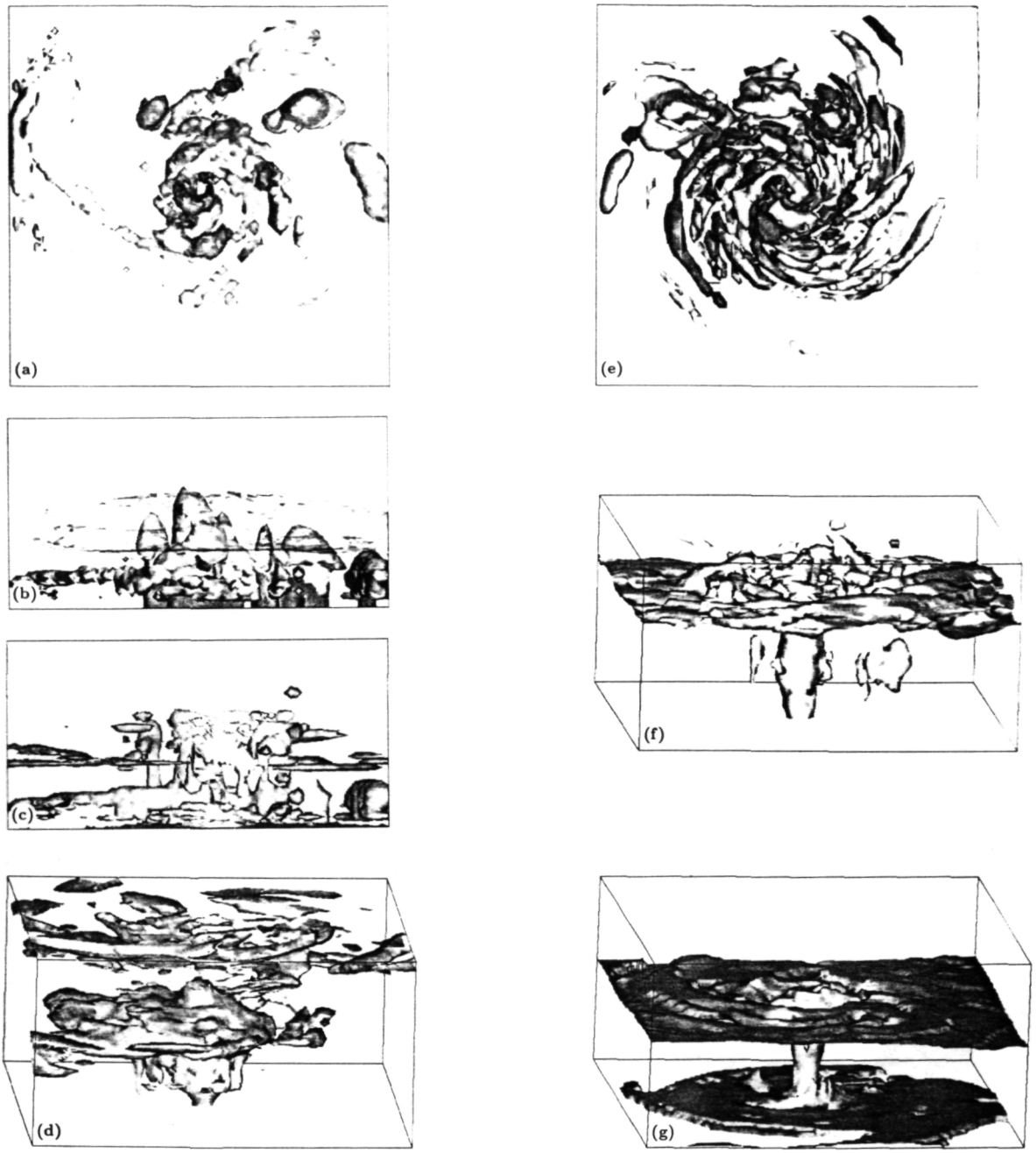

Fig. 11. Same as Fig. 4, except for 6 hr of integration

vorticity and θ_e fields on the high resolution grid. Here the fine structure of the spiraling vertical motion field was readily apparent. The eye wall was constructed of several asymmetric bands of vertical motion. The cloud field was shown to almost totally lack cumulus structure and instead gave the appearance of a spiral shaped stratiform-like cloud. The spiral structure appeared strongly in the upper surface of the cirrus canopy, as was commonly observed by satellite.

Note also the more continuous spiraling appearance of the θ_e surface maximum. This was a

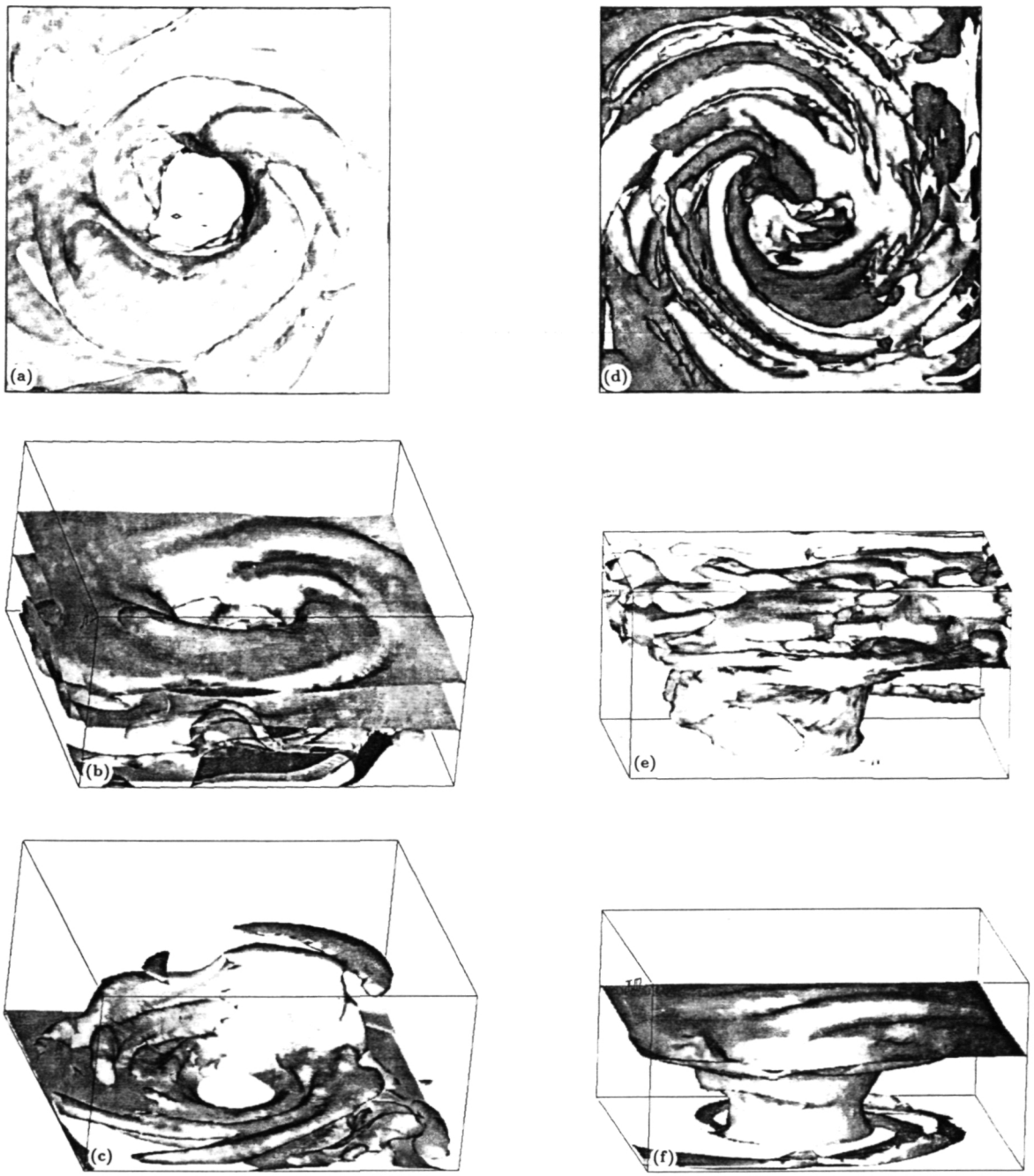

Fig. 12. The simulated three-dimensional storm structure viewed over the fourth grid (region is 198 km across) and at 56 hr showing: (a) an overhead view of the 0.01×10^{-6} gm^{-3} condensate surfaces with north pointing up (the rain and graupel surfaces are darkly shaded, the cloud water and snow surfaces are lightly shaded, and the pristine crystal surface is partially transparent), (b) oblique view of the condensate surfaces described in a) from the south showing wave like texture to upper cloud surface, (c) oblique view of the 2 m^2 s^{-2} TKE surface from the northwest, (d) overhead view of the 0.5 ms^{-1} (lightly shaded) and -0.5 ms^{-1} (darkly shaded) vertical motion surface from below with north facing up (note east is to the left here) (e) oblique view of the 0.5 PVU surface from the south, and (f) an oblique view of the 359 K θ_e surface from the south

result of the higher resolution depiction of the eye wall density current next to the inwardly spiraling flow.

The TKE field was confined to a considerably smaller region of the plume. This might be expected because a greater portion of the updraft was resolved explicitly and so the subgrid scale became relatively laminar, except for some local instabilities within bands.

The potential vorticity field shows considerably more structure presented on this grid. Contour cross sections through the field at low levels (not shown here) show that the potential vorticity was layered into the main center forming a spiral itself.

The inertial stability can be defined as:

$$I^2 = (\zeta + f)\left(\frac{2V_T}{R} + f\right) \tag{4}$$

where ζ is the relative vorticity, f is the coriolis parameter, R is the distance to the vortex center, and V_T is the tangential wind of the vortex.

Examination of the low level inertial field (not shown) showed that the minimum of potential vorticity within the spiral was associated with a minimum of inertial stability just outside the eye wall. In fact, the inertial stability formed a spiral relative to the near spiral bands and even became negative within the split where the band formation was occurring at 56 hours. The inertial stability was maximum within the band, but still very strong within the eye itself. Hence where outward propagating bands were forming, a strong gradient of inertial stability was apparent across the region of convection.

This explains why the outer edge of the eye wall convection split forming the band. On the inside of the eye wall, the subsidence warming by the convection resulted in gradient wind adjustment and thus not weakening the eye wall. On the outside edge of the eye wall, angular momentum transported in and brought around the storm led to weakened inertial stability. This caused convergence into the eye wall to be weakened, throwing the eye wall convection into a breakdown mode. This in turn caused the surface pressure in that region of the eye wall to rise while the pressure outside the eye wall had fallen. This weakened the updraft and gave rise to an outwardly propagating inertia-gravity wave.

It is clear that a coherent relationship between the stratospheric wave and the tropospheric wave was maintained throughout the inertia-gravity wave spawning process. The extent to which resonance helped determine the frequency of convection, the optimal storm rotation rate and a host of other possibilities, will remain an intriguing topic for future research.

5. Discussion of Scale Interactions

The above results described several scale interaction processes occurring during the simulated tropical cyclogenesis. These processes have not been widely discussed before, primarily because previous numerical simulations have used simplified hydrostatic, axisymmetric and cumulus parameterized frameworks which poorly represent such processes. Although all of the processes described have not been proven to exist in nature, their existence in this simulation raises interesting questions and gives valuable insight into how three-dimensional tropical cyclones might explicitly interact with convection. The following is a brief summary of the major types of scale interaction noted and their impact on the lifecycle of the simulated storm.

5.1 Simulated Mesoscale Convective Systems

Convective plumes were simulated to preferentially separate into meso-β scale entities in the early stages of storm development, before strong inertial balance evolved. These entities, showing both a strong gradient balanced and divergent part, appeared in the pressure field as Rossby wave-like disturbances as described by Challa and Pfeffer (1980). The divergent part of the circulation, driven by the latent heating aloft, acted to contract the vorticity field to produce the wave.

The simulated embedded MCSs underwent growth and breakdown cycles similar to those described by Tripoli and Cotton (1989a). These cycles were characterized by a period of growth where low level convergence was drawn into the system at sufficient magnitudes to fuel the system growth, while divergence or detrainment was produced at upper levels. When low level convergence was interrupted, the level of no horizontal divergence in the updraft moved upward leading to the entrainment of lower θ_e air and further interruption of the fuel supply. The process has been described theoretically by Raymond (1984) for the wave-CISK problem, where the growth

process was described as an *advective* mode of unstable growth. Raymond found breakdown occurred when the surface divergence/convergence was upset giving rise to *propagating* modes of inertia-gravity waves.

The breakdown process, once initiated continues because the propagating gravity waves become the growth modes. This process was fundamentally similar to that which led to the breakdown of single cumulus plumes on the 20 minute time scale except for the processes which initiate the breakdown phase. Hence, the field of growing and collapsing MCSs continuously emitted inertia gravity waves which move about the storm, and eventually dissipate or move into the stratosphere where the model's absorbing layer removed them.

5.2 *Gravity Wave Formation Induced by Locally Weakened Inertial Stability*

Asymmetric surface deformation fields produced by convergence of flow into the developing ring of maximum winds produced a tangentially varying angular momentum field, which when advected around the vortex produced regions of weak inertial stability or inertial instability. These instabilities locally weakened the convergent flow into the eye wall by allowing the flow to respond to subsidence warming outside the eye wall. This could initiate the local breakdown of convection in the eye wall and could give rise to the formation of a propagating inertia gravity waves in the form of outward propagating bands of vertical motion, temperature and pressure. These waves, which propagated radially outward at approximately 35 ms^{-1}, appeared to be always coupled with a stratospheric wave of similar phase speed.

The relationship between the upper and lower inertia-gravity waves was by no means just a coincidence, since the breakdown of the eyewall convection at lower levels was also tied to a collapsing cloud top. It is interesting that the growth and breakdown process seemed to take on a continuous form in the eye wall. The convection moved around the eye wall with different sections of the eye wall containing convections at different stages of the process, and thus resulting in the continuous emission of spiral gravity wave activity. At later times, the simulation seemed to prefer a tangential wave number one, with one dominate side of upward motion. There were periods however, where more continuous eye wall convective activity was apparent.

5.3 *Simulated Eye Wall During the Hurricane Stage*

As the tangential wind of the vortex increased in the hurricane stage, and the core reaches greater inertial stability, the tendency for the low level wave like MCSs within the eye wall jet were suppressed. Whereas individual air trajectories move vertically in the early phase of the storm within the individual convective systems, the grid scale air trajectories in the eye wall move in a cyclonic upward spiral during the hurricane stage, suggesting a dominance of the inertial frequency over the convective frequency within the eye wall. Since the predicted TKE magnitudes remained high in the eye wall, this motion could be interpreted to mean that individual convective plumes may have remained, although perhaps not always traversing the entire depth of the troposphere. On the other hand, MCSs, or substorm scale groupings of convective plumes were eliminated during the hurricane stage in favor of a single eye wall system.

The lowered production of gravity wave energy in a hurricane has been discussed before as a natural consequence of the geostrophic adjustment process (Schubert and Hack, 1982). The lesser production of gravity wave energy has been linked to greater efficiency of growth and hence an acceleration of the growth rate of tropical cyclones as they approach maturity. In this simulation, this process was seen explicitly as a changing form of the mesoscale convective entity from cyclic convective entities to a quasi-steady eye wall circulation.

5.4 *Simulated Eye During the Hurricane Stage*

Another consequence of the inertial stabilization of the eye wall convection was the formation of a true "eye". In the early depression and tropical storm stages, an intermittent clearing in the storm center resulted by the emphasis of convection along the annulus of high winds where the surface fluxes of θ_e were highest. This appearance would be less likely if a mean wind was considered, which would somewhat negate that effect. The early eye-like structure exhibited only intermittent subsidence within and was too large in diameter.

As the storm reached hurricane stage, however, and upward motion became more axisymmetric along the eye wall, mean subsidence developed to the surface producing an eye 20–30 km in diameter. This resulted because the inertial frequency of the storm increased to exceed that of the formation of divergent wind in response to the convective heating. As a result, the central warming no longer forced nor fueled convective breakdown in the eye wall and became long-lived.

5.5 *Microphysics Feedbacks*

It was found that the convective regions produced surface θ_e minima, primarily as a result of extensive melting in the 3–5 km AGL layer. These surface θ_e minima outside the eye wall showed long-term persistence and, once formed, tended to be associated with more long term convection.

The θ_e minima, became entrained into the inward spiral of inflow and seemed to be focal points for the initiation of new convection by spirally banded banded gravity waves passing overhead.

The ice anvil also appeared to have a substantial impact on storm structure and growth. It is expected that the radiative feedback was of importance because of its affect on increasing upper tropospheric stability during the day and decreasing it during the night. This may have helped to duct tropospheric gravity waves beneath the unstable anvil top, enabling spiral cloud band formation at greater distances from the core.

The other major impact hypothesized was the anvil's impact on low level stability as a result of the virga falling into the θ_e minimum. This would have tended to weaken tropospheric stability to the point that outwardly propagating inertia-gravity waves could more easily initiate clouds or unstable convection.

The present results combined with earlier studies such as Kanak (1990), Kanak and Tripoli (1990) and other unreported preliminary experiments suggest, that in the absence of a high ice water content in the cirrus anvil, the diurnal radiative effect of the anvil dominates and the existence of outlying convective activity took on a diurnal fluctuation in intensity. When essentially no anvil was present, outlying convection became virtually nonexistent. Hence radiative transfer combined with microphysical processes seemed to have a dramatic influence on storm intensity, structure, growth rate and nature of internal scale interaction.

6. Conclusions

This paper described nonhydrostatic and three-dimensional scale interaction simulations of a tropical cyclone. The numerical model successfully simulated tropical cyclogenesis using a 1.5 level turbulence closure to represent cumulus activity on a meso-β scale nested grid system. Although the simulation was not of a real storm situation, the storm genesis from a prescribed initial vortex appeared to be realistic. Important features of simulated scale interaction were demonstrated, some of which have been implied in the literature previously and some of which are apparently new. The major conclusions are as follows:

- The scale interaction within the simulated tropical cyclone occurred in two major phases summarized as follows:

  (1) The "tropical depression" stage made up the first phase of scale interaction, and was characterized by the proliferation of meso-β scale growing and decaying convectively forced disturbances which spawned deep propagating inertia-gravity waves. Waves were spawned in both the stratosphere and troposphere. The amplitude of the pressure ocscillation beneath the MCSs exceeded the central pressure of the storm.

  (2) The "tropical storm" stage marked the period where the storm's central pressure exceeded the amplitude of the fluctuation in the surrounding convection. This enabled a continuous warm core to grow and maintain mean subsidence for increasingly long periods. Convective entities within the eye wall became longer lived as a result of an increasing portion of the induced warming being captured within the inertially stable eye wall without loss of surface convergence.

  (3) The "hurricane" stage was characterized by inertially balanced eye-wall convection which maintained continuous eye subsidence which allowed the subsidence to work it way to the surface. At that time the cyclone pressure became nearly axisymmetric inernally although important asymmetries existed outside the eye wall due to inertia-gravity

wave activity. Convection in the eye wall became less cellular with updrafts taking on the scale and geometry of the eye wall itself. The eye wall at times seemed to behave as a single convective cell.

- Simulation with a meso-γ resolving grid of 3.3 km spacing within the eye wall had a surprisingly small impact on the storm evolution, although much smoother and more coherent looking features were simulated. The results suggested that a 10 km horizontal resolution was likely sufficient to approximate mature tropical cyclone dynamics.

These simulations crudely begin to unfold the explicit scale interaction processes which exist, but offer the reader little confidence that the answer has been found. Instead, they instill a sense of uneasiness that there are important essential processes within convective weather systems that have never been addressed theoretically or within predictive model frameworks. As research numerical models become capable of better representing these process by simulating a greater range of wave numbers using larger grids, a better understanding of the processes will be achieved, new questions will be posed and requirements for accurate prediction will be made.

Acknowledgements

The author would like to thank Kathy Kanak, who helped develop the tropical cyclone model and analysis system and ran the first high resolution three-dimensional experiments. Thanks also to Pete Pokrandt for useful comments and his help with model development. Thanks to Mr. Brain Paul for his considerable help in customizing visualization software for this paper. Jill Bushner and Bonnie Tripoli helped editing the manuscript and preparing the figures. The numerical computations were performed, on the NCAR (National Center for Atmospheric Research) CRAYs and on the University of Wisconsin-Madison Space Sciences and Engineering Center Stellar GS-2000 super graphics work station. NCAR is partially supported by the National Science Foundation. This work was supported under NSF Grants ATM-88-05460 and ATM-91-01434 and NASA grant NAG8-828 for the visualization.

Appendix A

Level 1.5 Closure

The level 1.5 closure scheme is based on the scheme presented by Redelsperger and Sommeria (1981) (hereafter referred to as RS) with some modification. For a complete derivation, the reader is referred to that paper. Here, the modifications to the RS scheme are presented and the final form of the implementation of the scheme is given.

The closure simply involves an alternate procedure for computing the mechanical (K_M) and thermal (K_H) eddy mixing coefficients used in the second order diffusion term. In the case of the hurricane, the scheme is only applied to the calculation of vertical eddy mixing coefficient. The TKE is defined as:

$$e = 0.5\, u_i'^2, \tag{A1}$$

where u_i is the three dimensional velocity tensor and the prime refers to the turbulent deviation from the resolved scale. Retaining only the largest order terms defined in RS, the vertical mechanical mixing coefficient is defined as:

$$K_m = C_m l_z e^{1/2} \frac{1 - \max(0, C_1 R_i)}{1 + \max(0, C_1 R_i)}, \tag{A2}$$

where l_z is the vertical grid spacing, $C_m = \frac{1}{15}$, and R_i is the Richardson number, given by:

$$R_i = \frac{N^2}{D_{ij}^2}, \tag{A3}$$

where N is the Brunt-Vaisala frequency and D_{ij} is the deformation tensor. The constant C_1 is defined to be:

$$C_1 = \frac{2}{3CC_\theta}, \tag{A4}$$

where $C = 4.0$ and $C_0 = 1.2$. The thermal eddy mixing coefficient is related to the mechanical mixing coefficient by the formula:

$$\frac{K_H}{K_M} = \frac{5}{2}\frac{C_M}{C_H}, \tag{A5}$$

where $C_H = 4$.

The tendency equation for TKE is given by:

$$\frac{de}{dt} = -C_M l_z e^{1/2}\left(D_{ij}^2 - \frac{K_H}{K_M} N^2\right)\frac{1 - \max(0, C_1 R_i)}{1 + \max(0, C_1 R_i)} - C_\epsilon l_z^{-1} e^{2/3} - e\frac{2}{3}\frac{\partial u_i}{\partial x_i}, \tag{A6}$$

where u_i is a velocity tensor, x_i is the cartesian distance, and C_ϵ is the dissipation coefficient which is set to $C_\epsilon = 0.7$ except near the surface where it is ramped up to $C_\epsilon = 2.7$.

The buoyancy production (term involving N^2 in A6) is the primary term driving the development of deep subgrid scale moist convective overturning. The vertical thermal buoyancy driven acceleration of velocity can be defined as:

$$\left(\frac{du_3}{dx_3}\right)_{\text{buoyancy}} = g\frac{\theta'_{vil}}{\theta_0}, \tag{A7}$$

where θ_0 is a reference state potential temperature and θ_{vil} is a mixed phase virtual potential temperature defined as:

$$\theta'_{vil} = \theta\frac{1 + 0.61 q_v}{1.0 + q_l + q_i} - \theta_0, \tag{A8}$$

where θ is the potential temperature, q_v, q_l, and q_i are the

specific humidities of vapor, liquid and ice in the system. This form of virtual potential temperature is useful because it takes into account both the effects of vapor and water loading in addition to that of temperature in creating buoyancy.

Following RS, the buoyancy production term is related to the vertical heat flux by:

$$\frac{g}{\theta_0}\overline{u_3'\theta_{vil}'} = -K_H\frac{g}{\theta_0}\left(\frac{\partial\theta_{vil}'}{\partial x_3} - \frac{L_m}{c_pT}\frac{dq_v}{dx_3}\right) = -K_HN^2, \tag{A9}$$

where c_p is the specific heat at constant pressure, T is the temperature and L_m is the latent heat of phase change defined as:

$$L_m = w_lL_{vl} + (1 + w_l)L_{vi}, \tag{A10}$$

where L_{vl} and L_{vi} are the latent heats of evaporation and sublimation respectively and w_l is a weighting function used to represent the change from liquid to ice in the temperature range of 253–273 K given by:

$$w_l = \min\left(1, \max\left(0, \frac{T - 273.16}{20.0}\right)\right). \tag{A11}$$

The latent heating term is to take into account the effects of heat released or absorbed by phase changes associated with either saturated ascent or melting and evaporation on unsaturated decent. The vapor change is defined as:

$$dq_v = \begin{cases} dq_m & \text{for } q_v > q_s \\ E_p - \min(dq_m + (q_s - q_v), q_l + q_i) & \text{for } q_v \leqslant q_s \end{cases}, \tag{A12}$$

where q_s is the phase weighted saturation mixing ratio, E_p is an assumed precipitation vaporization efficiency, taken here to be $E_p = 0.5$, and q_m is the change in vapor mixing ratio due to phase change over one grid volume which can be estimated (Cotton and Tripoli, 1978) as:

$$dq_m = \frac{g\Delta zq_v}{RT}\frac{\dfrac{\varepsilon L_m}{c_pT} - 1}{\dfrac{\varepsilon L_mq_v}{Rc_pT^2} - 1}, \tag{A13}$$

where R is the gas constant and $\varepsilon = 0.622$.

The Brunt-Vaisala frequency implied by Eq. A9, is also used to define the Richardson number. This from of the Brunt-Vaisala frequency has been found to effectively force deep mixing plumes, which represent cumulus reasonably well, when the circulation is partially resolved by the grid.

Appendix B

Rankine Vortex Initialization

The initial wind circulation is built from a specified vorticity field by solving the Poisson equation for stream function. The vorticity field is given by:

$$\zeta = \begin{cases} w_\zeta(x_3)(\zeta_1 - \zeta_2) & \text{for } R < R_2 \\ 0 & \text{for } R \geqslant R_2 \end{cases}, \tag{B1}$$

where ζ, ζ_1, and ζ_2 are relative vorticity values for the vortex and two components comprising the vortex respectively and R and $R_2 = 800$ km are the great circle distances to the vortex center for an arbitrary location and the location and the outer limits of the circulation respectively. The vortex strength is varied with height by the relationship:

$$w_\zeta(x_3) = \begin{cases} 1 & \text{for } x_3 < z_1 \\ e^{(-x_3 - z_1/z_2 - z_1)} & \text{for } x_3 \geqslant z_1 \end{cases}, \tag{B2}$$

where $z_1 = 10$ km and $z_2 = 13$ km. The inner vorticity component is defined:

$$\zeta_1 = \zeta_{\max}e^{\ln(0.1)(R/R_1)^4}, \tag{B3}$$

where $R_1 = 400$ km is the radius by which the inner vorticity is reduced by one order of magnitude, and $\zeta_{\max} = 2.0 \times 10^{-4}$ is the center vorticity value.

The outer vorticity field is added to force the average vorticity added by the perturbation to zero and is given by:

$$\zeta_2 = \sin\left(\pi\frac{R}{R_2}\right)\frac{[\zeta_1]}{\left[\sin\left(\pi\dfrac{R}{R_2}\right)\right]} \tag{B4}$$

where the horizontal averaging operator is defined for some arbitrary variable Ψ to be:

$$[\Psi] = \frac{\int_A \Psi dA}{\int_A dA}, \tag{B5}$$

where A is the area over which the average is taken, which in this case, is the area within R_2.

References

Arakawa, A., Schubert, W. H., 1974: Interaction of a cumulus ensemble with the large scale environment. Part I. *J. Atmos. Sci.*, **31**, 674–701.

Challa, M., Pfeffer, R. L., 1980: Effects of eddy fluxes of angular momentum on model hurricane development. *J. Atmos. Sci.*, **37**, 1603–1618.

Charney, J., Eliassen, A., 1964: On the growth of the hurricane depression. *J. Atmos. Sci.*, **21**, 68–75.

Chen, C., Cotton, W. R., 1983: A one-dimensional simulation of the stratocumuluscapped mixed layer. *Bound.-Layer Meteor.*, **25**, 289–321.

Clark, T. L., 1977: A small scale dynamic model using a terrain-following coordinate transformation. *J. Comp. Phys.*, **24**, 186–215.

Clark, T. L., Farley, R. D., 1984: Severe downslope windstorm calculations in two and three spatial dimensions using anelastic interactive grid nesting: A possible mechanism for gustiness. *J. Atmos. Sci.*, **41**, 329–350.

Cotton, W. R., Tripoli, G. J., 1978: Cumulus convection in shear flow – three-dimensional numerical experiments. *J. Atmos. Sci.*, **35**, 1503–1521.

Cotton, W. R., Tripoli, G. J., Rauber, R. M., Mulvihill, E. A., 1986: Numerical simulation of the effects of varying ice crystal nucleation rates and aggregation on the processes on orographic snowfall. *J. Climate. Appl. Meteor.*, **25**, 1658–1680.

Delsol, F., Miyakoda, K., Clarke, R. H., 1971: Parameterized processes in the surface boundary layer of an atmospheric circulation model. *Quart. J. Roy. Meteor. Soc.*, **97**, 181–208.

Durran, D. R., 1981: The effects of moisture on mountain lee waves. Technical Report PhD Thesis NTIS PB82156621, Massachusetts Institute of Technology.

Fritsch, J. M., Chappel, C. F., 1980: Numerical simulation of convectively driven mesoscale pressure systems. Part I. Convective parameterization. *J. Atmos. Sci.*, **37**, 1722–1733.

Holland, G. J., 1983: Angular momentum transports in tropical cyclones. *Quart. J. Roy. Meteor. Soc.*, **109**, 187–209.

Kanak, K. M., 1990: *Three-Dimensional Nonhydrostatic Numerical Simulation of a Developing Tropical Cyclone.* Master's thesis, University of Wisconsin-Madison.

Kanak, K. M., Tripoli, G. J., 1990: Three-dimensional, nonhydrostatic simulation of scale interaction within a tropical cyclone. In: *Preprints, Fourth Conference on Mesoscale Processes, Boulder, Colorado*, pp. 28 – 29. Boston, MA: American Meteorological Society.

Klemp, J. B., Wilhelmson, R. B., 1978: The simulation of three-dimensional convective storm dynamics. *J. Atmos. Sci.*, **35**, 1070–1096.

Kreitzberg, C. W., Perky, D. J., 1976: Release of potential instability: Part I. A sequential plume model within a hydrostatic primitive equation model. *J. Atmos. Sci.*, **33**, 456–475.

Kurihara, Y., 1976: On the development of spiral bands in tropical cyclones. *J. Atmos. Sci.*, **33**, 940–958.

Lindzen, R. S., 1974: Wave-CISK in the tropics. *J. Atmos. Sci.*, **31**, 156–179.

Louis, J. F., 1979: A parameteric model of vertical eddy fluxes in the atmosphere. *Bound.-Layer Meteor.*, **17**, 187–202.

Ooyama, K., 1964: A dynamical model for the study of tropical cyclone development. *Geofis. Intern.*, **4**, 187–198.

Raymond, D. J., 1975: A model for predicting the movement of continuously propagating convective storms. *J. Atmos. Sci.*, **32**, 1308–1317.

Raymond, D. J., 1976: Wave-CISK and convective mesosystems. *J. Atmos. Sci.*, **33**, 2392–2398.

Raymond, D. J., 1984: A Wave-CISK model of squall lines. *J. Atmos. Sci.*, **41**, 1946–1948.

Redelsperger, J. L., Sommeria, G., 1981: Methode de representation de la turbulence d'echelle inferieure a la maille pour un modele tri-dimensional de convection nuageuse. *Bound.-Layer Meteor.*, **21**, 509–529.

Redelsperger, J. L., Sommeria, G., 1982: Methode de representation de la turbulence associe aux precipitations dans unmodele tri-dimensional de convection nuageuse. *Bound.-Layer Meteor.*, **24**, 231–252.

Rosenthal, S. L., 1978: Numerical simulation of a tropical cyclone development with latent heat release by the resolvable scales I: Model description and preliminary results. *J. Atmos. Sci.*, **35**, 258–271.

Schubert, W. H., Hack, J. J., 1982: Inertial stability and tropical cyclone development. *J. Atmos. Sci.*, **39**, 1687–1697.

Stull, R. B., 1989: *An Introduction to Boundary Layer Meteorology.* Kluger Academic press.

Tremback, C. J., 1990: *Numerical Simulation of a Convective Comple: Model Development and Numerical Results.* PhD thesis, Colorado State University, Fort Collins, CO 80523.

Tripoli, G. J., 1992: A nonhydrostatic numerical model designed to simulate scale interaction. *Mon. Wea. Rev.*, **120**, (in press).

Tripoli, G. J., Cotton, W. R., 1982: The Colorado State University three-dimensional model – 1982. Part I: General theoretical framework and sensitivity experiments. *J. Rech. Atmos.*, **16**, 185–220.

Tripoli, G. J., Cotton, W. R., 1989: Numerical study of an observed orogenic mesoscale convective system. Part 1: Simulated genesis and comparison with observations. *Mon. Wea. Rev.*, **117**, 273–304.

Willoughby, H. E., 1977: Inertia-buoyancy waves in hurricanes. *J. Atmos. Sci.*, **34**, 1028–1039.

Willoughby, H. E., 1978: A possible mechanism for the formation of hurricane rainbands. *J. Atmos. Sci.*, **35**, 838–848.

Willoughby, H. E., 1979: Excitation of spiral bands in hurricanes by the interaction between the symmetric mean vortex and a shearing environmental steering current. *J. Atmos. Sci.*, **36**, 1226–1235.

Author's address: G. J. Tripoli, University of Wisconsin-Madison, Department of Meteorology, Madison, Wisconsin 53706, U.S.A.

Verleger: Springer-Verlag KG, Sachsenplatz 4—6, A-1201 Wien. — Herausgeber: Prof. Dr. Elmar R. Reiter, Wels Research Corporation, Boulder, CO 80301, U.S.A. — Redaktion: Türkenschanzstraße 18, A-1180 Wien. — Satz und Umbruch: Thomson Press (India) Ltd., New Delhi; Reproduktion und Offsetdruck: Druckerei G. Grasl, A-2540 Bad Vöslau. — Verlagsort: Wien. — Herstellungsort: Bad Vöslau. — Printed in Austria.

Offenlegung gem. § 25 Abs. 1 bis 3 Mediengesetz
Unternehmensgegenstand: Verlag von wissenschaftlichen Büchern und Zeitschriften.
An der Springer-Verlag KG ist beteiligt: Dr. Konrad F. Springer, Sachsenplatz 4—6, A-1201 Wien, als Kommanditist zu 52,38%. Geschäftsführer: Dr. Konrad F. Springer, Prof. Dr. Dietrich Götze, Ing. Wolfram F. Joos, Dipl.-Kfm. Claus Michaletz und Rudolf Siegle, alle Sachsenplatz 4—6, A-1201 Wien.

N15

A Lattice Model for Data Display

William L. Hibbard[1&2], Charles R. Dyer[2] and Brian E. Paul[1]
[1]Space Science and Engineering Center
[2]Computer Sciences Department
University of Wisconsin - Madison

Abstract

In order to develop a foundation for visualization, we develop lattice models for data objects and displays that focus on the fact that data objects are approximations to mathematical objects and real displays are approximations to ideal displays. These lattice models give us a way to quantize the information content of data and displays and to define conditions on the visualization mappings from data to displays. Mappings satisfy these conditions if and only if they are lattice isomorphisms. We show how to apply this result to scientific data and display models, and discuss how it might be applied to recursively defined data types appropriate for complex information processing.

1 Introduction

Robertson et.al. have described the need for formal models that can serve as a foundation for visualization techniques and systems [13]. Models can be developed for data (e.g., the fiber bundle data model [4] describes the data objects that computational scientists use to approximate functions between differentiable manifolds), displays (e.g., Bertin's detailed analysis of static 2-D displays [1]), users (i.e., their tasks and capabilities), computations (i.e., how computations are expressed and executed), and hardware devices (i.e., their capabilities).

Here we focus on the process of transforming data into displays. We define a data model as a set U of data objects, a display model as a set V of displays, and a visualization process as a function D: $U \rightarrow V$. The usual approach to visualization is synthetic, constructing the function D from simpler functions. The function may be synthesized using rendering pipelines [5, 11, 12], defining different pipelines appropriate for different types of data objects within U. Object oriented programming may be used to synthesize a polymorphic function D [9, 15] that applies to multiple data types within U.

We will try to address the need for a formal foundation for visualization by taking an analytic approach to defining D. Since an arbitrary function D: $U \rightarrow V$ will not produce displays $D(u)$ that effectively communicate the information content of data objects $u \in U$, we seek to define conditions on D to ensure that it does. For example, we may require that D be injective (i.e., one-to-one), so that no two data objects have the same display. However, this is clearly not enough. If we let U and V both be the set of images of 512 by 512 pixels with 24 bits of color per pixel, then any permutation of U can be interpreted as an injective function D from U to V. But an arbitrary permutation of images will not effectively communicate information. Thus we need to define stronger conditions on the function D. Our investigation depends on some complex mathematics, although we will only present the conclusions in this paper. The details are available in [7].

2 Lattices as data and display models

The purpose of data visualization is to communicate the information content of data objects in displays. Thus if we can quantify the information content of data objects and displays this may give us a way to define conditions on the visualization function D. The issue of information content has already been addressed in the study of programming language semantics [14], which seeks to assign meanings to programs. This issue arises because there is no algorithmic way to separate non-terminating programs from terminating programs, so the set of meanings of programs must include an *undefined* value for non-terminating programs. This value contains less information (i.e., is less precise) than any of the values that a program might produce if it terminates, and thus introduces an order relation based on information content into the set of program meanings. In order to define a correspondence between the ways that programs are constructed, and the sets of meanings of programs, Scott

PRECEDING PAGE BLANK NOT FILMED

developed an elegant lattice theory for the meanings of programs [16].

Scientists have data with undefined values, although their sources are numerical problems and failures of observing instruments rather than non-terminating computations. An undefined value for pixels in satellite images contains less information than valid pixel radiances and thus creates an order relation between data values. Data are often accompanied by metadata [18] that describe their accuracy, for example as error bars, and these accuracy estimates also create order relations between data values based on information content (i.e., precision). Finally, array data objects are often approximations to functions, as for example a satellite image is a finite approximation (i.e., a finite sampling in both space and radiance) to a continuous radiance field, and such arrays may be ordered based on the resolution with which they sample functions.

In general scientists use computer data objects as finite approximations to the objects of their mathematical models, which contain infinite precision numbers and functions with infinite ranges. Thus metadata for missing data indicators, numerical accuracy and function sampling are really central to the meaning of scientific data and should play an important role in a data model. We define a data model U as a lattice of data objects, ordered by how precisely they approximate mathematical objects. To say that U is a *lattice* [2] means that there is a partial order on U (i.e., a binary relation such that, for all $u_1, u_2, u_3 \in U$, $u_1 \le u_1$, $u_1 \le u_2$ & $u_2 \le u_1 \Rightarrow u_1 = u_2$ and $u_1 \le u_2$ & $u_2 \le u_3 \Rightarrow u_1 \le u_3$) and that any pair $u_1, u_2 \in U$ have a least upper bound (denoted by $u_1 \vee u_2$) and a greatest lower bound (denoted by $u_1 \wedge u_2$).

The notion of precision of approximation also applies to displays. They have finite resolutions in space, color and time (i.e., animation). 2-D images and 3-D volume renderings are composed of finite numbers of pixels and voxels and are finite approximations to idealized mathematical displays. Thus we will assume that our display model V is a lattice and that displays are ordered according to their information content (i.e., precision of approximation to ideal displays). In Sections 4 and 5 we will present examples of scientific data and display lattices.

We assume that U and V are complete lattices, so that they contain the mathematical objects and ideal displays that are limits of sets of data objects and real displays (a lattice is *complete* if any subset has a least upper bound and a greatest lower bound). Just as we study functions of rational numbers in the context of functions of real numbers (the completion of the rational numbers), we will study visualization functions between the complete lattices U and V, recognizing that data objects and real displays are restricted to countable subsets of U and V.

3 Conditions on visualization functions

The lattice structures of U and V provide a way to quantize information content and thus to define conditions on functions of the form D: $U \rightarrow V$. In order to define these conditions we draw on the work of Mackinlay [10]. He studied the problem of automatically generating displays of relational information and defined *expressiveness conditions* on the mapping from relational data to displays. His conditions specify that a display expresses a set of facts (i.e., an instance of a set of relations) if the display encodes all the facts in the set, and encodes only those facts.

In order to interpret the expressiveness conditions we define a fact about data objects as a logical predicate applied to U (i.e., a function of the form P: $U \rightarrow$ {*false*, *true*}). However, since data objects are approximations to mathematical objects, we should avoid predicates such that providing more precise information about a mathematical object (i.e., going from u_1 to u_2 where $u_1 \le u_2$) changes the truth value of the predicate (e.g., $P(u_1)$ = *true* but $P(u_2)$ = *false*). Thus we consider predicates that take values in {*undefined*, *false*, *true*} (where *undefined* < *false* and *undefined* < *true*), and we require predicates to preserve information ordering (that is, if $u_1 \le u_2$ then $P(u_1) \le P(u_2)$; functions that preserve order are called *monotone*). We also observe that a predicate of the form P: $U \rightarrow$ {*undefined*, *false*, *true*} can be expressed in terms of two predicates of the form P: $U \rightarrow$ {*undefined*, *true*}, so we will limit facts about data objects to monotone predicates of the form P: $U \rightarrow$ {*undefined*, *true*}.

The first part of the expressiveness conditions says that every fact about data objects is encoded by a fact about their displays. We interpret this as follows:

Condition 1. For every monotone predicate P: $U \rightarrow$ {*undefined*, *true*}, there is a monotone predicate Q: $V \rightarrow$ {*undefined*, *true*} such that $P(u) = Q(D(u))$ for each $u \in U$.

This requires that D be injective (if $u_1 \neq u_2$ then there are P such that $P(u_1) \neq P(u_2)$, but if $D(u_1) = D(u_2)$ then $Q(D(u_1)) = Q(D(u_2))$ for all Q, so we must have $D(u_1) \neq D(u_2)$).

The second part of the expressiveness conditions says that every fact about displays encodes a fact about data objects. We interpret this as follows:

Condition 2. For every monotone predicate Q: $V \rightarrow \{undefined, true\}$, there is a monotone predicate P: $U \rightarrow \{undefined, true\}$ such that $Q(v) = P(D^{-1}(v))$ for each $v \in V$.

This requires that D^{-1} be a function from V to U, and hence that D be bijective (i.e., one-to-one and onto). However, it is too strong to require that a data model realize every possible display. Since U is a complete lattice it contains a maximal data object X (the least upper bound of all members of U). Then $D(X)$ is the display of X and the notation $\downarrow D(X)$ represents the complete lattice of all displays less than $D(X)$. We modify Condition 2 as follows:

Condition 2'. For every monotone predicate Q: $\downarrow D(X) \rightarrow \{undefined, true\}$, there is a monotone predicate P: $U \rightarrow \{undefined, true\}$ such that $Q(v) = P(D^{-1}(v))$ for each $v \in \downarrow D(X)$.

These conditions quantify the relation between the information content of data objects and the information content of their displays. We use them to define a class of functions:

Definition. A function D: $U \rightarrow V$ is a *display function* if it satisfies Conditions 1 and 2'.

In [7] we prove the following result about display functions:

Proposition 1. A function D: $U \rightarrow V$ is a display function if and only if it is a lattice isomorphism from U onto $\downarrow D(X)$ [i.e., for all $u_1, u_2 \in U$, $D(u_1 \vee u_2) = D(u_1) \vee D(u_2)$ and $D(u_1 \wedge u_2) = D(u_1) \wedge D(u_2)$].

This result may be applied to any complete lattice models of data and displays. In the next three sections we will explore its consequences in one setting.

4 A Scientific data model

We will develop a scientific data model that integrates metadata for missing data indicators, numerical accuracy and function sampling. We will develop this data model in terms of a set of data types, starting with *scalar* types used to represent the primitive variables of mathematical models. Given a scalar type s, let I_s denote the set of possible values of a data object of the type s. First we define *continuous* scalars to represent real variables, such as *time*, *temperature* and *latitude*. If s is continuous then I_s includes the undefined value, which we denote by the symbol $\perp$ (usually used to denote the least element of a lattice), and also includes all closed real intervals. We interpret the closed real interval $[x, y]$ as an approximation to an actual value that lies between x and y. In our lattice structure, these intervals are ordered by the *inverse* of set containment, since a smaller interval provides more precise information than a containing interval. Figure 1 illustrates the order relation on a continuous scalar type. Of course, an actual implementation can only include a countable number of closed real intervals (such as the set of rational intervals).

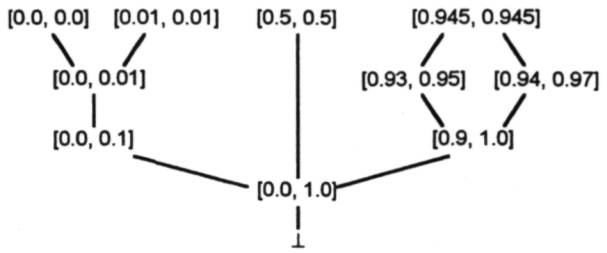

Figure 1. The order relations among a few values of a continuous scalar.

We also define *discrete* scalars to represent integer and string variables, such as *year*, *frequency_count* and *satellite_name*. If s is discrete then I_s includes $\perp$ and a countable set of incomparable values (no integer is more precise than any other integer). Figure 2 illustrates the order relation on a discrete scalar type.

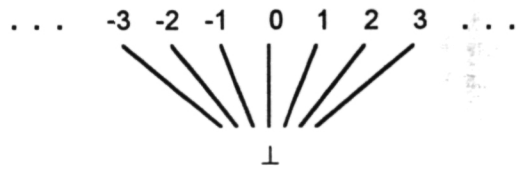

Figure 2. The order relations among a few values of a discrete scalar.

Complex data types are constructed from scalar data types as arrays and tuples. An array data type represents a function between mathematical variables. For example, a function from *time* to *temperature* is approximated by data objects of the type (*array* [*time*] *of temperature*;). We say that *time* is the *domain* type of this array, and *temperature* is its *range* type. Values of an array type are sets of 2-tuples that are (domain, range) pairs. The set {([1.1, 1.6], [3.1, 3.4]), ([3.6, 4.1], [5.0, 5.2]), ([6.1, 6.4], [6.2, 6.5])} is an array data object that contains three samples of a function from *time* to *temperature*. The domain value of a sample lies in the first interval of a pair and the range value lies in the second interval of a pair, as illustrated in Figure 3. Adding more samples, or

increasing the precision of samples, will create a more precise approximation to the function. Figure 4 illustrates the order relation on an array data type. The domain of an array must be a scalar type, but its range may be any scalar or complex type (its definition may not include the array's domain type).

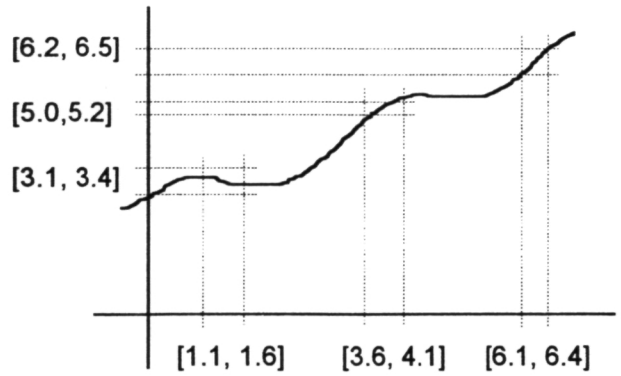

Figure 3. An array samples a real function as a set of pairs of intervals.

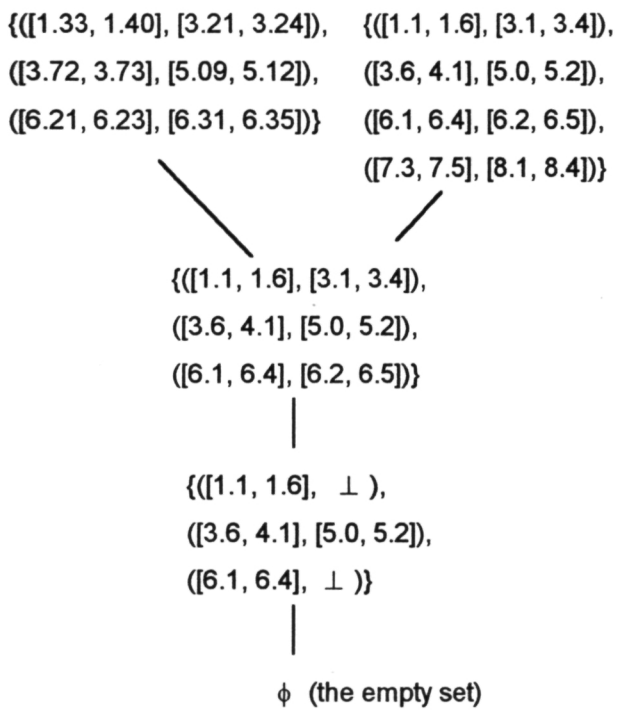

Figure 4. The order relations among a few arrays.

Tuple data types represent tuples of mathematical objects. For example, a 2-tuple of values for temperature and pressure is represented by data objects of the type *struct*{*temperature*; *pressure*;}. Data objects of this type are 2-tuples (*temp*, *pres*) where $temp \in I_{temperature}$ and $pres \in I_{pressure}$. We say that *temperature* and *pressure* are *element* types of the tuple. The elements of a tuple type may be any complex types (they must be defined from disjoint sets of scalars). A tuple data object x is less than or equal to a tuple data object y if every element of x is less than or equal to the corresponding element of y, as illustrated in Figure 5.

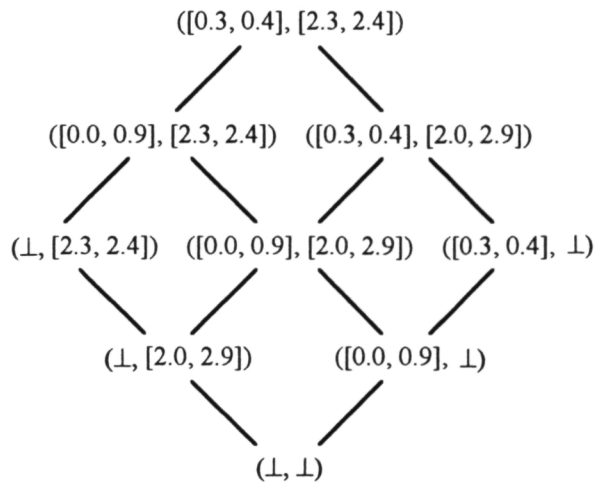

Figure 5. The order relations among a few tuples.

This data model is applied to a particular application by defining a finite set S of scalar types (these would represent the primitive variables of the application), and defining T as the set of all types that can be constructed as arrays and tuples from the scalar types in S. For each type $t \in T$ we can define a countable set H_t of data objects of type t (these correspond to the data objects that are realized by an implementation).

In order to apply our lattice theory to this data model, we must define a single lattice U and embed each H_t in U. First define $X = \mathbf{X}\{I_s \mid s \in S\}$ as the cross product of the value sets of the scalars in S. Its members are tuples with one value from each scalar in S, ordered as illustrated in Figure 5. Now we would like to define U as the power set of X (i.e., the set of all subsets of X). However, power sets have been studied for the semantics of parallel languages and there is a well known problem with constructing order relations on power sets [14]. We expect this order relation to be consistent with the order relation on X and also consistent with set containment. For example, if $a, b \in X$ and $a < b$, we would expect that $\{a\} < \{b\}$. Thus we might define an order relation between subsets of X by:

(1) $\forall A, B \subseteq X.\ (A \leq B \Leftrightarrow \forall a \in A.\ \exists b \in B.\ a \leq b)$

However, given $a < b$, (1) implies that $\{b\} \leq \{a, b\}$ and $\{a, b\} \leq \{b\}$ are both true, which contradicts $\{b\} \neq$

$\{a, b\}$. This problem can be resolved by restricting the lattice U to sets of tuples such every tuple is maximal in the set. That is, a set $A \subseteq X$ belongs to the lattice U if $a < b$ is not true for any pair $a, b \in A$. The members of U are ordered by (1), as illustrated in Fig. 6, and form a complete lattice (see [7] for more details).

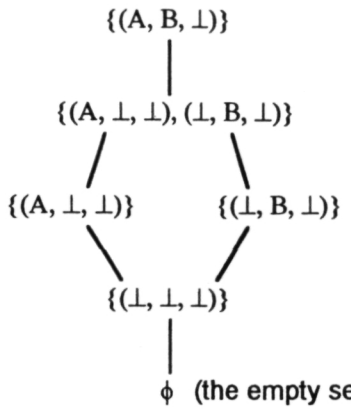

Figure 6. The order relations among a few members of a data lattice *U* defined by three scalars.

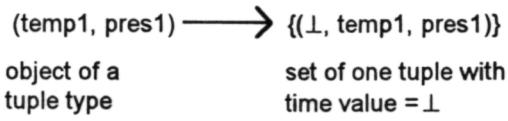

Figure 7. An embedding of a tuple type into a lattice.

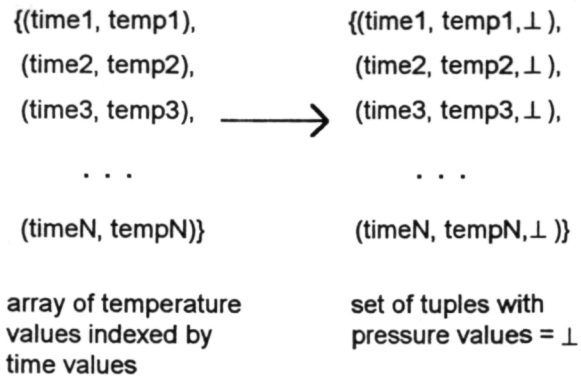

Figure 8. An embedding of an array type into a lattice.

To see how the data objects in H_t are embedded in U, consider a data lattice U defined from the three scalars *time*, *temperature* and *pressure*. Objects in the lattice U are sets of tuple of the form (*time*, *temperature*, *pressure*). We can define a tuple data type *struct*{*temperature*; *pressure*;}. A data object of this type is a tuple of the form (*temp*, *pres*) and can be mapped to a set of tuples (actually, it is a set consisting of one tuple) in U with the form $\{(\perp, temp, pres)\}$. This embeds the tuple data type in the lattice U, as illustrated in Figure 7.

Similarly, we can embed array data types in the data lattice. For example, consider an array data type (*array* [*time*] *of temperature*;). A data object of this type consists of a set of pairs of (*time*, *temp*). This array data object can be embedded in U as a set of tuples of the form (*time*, *temp*, $\perp$). Figure 8 illustrates this embedding. The basic ideas presented in Figs. 7 and 8 can be combined to embed complex data types, defined as hierarchies of tuples and arrays, in data lattices (see [7] for details).

5 A scientific display model

For our scientific display model, we start with Bertin's analysis of static 2-D displays [1]. He modeled displays as sets of graphical marks, where each mark was described by an 8-tuple of graphical primitive values (i.e., two screen coordinates, size, value, texture, color, orientation and shape). The idea of a display as a set of tuple values is quite similar to the way we constructed the data lattice U. Thus we define a finite set DS of *display scalars* to represent graphical primitives, we define $Y = \mathbf{X}\{I_d \mid d \in DS\}$ as the cross product of the value sets of the display scalars in DS, and we define V as the complete lattice of all subsets A of Y such that every tuple is maximal in A.

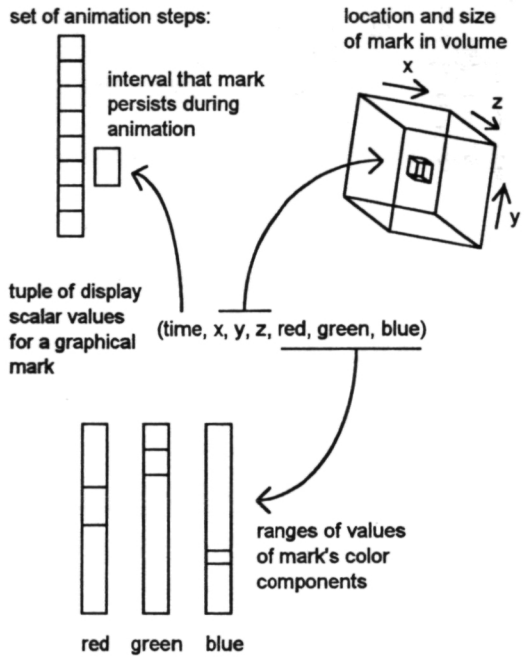

Figure 9. The roles of display scalars in an animated 3-D display model.

We can define a specific lattice V to model animated 3-D displays in terms of a set of seven continuous display scalars: (x, y, z, *red*, *green*, *blue*, *time*}. A tuple of values of these display scalars represents a graphical mark. The interval values of x, y and z represent the locations and sizes of graphical marks in the volume, the interval values of *red*, *green* and *blue* represent the ranges of colors of marks, and the interval values of *time* represent the place and duration of persistence of marks in an animation sequence. This is illustrated in Figure 9. A display in V is a set of tuples, representing a set of graphical marks.

Display scalars can be defined for a wide variety of attributes of graphical marks, and need not be limited to simple values. For example, a discrete display scalar may be an index into a set of complex shapes (i.e., icons).

6 Scalar mapping functions

Proposition 1 said that a function of the form D: $U \rightarrow V$ satisfies the expressiveness conditions (i.e., is a display function) if and only if D is a lattice isomorphism from U onto $\downarrow D(X)$, a sublattice of V. We can now apply this to the scientific data and display lattices described in Section 4 and 5.

The scalar and display scalar types play a special role in characterizing display functions in the context of our scientific models. Given a scalar type $s \in S$, define $U_s \subseteq U$ as the set of embeddings of objects of type s in U. That is, U_s consists of sets of tuples of the form $\{(\perp,...,b,...,\perp)\}$ (this notation indicates that all components of the tuple are $\perp$ except the s component, which is b). Similarly, given a display scalar type $d \in DS$, define $V_d \subseteq V$ as the set of embeddings of objects of type d in V. In [7] we prove the following result:

Proposition 2. If D: $U \rightarrow V$ is a display function, then we can define a mapping MAP_D: $S \rightarrow POWER(DS)$ (this is the power set of DS) such that for all scalars $s \in S$ and all for $a \in U_s$, there is $d \in MAP_D(s)$ such that $D(a) \in V_d$. The values of D on all of U are determined by its values on the scalar embeddings U_s. Furthermore,

(a) If s is discrete and $d \in MAP_D(s)$ then d is discrete,

(b) If s is continuous then $MAP_D(s)$ contains a single continuous display scalar.

(c) If $s \neq s'$ then $MAP_D(s) \cap MAP_D(s') = \phi$.

This tells us that display functions map scalars, which represent primitive variables like *time* and *temperature*, to display scalars, which represent graphical primitives like screen axes and color components. Most displays are already designed in this way, as, for example, a time series of temperatures may be displayed by mapping *time* to one axis and *temperature* to another. The remarkable thing is that Proposition 2 tells us that we don't have to take this way of designing displays as an assumption, but that it is a consequence of a more fundamental set of expressiveness conditions. Figure 10 provides examples of mappings from scalars to display scalars (*lat_lon* is a *real2d* scalar, as described in Section 7).

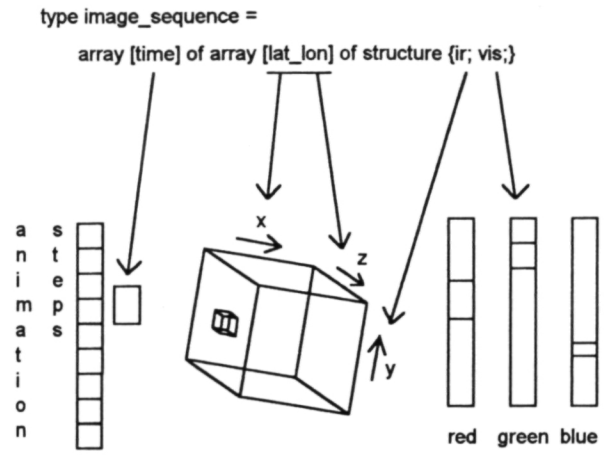

Figure 10. Mappings from scalars to display scalars.

In [7] we present a precise definition (the details are complex) of *scalar mapping functions* and show that D: $U \rightarrow V$ is a display function if and only if it is a scalar mapping function. Here we will just describe the behavior of display functions on continuous scalars. If s is a continuous scalar and $MAP_D(s) = d$, then D maps U_s to V_d. This can be interpreted by a pair of functions g_s:$\mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$ and h_s:$\mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$ (where $\mathbf{R}$ denotes the real numbers) such that for all $\{(\perp,...,[x, y],...,\perp)\}$ in U_s, $D(\{(\perp,...,[x, y],...,\perp)\}) = \{(\perp,...,[g_s(x, y), h_s(x, y)],...,\perp)\}$, which is a member of V_d. Define functions g'_s:$\mathbf{R} \rightarrow \mathbf{R}$ and h'_s:$\mathbf{R} \rightarrow \mathbf{R}$ by $g'_s(z) = g_s(z, z)$ and $h'_s(z) = h_s(z, z)$. Then the functions g_s and h_s can be defined in terms of g'_s and h'_s as follows:

(2) $g_s(x, y) = min\{g'_s(z) \mid x \leq z \leq y\}$ and

(3) $h_s(x, y) = max\{h'_s(z) \mid x \leq z \leq y\}$.

These functions must satisfy the conditions illustrated in Figure 11.

Although the complete lattices U and V include members containing infinite numbers of tuples (these are mathematical objects and ideal displays) in [7] we prove the following:

Proposition 3. Given a display function D: $U \rightarrow V$, a data type $t \in T$ and an embedding of a data object from H_t to $a \in U$, then a contains a finite number of tuples and $D(a) \in V$ contains a finite number of tuples.

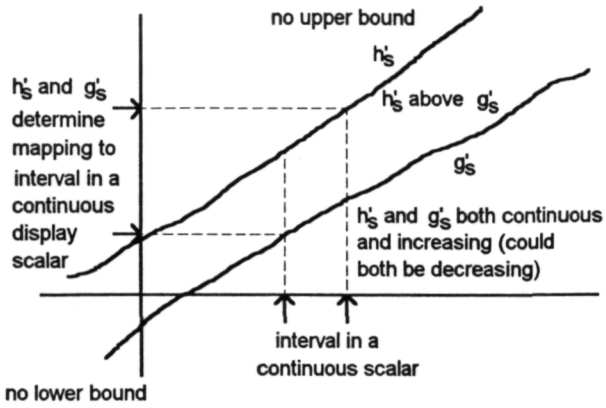

Figure 11. The behavior of a display function D on a continuous scalar interpreted in terms of the behavior of functions h'_S and g'_S.

7 Implementation

The data and display models described in Sections 4 and 5, and the scalar mapping functions described in Section 6, are implemented in our VIS-AD system [6, 8]. This system is intended to help scientists experiment with their algorithms and steer their computations. It includes a programming language that allows users to define scalar and complex data types and to express scientific algorithms. The scalars in this language are classified as *real* (i.e., continuous), *integer* (discrete), *string* (discrete), *real2d* and *real3d*. The *real2d* and *real3d* scalars have no analog in the data model presented in Section 4, but are very useful as the domains of arrays that have non-Cartesian sampling in two and three dimensions. Users control how data are displayed by defining a set of mappings from scalar types (that they declare in their programs) to display scalar types. By defining a set of mappings a user defines a display function D: $U \rightarrow V$ that may be applied to display data objects of any type.

The VIS-AD display model includes the seven display scalars described for animated 3-D displays in Section 5, and also includes display scalars named *contour* and *selector*. Multiple copies of each of these may exist in a display lattice (the numbers of copies are determined by the user's mappings). Scalars mapped to *contour* are depicted by drawing isolevel curves and surfaces through the field defined by the *contour* values in graphical marks. For each *selector* display scalar, the user selects a set of values and only those graphical marks whose *selector* values that overlap this set are displayed. *Contour* is a *real* display scalar and *selector* display scalars take the type of the scalar mapped to them. We plan to add *real* display scalars for *transparency* and *reflectivity* to the system (to be interpreted by complex volume rendering of graphical marks), as well as a *real3d* display scalar for *vector* (to be interpreted by flow rendering techniques).

VIS-AD is available by anonymous ftp from iris.ssec.wisc.edu (144.92.108.63) in the pub/visad directory. Get the README file for complete installation instructions.

8 Recursively defined data types

The data model in Section 4 is adequate for scientific data, but is inadequate for complex information processing which involves *recursively defined data types* [14]. For example, binary trees may be defined by the type *bintree* = *struct*{*bintree*; *bintree*; *value*;} (a leaf node is indicated when both *bintree* elements of the tuple are undefined). Several techniques have been developed to model such data using lattices. In the current context, the most promising is called *universal domains* [3, 17]. Just as we embedded data objects of many different types in the domain U in Section 4, data objects of many different recursively defined data types are embedded in a universal domain (which we also denote by U). However, these embeddings have been defined in order to study programming language semantics, and have a serious problem in the visualization context. Data objects of many different types are mapped to the same member of U. For example, an integer and a function from the integers to the integers may be mapped to the same member of U, and thus any display function of the form D: $U \rightarrow V$ will generate the same display for these two data objects. Thus, in order to extend our lattice theory of visualization to recursively defined data types, other embeddings into universal domains must be developed.

A suitable display lattice V must also be developed such that there exist lattice isomorphisms from a universal domain U into V. Displays involving *diagrams* and *hypertext links* are analogous to the pointers usually used to implement recursively defined data types. Thus the interpretation of V as a set of actual displays may involve these graphical techniques. However, since a large class of recursively defined data types can be embedded in U, and since V is isomorphic to U, these graphical techniques must be applied in a very abstract manner to define a suitable lattice V.

9 Conclusions

It is easy to think of metadata as secondary when we are focused on the task of making visualizations of data. However, it is central to the meaning of scientific data that they are approximations to mathematical objects, and lattices provide a way to integrate metadata about precision of approximation into a data model. By bringing the approximate nature of data and displays into central focus, lattices provide a foundation for understanding the visualization process and an analytic approach to defining the mapping from data to displays. While Proposition 2 just confirms standard practice in designing displays, it is remarkable that this practice can be deduced from the expressiveness conditions.

Although we have not derived any new rendering techniques by using lattices, the high level of abstraction of scalar mapping functions do provide a very flexible user interface for controlling how data are displayed.

There will be considerable technical difficulties in extending this work to recursively defined data types, but we are confident that the results will be interesting.

Acknowledgments

This work was support by NASA grant NAG8-828, and by the National Science Foundation and the Defense Advanced Research Projects Agency under Cooperative Agreement NCR-8919038 with the Corporation for National Research Initiatives.

References

[1] Bertin, J., 1983; Semiology of Graphics. W. J. Berg, Tr. University of Wisconsin Press.

[2] Davey, B. A. and H. A. Priestly, 1990; Introduction to Lattices and Order. Cambridge University Press.

[3] Gunter, C. A. and Scott, D. S., 1990; Semantic domains. In the Handbook of Theoretical Computer Science, Vol. B., J. van Leeuwen ed., The MIT Press/Elsevier, 633-674.

[4] Haber, R. B., B. Lucas and N. Collins, 1991; A data model for scientific visualization with provisions for regular and irregular grids. Proc. Visualization 91. IEEE. 298-305.

[5] Haberli, P., 1988; ConMan: A visual programming language for interactive graphics; Computer Graphics 22(4), 103-111.

[6] Hibbard, W., C. Dyer and B. Paul, 1992; Display of scientific data structures for algorithm visualization. Visualization '92, Boston, IEEE, 139-146.

[7] Hibbard, W. L., and C. R. Dyer, 1994; A lattice theory of data display. Tech. Rep. # 1226, Computer Sciences Department, University of Wisconsin-Madison. Also available as compressed postscript files by anonymous ftp from iris.ssec.wisc.edu (144.92.108.63) in the pub/lattice directory.

[8] Hibbard, W. L., B. E. Paul, D. A. Santek, C. R. Dyer, A. L. Battaiola, and M-F. Voidrot-Martinez, 1994; Interactive visualization of Earth and space science computations. IEEE Computer special July issue on visualization.

[9] Hultquist, J. P. M., and E. L. Raible, 1992; SuperGlue: A programming environment for scientific visualization. Proc. Visualization '92, 243-250.

[10] Mackinlay, J., 1986; Automating the design of graphical presentations of relational information; ACM Transactions on Graphics, 5(2), 110-141.

[11] Nadas, T. and A. Fournier, 1987; GRAPE: An environment to build display processes, Computer Graphics 21(4), 103-111.

[12] Potmesil, M. and E. Hoffert, 1987; FRAMES: Software tools for modeling, animation and rendering of 3D scenes, Computer Graphics 21(4), 75-84.

[13] Robertson, P. K., R. A. Earnshaw, D. Thalman, M. Grave, J. Gallup and E. M. De Jong, 1994; Research issues in the foundations of visualization. Computer Graphics and Applications 14(2), 73-76.

[14] Schmidt, D. A., 1986; Denotational Semantics. Wm.C.Brown.

[15] Schroeder, W. J., W. E. Lorenson, G. D. Montanaro and C. R. Volpe, 1992; VISAGE: An object-oriented scientific visualization system, Proc. Visualization '92, 219-226.

[16] Scott, D. S., 1971; The lattice of flow diagrams. In Symposium on Semantics of Algorithmic Languages, E. Engler. ed. Springer-Verlag, 311-366.

[17] Scott, D. S., 1976; Data types as lattices. Siam J. Comput, 5(3), 522-587.

[18] Treinish, L. A., 1991; SIGGRAPH '90 workshop report: data structure and access software for scientific visualization. Computer Graphics 25(2), 104-118.

Display of Scientific Data Structures for Algorithm Visualization

William Hibbard[1&2], Charles R. Dyer[2] and Brian Paul[1]

[1]Space Science and Engineering Center [2]Department of Computer Sciences
University of Wisconsin-Madison

Abstract

We present a technique for defining graphical depictions for all the data types defined in an algorithm. The ability to display arbitrary combinations of an algorithm's data objects in a common frame of reference, coupled with interactive control of algorithm execution, provides a powerful way to understand algorithm behavior. Type definitions are constrained so that all primitive values occurring in data objects are assigned scalar types. A graphical display, including user interaction with the display, is modeled by a special data type. Mappings from the scalar types into the display model type provide a simple user interface for controlling how all data types are depicted, without the need for type-specific graphics logic.

1: Introduction

Designing scientific algorithms is something of an art. For example, algorithms for extracting useful information from remotely sensed data are based on well understood mathematical and statistical techniques, but often combine these techniques in problem specific ways that can only be determined experimentally. Scientists can usually recognize incorrect results in graphical depictions of the output of their algorithms. To find the source of errors they need a way to apply this same visual understanding to the internal logic of their algorithms. Interactive debugging systems allow scientists to step through program logic and to print the values of program variables and arrays, in order to track down low-level bugs. They need the same sort of interactive capability applied to diagnosing problems with high-level algorithm behavior. However, where low-level logic can be understood from a few printed values, high-level behavior involves masses of data that can only be understood through visualization. Thus there is a need for techniques for generating graphical depictions of the internal data objects of scientific algorithms. In order to be useful to scientists, the user interface for controlling these depictions should be simple and not require graphics expertise.

The data flow architecture [3] is widely used for scientific visualization, with implementations including AVS [9], SGI Explorer, Khoros [7] and apE [2]. It provides a graphical user interface for specifying algorithms as networks of modules. The data flow architecture is popular because of the flexibility of mixing calculation modules with display modules, and because of its easy graphical user interface. However, data flow networks are not generally used for developing detailed algorithms. Current data flow implementations support finite sets of data structures; in order to support algorithm details they would need to support user-definable, application-specific data structures.

The Balsa and Zeus systems [1] provide a set of tools for designing visualization environments for algorithms. Demonstration environments produced using Zeus provide very detailed and effective views of the internal workings of complex algorithms. However, the environment must be custom designed for every algorithm.

The Powervision system [6] uses an object-oriented language to support interactive development of image processing algorithms. The system includes a fixed set of display methods, defined in terms of a set of virtual functions for accessing data objects. As algorithm designers define new object classes, they must ensure that the virtual access functions extend to those classes, and may need to design new display methods for particularly novel classes. The Powervision system exploits object-oriented techniques to reduce the amount of program logic needed to display new object classes, but the system does not eliminate it.

In this paper we describe a technique, that we call the "scalar mapping technique", for generating graphical depictions of the internal data objects of scientific algorithms, without the need for type-specific display logic. We also describe an implementation of this technique in the VIS-AD (VISualization for Algorithm Development) system, an experimental laboratory for developing algorithms.

2: The scalar mapping technique

The scalar mapping technique defines an infinite set of data types that can serve as the types of data objects in a programming language. The data types are defined in such a manner that every primitive value occurring in a data object has one of a finite set of *scalar* types. The technique also models a graphical display, including user

0-8186-2897-9/92 $03.00 © 1992 IEEE

PRECEDING PAGE BLANK NOT FILMED

interaction with the display, as a special data type whose primitive values have one of a finite set of *display scalar* types. Mappings from the scalar types to the display scalar types provide a simple user interface for controlling how all data types are displayed, since the graphical depiction of a data object of any data type can be derived from the scalar mappings.

2.1: Data types

Define T as the set of types for the data objects in an algorithm. It is common for a programming language to define a set of primitive types (e.g. *int*, *real*), and to define a set of type constructors for building the types in T from the primitive types. We modify this by interposing a finite set S of scalar types between T and the primitive types. We define the primitive types as:

$$PRIM = \{int, string, real, real2d, real3d\}$$

where *real2d* and *real3d* are pairs and triples of real numbers. An algorithm designer defines a finite set S of scalar types, and binds them to the primitive types by a function $P: S \rightarrow PRIM$. An infinite set T of types can be defined from S by:

$$S \subset T$$

$$(\text{for } i = 1,\ldots,n. t_i \in T) \Rightarrow (t_1,\ldots,t_n) \in T$$

$$(s \in S \wedge t \in T) \Rightarrow (s \rightarrow t) \in T$$

where $(t_1,\ldots,t_n)$ is a tuple type constructor with element types t_i, and $(s \rightarrow t)$ is an array type constructor with value type t and index type s.

Every primitive value, including an array index value, occurring in a data object of type $t \in T$, has a scalar type in S. This is the key to providing a simple user interface for controlling the display of all algorithm data objects. By defining mappings from the scalar types into a type model of a graphical display, the user can control the way that all data types are displayed.

2.2: Model of a graphical display as a data type

We model a graphical display by defining a special *display* data type. The contents of the display, including the way its contents change in response to user controls, form the value of a data object of type *display*.

The *display* type is defined in terms of a set DS of display scalars:

$$DS = \{x\_axis, y\_axis, z\_axis, xy\_plane, xz\_plane, yz\_plane, xyz\_volume, color, contour\_1, \ldots, contour\_n, animation, selector\_1, \ldots, selector\_m\}$$

The scalar *xyz_volume* has primitive type *real3d* and is a three-dimensional voxel coordinate. The scalars *x_axis*, *y_axis*, *z_axis*, *xy_plane*, *xz_plane* and *yz_plane* are *real* and *real2d* Cartesian factors of *xyz_volume*. The three-dimensional array of voxels is projected onto the two-dimensional display screen, and the projection can be interactively rotated, panned and zoomed under user control. The scalar *color* has primitive type *real3d* and is the color value of a voxel. The scalars *contour_i* have primitive type *real* and are values attached voxels that are depicted by iso-value surfaces or iso-value lines drawn through voxels. The scalar *animation* has primitive type *int* and is the index of an array of voxel volumes that can be rendered in sequence for animation, under user control. The scalars *selector_i* have any of the primitive types (*real*, *real2d*, *real3d*, *int* or *string*) and are indices of display contents. The display contents change in response to user control of *selector_i* values, providing a way for the *display* type to model abstract user interactions with a graphical display.

The *display* type is defined by:

$$voxel = (color, contour\_1, \ldots, contour\_n)$$

$$display = (selector\_1 \rightarrow \ldots(selector\_m \rightarrow (animation \rightarrow (xyz\_volume \rightarrow voxel)))\ldots)$$

Each *voxel* object includes a *color* value and a set of zero or more *contour_i* values that are depicted by iso-level contours. The number of *contour_i* values in the *voxel* tuple is the number of scalars $s \in S$ such that $F(s) = contour$, where the function F is part of the display frame of reference described in Section 2.3. The *voxel* objects are organized into a three-dimensional volume array. An array of voxel volumes is used to model animation, and nested arrays indexed by *selector_i* are used to model user control over the display. The number of *selector_i* indices in the *display* type is the number of scalars $s \in S$ such that $F(s) = selector$.

2.3: Display frame of reference

Types in T are defined in terms of the scalar types S, and the *display* type used to model a graphical display is defined in terms of the display scalar types DS. Mappings from S to DS create a frame of reference for generating graphical depictions of data objects with types in T. A *display frame of reference* is defined by functions:

$$F: S \rightarrow DS \cup \{nil\}$$

$$FD(s): D(s) \rightarrow D(F(s)) \text{ for } s \in S$$

where $D(t)$ is the set of data objects a type t. If $F(s) = nil$ then $FD(s)$ is undefined and the values of s are ignored in the display. The function FD determines how display scalar values are computed from scalar values. The function:

$$DISPLAY(F, FD, t): D(t) \rightarrow D(display)$$

is derived from F and FD, and produces data objects of the *display* type from data objects of any type $t \in T$. The functions F and FD provide a simple way for the user to control the *DISPLAY* function, and therefore to control the display of data objects.

3: The VIS-AD system

The scalar mapping technique is implemented in the VIS-AD system [5], which has been used to demonstrate the effectiveness of the technique for supporting experiments with a variety of algorithms, including an algorithm for discriminating clouds in multi-spectral satellite images.

The VIS-AD system provides a simple syntax for defining the set S of scalar types and the function $P: S \rightarrow PRIM$. The following are examples of scalar types defined for the cloud discrimination algorithm:

```
type brightness = real;
type temperature = real;
type variance = real;
type earth_location = real2d;
type image_region = int;
type time = real;
type count = int;
```

Here *brightness* and *temperature* are the visible and infrared radiance values of pixels in satellite images, *variance* is derived from *temperature*, *earth_location* is a pair of values for the latitude and longitude of pixel locations, *image_region* is an index into rectangular sub-images, *time* is an index for image sequences, and *count* is used for histograms.

The VIS-AD system also provides a simple syntax for defining types in T using the tuple and array type constructors. The keyword *structure* is used for the tuple constructor. The following are examples of complex types defined for the cloud discrimination algorithm:

```
type visir_image =
  array [earth_location] of
    structure {
      .visir_ir = temperature;
      .visir_vis = brightness;
    };
type visir_set= array [image_region] of visir_image;
type visir_set_sequence = array [time] of visir_set;
type vvi_image =
  array [earth_location] of
    structure {
      .vvi_ir = temperature;
      .vvi_var = variance;
      .vvi_vis = brightness;
    };
type vvi_set = array [image_region] of vvi_image;
type var_image = array [earth_location] of variance;
type var_set = array [image_region] of var_image;
type histogram = array [temperature] of count;
type histogram_set =
  array [image_region] of
    structure {
      .hist_location = earth_location;
      .hist_histogram = histogram;
    };
```

Data objects of type *visir_image* are two-dimensional images of *temperature* and *brightness* values, indexed by *earth_location* values. The cloud discrimination algorithm partitions images into regions, and a data object of type *visir_set* is an image with partitions indexed by *image_region* values. A data object of type *visir_set_sequence* is a *time* sequence of partitioned images. The *vvi_image* and *vvi_set* types are similar to the *visir_image* and *visir_set* types, with *temperature*, *variance* and *brightness* values at each image pixel. The *var_image* and *var_set* types are also similar to the *visir_image* and *visir_set* types, with only a *variance* value at each image pixel. A *histogram* data object attaches a frequency *count* to a set of *temperature* values, and a *histogram_set* object contains a *histogram* and an *earth_location* value for each *image_region* value.

The VIS-AD system provides a simple syntax for specifying a display frame of reference. An example of a frame of reference definition is:

```
map earth_location to xz_plane;
map temperature to y_axis;
map brightness to color;
map variance to y_axis;
map time to animation;
map count to x_axis;
map image_region to selector;
```

Each of these *map* statements defines the value of the function F for a single scalar type in S. *Map* statements can also specify values for the FD function; these examples specify none so defaults are used. $F(s)=nil$ for any $s \in S$ that does not occur in a *map* statement.

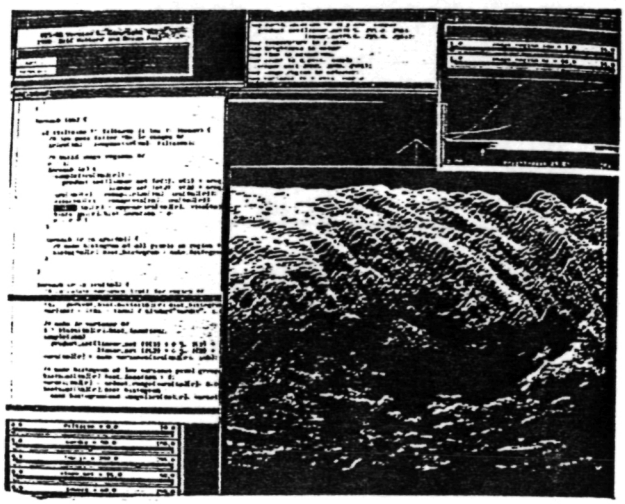

Figure 1: Cloud discrimination algorithm input.

Using this frame of reference, the lower-right window of Figure 1 shows a top-south view of a data object of type *visir_set_sequence*. This data object is the input to the cloud discrimination algorithm. The text editor window on the left shows a section of the cloud discrimination algorithm coded in a language similar to

(See color plates, p. CP-17.)

ORIGINAL PAGE IS
OF POOR QUALITY

C. A data object is selected for display by placing the cursor over any occurrence of its name in this window and clicking a mouse button. Any combination of data objects may be selected for display, and all occurrences of their names are highlighted in reverse video. Execution breakpoints are set and cleared using the mouse in this window, and the next program statement to be executed is highlighted. The small text editor window at the top of the screen contains the display frame of reference. The widgets in the upper-right corner of the screen are used to control animation, to select ranges for scalars mapped to *selector*, to adjust color look-up tables for *real* scalars mapped to *color*, and to select iso-levels for scalars mapped to *contour*.

Since *F(time)=animation* in the frame of reference example, only a single *visir_set* sub-object is displayed in Figure 1. Toggling the ANIMATE widget causes the display to sequence through the object's *visir_set* sub-objects. Since *F(image_region)=selector*, two slider widgets in the upper-right corner are used to select a range of values for *image_region*; all *visir_image* sub-objects are selected in Figure 1. Since *F(brightness)=color*, the pixel colors are functions of their *brightness* values, according to the color widget in the upper-right corner of the screen. Since *F(earth_location)=xz_plane*, the pixels are laid out horizontally. Since *F(temperature)=y_axis*, the *temperature* values of pixels determine their height in the display. The object depiction may be interactively rotated, zoomed and translated in 3-D with simple mouse controls.

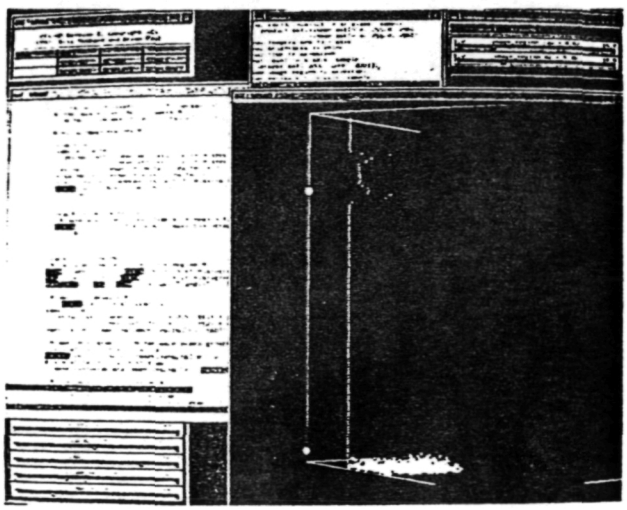

Figure 2: A step in discriminating clouds.

Objects are depicted in monochrome when no object component is mapped to *color*. When several monochrome objects are displayed simultaneously, each object has a different color. Figure 2 shows a south-west view of five monochrome data objects. The tall white graph is an object of type *histogram_set*. The small blue and yellow spheres indicate the values of scalar objects of type *temperature*, calculated by the cloud discrimination algorithm as the 10th and 90th percentiles of the histogram. The purple sphere indicates the value of a scalar object of type *variance*, calculated from the *temperature* percentiles. The blue-green object near the bottom has type *var_set*, and is calculated from another *var_set* object by setting those pixels whose *variance* is greater than the value depicted by the purple sphere to a special *missing* value; these *missing* pixels are invisible in the display. Only a single value of *image_region* is selected in Figure 2, so the depictions of the *histogram_set* and *var_set* objects are restricted to single *histogram* and *var_image* sub-objects.

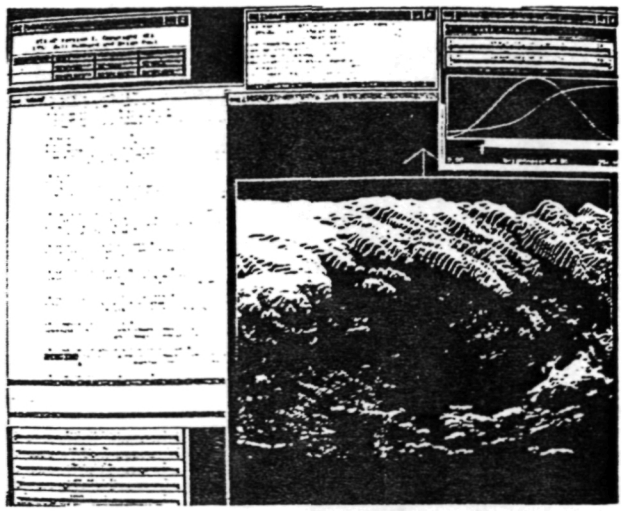

Figure 3: The discriminated clouds.

Figure 3 shows an object of type *visir_set_sequence* that is the output of the cloud discrimination algorithm. It is identical to the object in Figure 1, except that the values of pixels judged by the algorithm not to be in clouds have been set to the *missing* value.

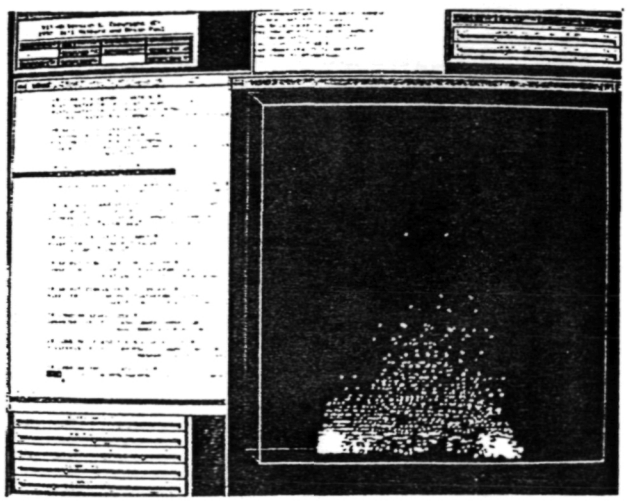

Figure 4: A 3-D scatter diagram of an image.

(See color plates, p. CP-17.)

ORIGINAL PAGE IS
OF POOR QUALITY

A second frame of reference example is:

```
map temperature to x_axis;
map brightness to z_axis;
map variance to y_axis;
map count to y_axis;
map image_region to selector;
map time to animation;
```

Since *earth_location* does not occur in these *map* statements, $F(earth\_location)=nil$ and *earth_location* values are ignored in the display. Using this frame of reference, Figure 4 shows an object of type *vvi_set* as a three-dimensional scatter diagram. The view in Figure 4 shows *temperature* along the horizontal axis and *variance* along the vertical axis, and is restricted to a single *vvi_image* sub-object.

The mappings of *temperature* and *time* in the first frame of reference example can be edited to get:

```
map earth_location to xz_plane;
map temperature to selector;
map brightness to color;
map variance to y_axis;
map time to y_axis;
map count to x_axis;
map image_region to selector;
```

Figure 5 shows the *visir_set_sequence* object from Figure 1 in this frame of reference. Since $F(temperature)=selector$, two slider widgets in the upper-right corner of the screen are used to select a range of values for *temperature*, and the display is restricted to those pixels whose *temperature* values are within the selected range. Since $F(time)=y\_axis$, the object's four *visir_set* sub-objects are stacked along the *y_axis*, showing the motion of cloud features.

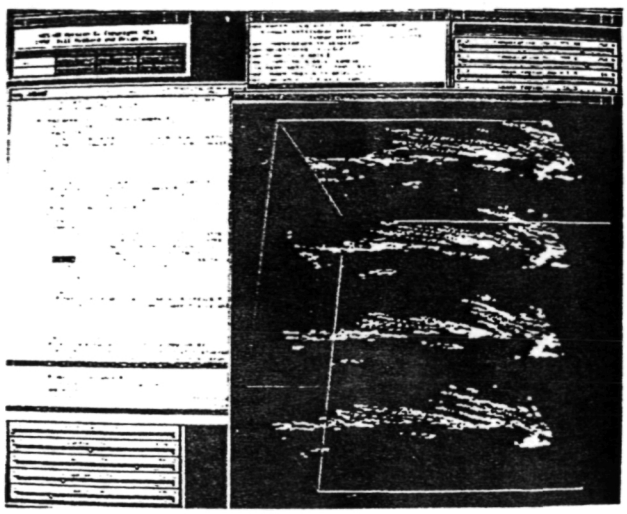

Figure 5: A time sequence image object.

The frame of reference can be edited again to get:

```
map earth_location to xz_plane;
map temperature to color;
map brightness to color;
map variance to y_axis;
map time to animation;
map count to x_axis;
map image_region to selector;
```

Figure 6 shows the *visir_set_sequence* object from Figure 1 in this frame of reference. Since $F(temperature)=color$ and $F(brightness)=color$, the color at each pixel is the average of the colors defined by the look-up tables for *temperature* and *brightness*. The color map widgets show red intensity proportional to *temperature* and blue-green intensity proportional to *brightness*. This way of looking at multi-spectral data is familiar to earth scientists.

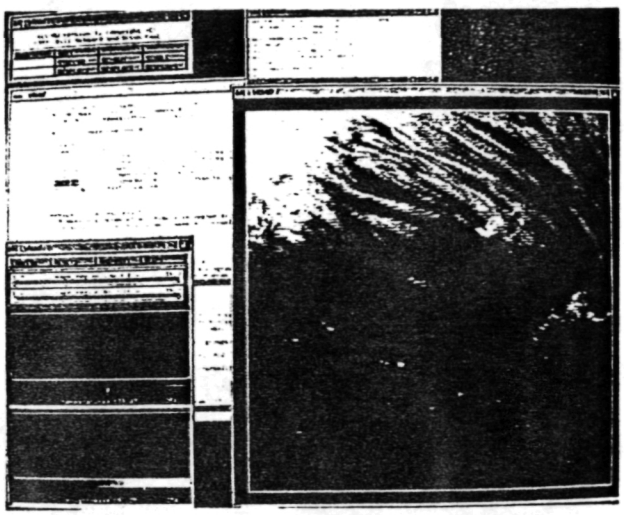

Figure 6: Looking at multiple spectra with color.

The essential feature of the VIS-AD system is its ability to generate displays of any combination of algorithm data objects, in a variety of frames of reference. Editing the algorithm, editing the frame of reference definitions, setting execution breakpoints, starting, stopping and single stepping algorithm execution, and displaying various combinations of data objects, can all be done highly interactively in an integrated environment. Data objects may be displayed in multiple frames of reference simultaneously. If a data object is enabled for display while the algorithm is executing, every time the object is modified it will be flagged for re-display. Thus VIS-AD can be used to produce animations of running algorithms.

4: Data semantics and data display

4.1: Data semantics

The *DISPLAY* function was defined in Section 2.3 as a function from the domain $D(t)$ of data objects of a type $t \in T$ to the domain $D(display)$, so we will describe these domains. The domains of scalar types are determined from the domains of their primitive types, by

(See color plates, p. CP-17.)

$D(s)=D(P(s))$. The domain of the primitive type *int* is the union of a set of finite sub-domains, each an interval of integers, as follows:

$$D(\text{int}_{i,j}) = \{k \mid i \le k \le j\}$$

$$D(\text{int}) = \{missing\} \cup \bigcup_{i \le j} D(\text{int}_{i,j})$$

where i, j and k are integers and the *missing* value indicates the lack of information (the use of special "missing data" codes is common in remote sensing algorithms). The domain of the primitive type *real* is the union of a set of finite sub-domains, each a set of half-open intervals, as follows:

$$D(real_{f,i,j,n}) = \{[f(k/2^n), f((k+1)/2^n)) \mid i \le k \le j\}$$

$$D(real) = \{missing\} \cup \bigcup_{f \in Fld} \bigcup_{i \le j, n \ge 0} D(real_{f,i,j,n})$$

where i, j, k and n are integers and *Fld* is a set of increasing continuous bijections from **R** (the set of real numbers) to **R**; the functions in *Fld* provide non-uniform sampling of *real* values. The domains *D(real2d)*, *D(real3d)* and *D(string)* are similarly defined as the unions of finite sub-domains.

$D((s \to t))$ is defined as the union of a set of function spaces, rather than as the single space of functions from $D(s)$ to $D(t)$, as follows:

$$D((s \to t)) = \{missing\} \cup \bigcup_{subs} (D(s_{subs}) \to D(t))$$

where *subs* ranges over the finite sub-domains of the scalar domain $D(s)$, and $(D(s_{subs}) \to D(t))$ denotes the set of all functions from the set $D(s_{subs})$ to the set $D(t)$. Every array object in $D((s \to t))$ contains a finite set of values from $D(t)$, indexed by values from one of the finite sub-domains of $D(s)$.

The domains of tuple types are defined by:

$$D((t_1,\ldots,t_n)) = \{missing\} \cup D(t_1) \times \ldots \times D(t_n)$$

Each domain $D(t)$ has a lattice structure [8], with the *missing* value as its least element. The half-open intervals in *D(real)* are approximations to values in **R** and are ordered by the inverse of set inclusion; that is, in the lattice structure, an interval is "less" than its sub-intervals. Values in *D(real2d)* and *D(real3d)* form similar lattices and are approximations to values in $\mathbf{R}^2$ and $\mathbf{R}^3$. The lattice structure can be extended to array and tuple types.

The lattice structure of domains, and the definition of array domains as unions of function spaces, provide a formal basis for interpreting array data objects whose indices have primitive types *real*, *real2d* or *real3d* as finite samplings of functions over **R**, $\mathbf{R}^2$ or $\mathbf{R}^3$. For example, a satellite image is a finite sampling of a continuous radiance field. The VIS-AD programming language allows arrays to be indexed by *real*, *real2d* and *real3d* values. Navigation (earth alignment) and calibration (radiance normalization) for satellite images can be implemented by appropriately defined sub-domains of *D(real2d)* and *D(real)*, so that raw satellite images can be accessed directly in terms of latitude, longitude and temperature. An expression like *image*[*location*] is evaluated by resampling the value of *location* to the nearest index value of the *image* array; the expression evaluates to *missing* if *location* is outside the range of index values of *image*. Furthermore, arithmetic expressions evaluate to *missing* if any operand is *missing*. Thus algorithms can combine data from multiple sources without the need for detailed logic for resampling, for checking data boundaries, and for checking for *missing* data. Although VIS-AD implements simple resampling for access to arrays with *real*, *real2d* and *real3d* indices, it would be possible to implement one or more interpolation schemes.

4.2: The *DISPLAY* function

There are two equivalent formulations of the *DISPLAY* function. One formulation composes the *DISPLAY* function from a sequence of basic type transformations [4]. The other is in terms of a tree structure defined for data objects, and is described here. The tree structure $TR(o)$ for objects o is defined recursively as follows:

1. If o is an array containing value objects o_i, $i=1,\ldots,n$, with corresponding scalar index values v_i, then $TR(o)$ is a branch node with sub-nodes $TR(o_i)$, and the value v_i is attached to $TR(o_i)$. If o has the *missing* value, then $TR(o)$ is a leaf node and the *missing* value is attached to $TR(o)$.
2. If o is a tuple containing scalar element objects v_i, $i=1,\ldots,m$, and non-scalar element objects o_i, $i=1,\ldots,n$, then $TR(o)$ is a branch node with sub-nodes $TR(o_i)$, and all the values v_i are attached to each $TR(o_i)$. If $n=0$ (o has only scalar elements) then $TR(o)$ is a leaf node, and the values v_i are attached to that node. If o has the *missing* value, then $TR(o)$ is a leaf node and the *missing* value is attached to $TR(o)$.
3. If o is a scalar not occurring as a tuple element, then $TR(o)$ is a leaf node and the value of o is attached to $TR(o)$.

Define $PATH(o)$ as the set of paths in $TR(o)$ from the root node to any leaf node. For any $p \in PATH(o)$ define $V(p)=v_1v_2\ldots v_n$ as the string of scalar values attached to nodes along the path p. Then a string of display scalar values is calculated from $V(p)$ as:

$$VD(p)=FD(s_1)(v_1)\ldots FD(s_n)(v_n)$$

where $v_i \in D(s_i)$ and where any spatial coordinate display scalar values among the $FD(s_i)(v_i)$ are factored into *x_axis*, *y_axis* and *z_axis* values in $VD(p)$.

The *DISPLAY* function is computed as:

$$DISPLAY(F,FD,t)(o) = COMPOSITE(\{DISP(VD(p))|p \in PATH(o)\})$$

where $DISP(VD(p))$ is a *display* object computed from the string of display scalar values $VD(p)$, and the *COMPOSITE* function computes a single object in $D(display)$ from a set of such objects.

If the leaf node of the path p is generated from an object with the *missing* value, then $DISP(VD(p))=missing$. Otherwise the *DISP* function is computed as follows. Given a string $VD(p)$, for each $s \in DS$ define N_s as the number of values of type s that occur in $VD(p)$. Compute a *voxel* object $vox=(w_{color}, w_{contour\_1}, \ldots, w_{contour\_n})$ as follows:

if $N_{color}=0$ and for i=1,...,n. $N_{contour\_i}=0$
then $vox=(SPECIAL_{color}, missing, \ldots, missing)$
else
  for $s=color, contour\_1, \ldots, contour\_n$
    if $N_s=0$ then $w_s=missing$ else $w_s=(u_1+\ldots+u_N)/N_s$

where $SPECIAL_{color}$ is the monochrome *color* value described in Section 3, and the u_i are the values of type s occurring in $VD(p)$. For $s=x\_axis$, *y_axis* and *z_axis*, compute w_s as follows:

if $N_s=0$ then $w_s=SPECIAL_s$ else $w_s=u_1+\ldots+u_N$

where $SPECIAL_s$ is the spatial coordinate of a distinguished plane perpendicular to the s axis, and the u_i are the values of type s occurring in $VD(p)$. For $s=animation$ and *selector_i*, compute w_s as follows:

If $N_s=0$ then $w_s=D(s)$ else $w_s=u_N$

where $w_s=D(s)$ indicates that all values in $D(s)$ are used for w_s, and u_N is the value of type s occurring farthest from the root in $VD(p)$.

Now the *display* object $d=DISP(VD(p))$ is computed as follows:

$$d[w_{selector\_1}]\ldots[w_{selector\_m}][w_{animation}][w_{x\_axis}][w_{y\_axis}][w_{z\_axis}]=vox$$

If $w_s=D(s)$ was selected for $s=animation$ or *selector_i*, then the equation above applies for all values of w_s in $D(s)$. All other *voxel* sub-objects of d are set to *missing*.

Thus, if the string $VD(p)$ contains exactly one value for each display scalar, then the *color* and *contour_i* values of $VD(p)$ are set in a single non-*missing* *voxel* sub-object of $DISP(VD(p))$, indexed by the spatial, *animation* and *selector_i* values of $VD(p)$. However, undefined and multiply-defined display scalar values are more complex, and the *DISP* function handles them in a way that varies between display scalars. If the value of a spatial coordinate is undefined in $VD(p)$, the depiction of p is embedded in a distinguished plane. However, if the value of $s=animation$ or *selector_i* is undefined in $VD(p)$, sets of *voxel* sub-objects in $DISP(VD(p))$ are set to *vox* so that the depiction of p is invariant to user control of s. Multiply-defined *color* and *contour_i* values are composited by taking their mean, but multiply-defined spatial coordinates are combined by taking their sum, so that, for example, the *histogram* in Figure 2 is positioned over the appropriate image region.

The *COMPOSITE* function computes an object in $D(display)$ from a set of such objects. This computation is done independently for each *voxel* sub-object (i.e. for each combination of *selector_i*, *animation* and spatial values indexing a *voxel* sub-object). The *color* value of a *voxel* is computed as the mean of the non-*missing* *color* values of the corresponding *voxel* sub-objects of the set of objects, and similarly for *contour_i* values. The *COMPOSITE* function is also used to combine depictions of multiple objects into a single display.

4.3: Discussion of data display

Although the spatial coordinate display scalar *xyz_volume* is factored into the one- and two-dimensional Cartesian factors *x_axis*, *y_axis*, *z_axis*, *xy_plane*, *xz_plane* and *yz_plane*, the generated displays need not conform to Cartesian coordinate systems. Two- and three-dimensional scalars may be mapped to spatial display scalars, and the *FD* functions for those scalars may include mathematical coordinate transformations into non-Cartesian coordinate systems. Similarly, three-dimensional scalars may be mapped to the *color* display scalar, and the *FD* functions for those scalars may include mathematical color transformations into color systems other than RGB (Red, Green and Blue).

The scalar mappings provide a flexible tool for projection pursuit for data sets in many dimensions. Given a higher dimensional data set, the user can map different dimensions of the data set to three spatial coordinates, three *color* dimensions, *animation*, and a variable number of *selector* dimensions.

It is certainly possible for the user to define a display frame of reference that produces depictions that poorly communicate the information content of data objects. However, the interactivity of the system allows the user to experiment with the scalar mappings, in order to understand how the mappings work and to find effective object depictions.

Since interactive response times are important, the VIS-AD implementation of the *DISPLAY* function uses shared-memory parallelism and is optimized for vectorization. It traverses paths in an object's tree structure in parallel. It divides the ranges of values of array indices into sections, and the paths through each section are traversed by a different processor. Also, the internal storage format for data objects has been designed to allow efficient vector processing of arrays of scalars and arrays of tuples of scalars. Running on an SGI 340 VGX, the *DISPLAY* function generated each of the figures in this paper in less than one second. This

performance permits interactive visualization of data objects large enough for real scientific algorithms, and smooth animations of the behavior of some algorithms.

Because of the nested arrays in the *display* type, a *display* data object may be very large. The VIS-AD implementation of the *DISPLAY* function minimizes this size by:

1. computing values for only those sub-objects of a *display* object that affect visible screen contents, and re-applying the *DISPLAY* function to data objects as *animation* and *selector* indices change.
2. Splitting the *display* type into two arrays, one for *contour* values and the other for *color* values.
3. Limiting the sampling resolution of *xyz_volume* for the array of *contour* values.
4. Using sparse representations for the array of *color* values; texture maps are used for voxels lying on the distinguished 2-D planes determined by the values $SPECIAL_{x\_axis}$, $SPECIAL_{y\_axis}$ and $SPECIAL_{z\_axis}$, and lists of non-*missing* voxels are used outside of the distinguished planes.

When data objects are transformed into dense sets of non-*missing* voxels it is impossible to see all the voxels, so VIS-AD provides a user-controlled clipping plane for creating a cut-away view of the display.

5: Plans for further development

We are generating a library of standard image analysis and remote sensing functions callable by VIS-AD programs. We are also adapting VIS-AD for distributed execution, enabling programs to call functions on remote computers.

We plan to extend the definition of the *display* type by including *real* display scalars for *transparency* and *reflectivity*, and a *real3d* display scalar for *vector*, in the *voxel* tuple. Scalars mapped to these new display scalars would be depicted by complex volume rendering and flow rendering techniques.

We plan to extend the set *T* of data types by adding type constructors for lists, trees and other complex linked structures. We will extend the *DISPLAY* function to generate diagrams of linked structures, and to provide interaction mechanisms that allow the user to traverse linked structures. In order to do this, linked structures will probably be included in an extended definition of the *display* type.

We plan to extend the parallel algorithm for the *DISPLAY* function to a scalable algorithm running on large numbers of processors, in order to increase interactivity for large data objects.

We plan to adapt VIS-AD to generate graphical execution traces of algorithm data objects, and graphical depictions of the way that algorithm behavior varies with respect to varying algorithm parameters and varying input data sets. These functions are possible because of the flexibility to define arrays of any data type. The system can trace a data object during execution by deriving a new array type of values of the data object, indexed by a scalar for algorithm step number. The system will execute the algorithm and store the value of the selected object in the derived array at user-declared trace points. Similarly, algorithm behavior over an ensemble of invocations can be studied by deriving a new array type of values of a selected algorithm data object, indexed by a scalar for a parameter that varies between algorithm invocations (this may be a *string* scalar for the name of an input data set that varies between invocations). The system would invoke the algorithm for each value of the parameter and save the final value of the selected object in the derived array. By mapping the index scalar of the derived array to a display scalar, the user will be able to generate flexible displays of an execution trace or of the way algorithm behavior varies over an ensemble of invocations.

Acknowledgment

We would like to thank James Dodge and Gregory Wilson for their support. This work was funded by NASA/MSFC (NAG8-828) and NSF (IRI-9022608).

References

[1] Brown, M., and R. Sedgewick, 1984; A system for algorithm animation; Computer Graphics 18(3), 177-186.

[2] Dyer, D., 1990; A dataflow toolkit for visualization; Computer Graphics and Applications, 10(4), 60-69.

[3] Haeberli, P., 1988; ConMan: A visual programming language for interactive graphics; Computer Graphics 22(4), 103-111.

[4] Hibbard, W. and C. Dyer, 1991; Automated display of geometric data types. UW Computer Sciences Technical Report #1015.

[5] Hibbard, W., C. Dyer and B. Paul, 1992; A development environment for data analysis algorithms. Preprints, Conf. Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology. Atlanta, American Meteorology Society. 101-107.

[6] McConnell, C. and D. Lawton, 1988; IU software environments; Proc. IUW, 666-677.

[7] Rasure, J., D. Argiro, T. Sauer, and C. Williams, 1990; A visual language and software development environment for image processing; International J. of Imaging Systems and Technology, Vol. 2, 183-199.

[8] Schmidt, D. A., 1986; Denotational Semantics. Wm. C. Brown Publishers.

[9] Upson, C., T. Faulhaber, Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, A. van Dam, 1989; The application visualization system: a computational environment for scientific visualization; Computer Graphics and Applications, 9(4), 30-42.

Computer Networks and ISDN Systems 22 (1991) 103–109
North-Holland

Interactive atmospheric data access via high-speed networks

NIS

William Hibbard, David Santek

Space Science and Engineering Center, University of Wisconsin-Madison, Madison, WI, USA

Gregory Tripoli

Meteorology Department, University of Wisconsin-Madison, WI, USA

Abstract

Hibbard, W. and D. Santek, Interactive atmospheric data access via high-speed networks, Computer Networks and ISDN Systems 22 (1991) 103–109.

The VIS-5D system running on large workstations lets scientists interactively explore atmospheric simulation data sets containing up to 5×10^7 points. A distributed version of VIS-5D running on a workstation and a supercomputer will make it possible to interactively explore data sets containing up to 10^{10} points. Wide area gigabit networks will bring this capability to scientists at most academic and research institutions. This software will help scientists to interactively develop numerical models of atmospheric phenomena.

Keywords: visualization, networking, interactivity, earth science.

1. Introduction

A basic problem for atmospheric scientists is the sheer size of their data sets. Numerical simulation models normally require several hours on the largest supercomputers and generate output data sets in excess of 10^9 grid points. Model runs for state-of-the-art simulations require hundreds of hours of supercomputer time and generate output data sets containing up to 10^{11} grid points. We are currently providing interactive access to data sets of up to 5×10^7 grid points. A wide area high-speed network will make it possible to extend this access to data sets of up to 10^{10} points. It will also make it possible to interactively develop small simulations.

2. Current applications

The VIS-5D (VISualization of 5-Dimensional data sets) system developed at the University of Wisconsin Space Science and Engineering Center (SSEC) gives scientists interactive access to model output data sets. VIS-5D runs on the Stardent GS-1000 and GS-2000 workstations. It is used to manage and analyze data sets containing up to 10^9 grid points, and to interactively visualize subsets containing up to 5×10^7 grid points [1,2].

The GS-2000 is a high-performance graphics workstation, with a peak vector floating point rate of 80 MFLOPS (million floating point operations per second) and capable of rendering 6×10^5 10-pixel Z-buffered 3-D vectors or 1.5×10^5 100-pixel 2-buffered 3-D Gouraud shaded polygons per second. Z-buffering is the standard hidden surface removal algorithm for 3-D graphics workstations, and Gouraud shading is a standard technique for interpolating shading across a polygon. Memory size is up to 128 Mb (megabytes) with a memory bandwidth of 640 Mbps. The GS-1000 is very similar to the GS-2000, except that the peak floating point rate is 40 MFLOPS.

A numerical forecast of cold fronts moving across the North Atlantic is typical of the model output data sets visualized using VIS-50. This data

0169-7552/91/$03.50 © 1991 – Elsevier Science Publishers B.V. All rights reserved

PRECEDING PAGE BLANK NOT FILMED

set was produced by the European Center for Medium-range Weather Forecasts (ECMWF) using their global weather forecast model. The North Atlantic region, at a resolution of 2.5 degrees, was extracted from their global 1.25 degree resolution grid, at one hour time steps over a period of one week. The data set includes values for pressure, temperature, potential temperature, specific humidity, horizontal wind speed, vertical wind speed, wind divergence and wind vorticity, at each three-dimensional spatial grid point and at each time step, yielding a total data set of about 2.2×10^7 points for visualization. VIS-5D stores these grid point data as scaled 8-bit integers in order to fit into the memory of the GS-2000 for visualization.

Figure 1 shows isobars (constant pressure lines) at a level 0.89 kilometers above sea level, and a semi-transparent surface of constant specific humidity, over a topographical map. The scene shows Europe, Greenland and the North Atlantic, viewed from the south. The western edge of the specific humidity surface marks the approximate location of a cold front moving across the North Atlantic from west to east. This front produced the record storm of February 1988 in England. The model at the ECMWF predicted this storm with great accuracy one week in advance of its development over England. Thus this data set combines the high resolution of a simulation with the accuracy of observations, and provides a valuable look into atmospheric processes.

The best way to understand VIS-5D is to use it interactively. Images like that shown in Fig. 1 are produced at between 5 and 10 per second, with the GS-2000 drawing the images as fast as they are displayed. If the "Animate" widget is selected, then the frames will show a succession of model time steps, and the iso-level contour surfaces and contour lines will show the time evolution of the storm. Independent of whether "Animate" is selected, the user can rotate and zoom the images

William Hibbard is a researcher at the space Science and Engineering Center (SSEC) of the University of Wisconsin-Madison. His research interests are interactive visualization and image processing for earth sciences. He is a member of ACM, SIGGRAPH, SIGART, and the IEEE Computer Society. Hibbard received a BA in Mathematics and an MS in Computer Science from the University of Wisconsin-Madison in 1970 and 1974, respectively. His address is: SSEC, 1225 W. Dayton St., Madison, WI 53706, and email: whibbard@vms.macc.wisc.edu.

David Santek is a scientific programmer at the Space Science and Engineering Center (SSEC) of the University of Wisconsin-Madison. His research interests include satellite data analysis, image analysis, and computer graphics. Santek received a BS in Atmospheric and Oceanic Science from the University of Michigan in 1975 and an MS in Meteorology from the University of Wisconsin in 1978. His address is: SSEC, 1225 W. Dayton St., Madison, WI 53706.

Gregory Tripoli is an Assistant Professor in the Department of Meteorology of the Uiversity of Wisconsin-Madison. His research interests include mesoscale meteorology, cloud dynamics, mesoscale and cloud modeling. Dr Tripoli received BS and MS degrees in Meteorology from Florida State University in 1972 and 1973 respectively, and a PhD in Atmospheric Science from Colorado State University in 1986. His address is Dept. of Meteorology, 1225 W. Dayton St., Madison, WI 53706.

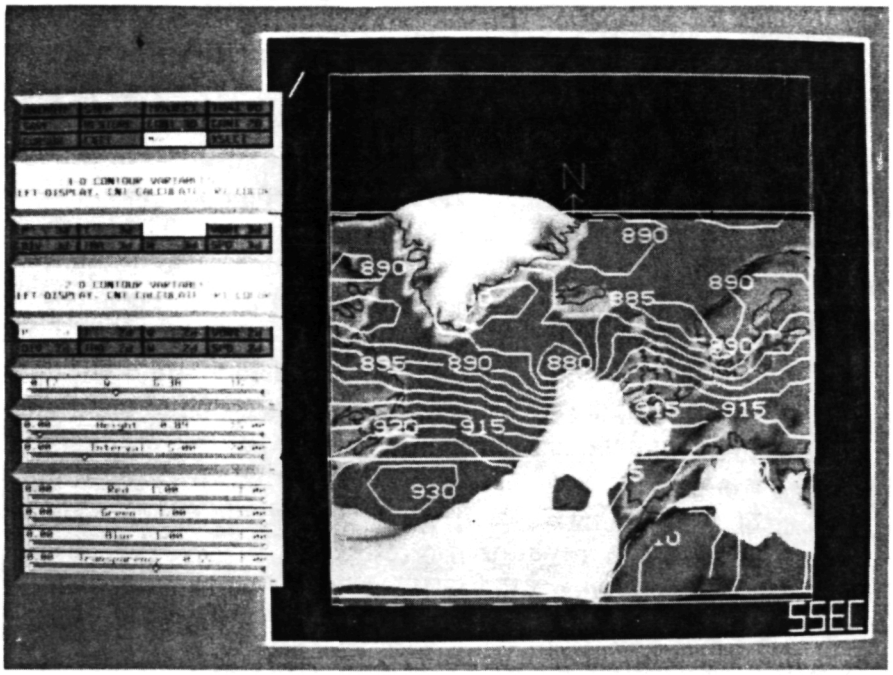

Fig. 1.

in three dimensions, to look at the physics from any angle. The user can also select arbitrary combinations of iso-level contour surfaces, to track down cause and effect relations between physical variables. Figure 1 shows the 6.38 grams per kilogram iso-level contour of specific humidity. The user may change this iso-level value using the slider widget labelled "$Q = 6.38$", and VIS-5D will compute the new surfaces and display them as quickly as possible. Because this cannot be done at the animation rate of between 5 and 10 frames per second, the new surfaces replace the old as they are computed, asynchronously with the animation. In other words, VIS-5D has an asynchronous user interface, which allows the user to explore the data in other ways while waiting for computationally expensive interactions.

The goal of VIS-5D is to give the user highly interactive exploration of very large data sets. In order to provide a high degree of interactivity, the data set must reside in main memory, because of the limited bandwidth of the workstation disk storage. The 128 Mb memory limit of the GS-2000 translates to a limit of 50×10^6 grid points. The computing speed of the GS-2000 also limits the rate at which new iso-level contour surfaces can be computed, to about two time steps per second for data sets such as shown in fig. 1 and 2. Increasing the size limit of data sets, and decreasing the response time for interactions, would both increase the utility of VIS-5D.

3. Interactive access to very large data sets

A high-speed wide-area network would provide a way to extend our visualizations to much larger data sets. The VIS-5D software would be distributed between a supercomputer and a high-performance workstation. This design exploits the large storage capacity and transfer rate of supercomputer disks, the arithmetic speed of the supercomputer, and the rendering performance of the workstaion. It will require network transfer rates in the region of 10^8 bits per second. Like many research institutions, the University of Wisconsin has several high-performance workstations, but it does not have a supercomputer. Wide-area networking is required to make visualization of very large data sets available to the scientists at the University of Wisconsin and at the majority of other institutions.

The distributed version of VIS-5D could be used to visualize a data set such as a hurricane simulation which has been produced using UW-RAMS and 100 hours of time on a CRAY-2. The hurricane data set consists of about 7×10^5 spatial grid points by 10 physical variables by 577

time steps (every 225 seconds for 36 hours). This data set contains 4×10^9 grid points, which will be stored in scaled 8-bit integers for visualization. This simulation provides a view of a relatively large-scale atmospheric phenomenon, namely a hurricane, modelled using the small-scale physics usually only applied to thunderstorms. Thus, interactive visual exploration of this data set would be quite useful.

The distributed version of VIS-5D can be understood by looking at the sequence of operations involved in visualizing the hurricane data set. We would seek to produce interactive animations from this data set at between 5 and 10 frames per second. Like the current version of VIS-5D, the system would compute the images fast enough for the user to perceive motion, and would give the user frame by frame control over the animation. During each frame time, the system would execute the following steps:

(a) The workstation will send to the supercomputer the user's controls for selecting which combination of physical variables to view, for selecting which time step to view, for selecting iso-levels for contouring variables, and for selecting the geographic extents of the region to view.

(b) The supercomputer will read the grids for the selected time step and physical variables from the disk. If we limit the number of simultaneous variables to 3, this would be 5 to 10 frames per second by 3 variables by 7×10^5 bytes per grid requiring between 1.05×10^7 and 2.1×10^7 bytes per second of disk transfer bandwidth. Because supercomputer disks have high enough transfer rates to support this access, the sizes of the data sets to be explored are not limited by main memory size.

(c) The supercomputer will generate subgrids of about 10^5 points each, by subsectoring and possible resolution reduction, according to the user's selection of geographic extents. Current limits on computer power require this reduction in grid size.

(d) The supercomputer will generate polygonal contour surfaces from the subgrids for each selected physical variable, according to the user's selection of iso-levels. The surfaces will contain very roughly 10^5 triangles. This is a heavy computational load and will require polygon-finding algorithms adapted to exploit the parallel and vector facilities of the supercomputer. Generating polygons from grids is adaptable to highly parallel architectures, depending on the speed of distributing the gridded data from the disks to the parallel computing elements, and the speed of collecting polygons for transmission.

(e) The supercomputer will transmit the triangles to the workstation. 10^5 triangles require 7.2×10^6 bytes of storage, so the overall data rate would be 5 to 10 frames per second by 7.2×10^6 bytes of triangles by 8 bits per byte giving between 2.88×10^8 and 5.76×10^8 bits per second. With some compression of the triangle data, this may be reduced to about 10^8 bits per second. The decision about how much compression to apply is a trade off between computational and communications resources.

(f) The workstation will render the triangles according to the user's selections for 3-D pan, zoom and rotation, surface color and transparency, and light source placement. This calls for the workstation to render between 5×10^5 and 10^6 triangles per second. Workstations capable of this rate are now available.

The images generated will include a variety of visual elements other than contour surfaces, such as wind trajectories, contour lines on planes, and maps. We have left these other elements out of the simple operation sequence described above because they present a much smaller computational load than contour surfaces. The polygons and lines depicting the maps can be stored locally in the workstation. The vectors representing trajectories may also fit in the workstation's memory. Contour lines on planes are generated from the gridded data stored in the supercomputer. However, the effort needed to compute them, and the bandwidth needed to transmit them, are small compared to the resources needed for contour surfaces.

4. Network traffic

The actual behavior of the system will vary somewhat, depending on the number of variables selected by the user for display and the complexity of the iso-level contour surfaces generated for those variables. The frame rate for the animation may vary between 2 and 10 frames per second. However, because the animation rate will be inversely linked to the number of polygons in a

frame, the total volume of data per second should be less variable.

A typical workstation session lasts from 30 minutes to 2 hours. The system will generate messages from the workstation to the supercomputer containing the user's controls. These will be sent at the frame rate (between 2 and 10 times per second) and be no longer than 2000 bits each. The messages from the supercomputer containing polygons and vector lines will be very large, with a volume of between 10^8 and 6×10^8 bits divided into as few as 5 or 10 messages per second. The user of interactive visualization will be sensitive to the magnitude and variance of network delays. Anyone who has used a screen editor on a virtual memory system knows how paging delays can disrupt the user's visual interaction with the text.

Standard data compression algorithms may not be effective with the polygon data communicated by this application. However, polygon data can be effectively compressed using techniques based on their special properties. We are assuming that the polygons are triangles, and that they are each represented by the coordinates and surface normals of their three vertices. First, we recognize that triangles can be collected into sequences called polytriangle strips, representing N triangles with $N + 2$ vertices. When N is large, this representation provides a 3 to 1 compression. In general, it is possible to represent iso-level contour surfaces as long polytriangle strips. In fact, this representation is necessary for efficient rendering on the Stardent GS-1000 and GS-2000 workstations. However, this representation can adversely affect rendered image quality for semi-transparent surfaces. The polytriangle representation is also very difficult to produce if polygon generation is running on a highly parallel architecture.

Another compression is available in the numeric precision of the three coordinate and three surface normal components of triangle vertices. They are produced as 32 bit floating point numbers, and the rendering process uses them in that format. However, for intermediate storage and transmission, there is no real loss of information if the vertex coordinates are compressed to 16 bit integers and the vertex normals are compressed to 8 bit integers. This yields an 8 to 3 compression. It is possible to compress further by using differences between successive vertex coordinates, and by encoding normals, which have unit length, as two spherical angles rather than three cartesian components.

Wide area gigabit networks will require large system buffers for storing data for retransmission in case of errors. However, visualization applications can tolerate errors, since any visual information will be replaced quickly and is not used as input to further computing processes. Thus, we would like to see a network that gives some control over error handling to the applications. The application should be able to determine whether to correct any detected errors, and whether the network system should maintain a buffer for retransmission or the application should be responsible to regenerate data for retransmission. This is a resource tradeoff between a system buffer which may contain 10 megabytes and significant computing, and the application designer should have control over that tradeoff.

There are periods during a session when the user is not animating time dynamics or changing the iso-levels of contour surfaces. The system can exploit these periods by storing the most recent polygons in the workstation and transmitting from the supercomputer only when the polygon sets change. Thus the session will alternate between intervals of full bandwidth and virtually zero bandwidth. The duration of these intervals is seconds to minutes.

5. Interactive model development

We are also interested in using distributed algorithms to help scientists interactively develop thunderstorm simulations, a task requiring numerous trial and error adjustments to initial atmospheric conditions and to model parameters. This is done through an iterative cycle of short simulation runs, inspecting the model output data and comparing them to known storm behavior, and adjusting the initial conditions and model parameters.

Interacting with a running model is tricky because of man–machine coupling mismatches. Large simulations are much too slow for interaction, and we would concentrate on simulations with about 10^5 spatial grid points. Even this size of simulation would progress too slowly to be directly visually interesting, so the visualization should be asynchronous with the model. The sys-

tem will maintain an accumulating model output data set on the supercomputer, and let the user move around in the time step sequence, rather than being in lock step with the model. For numeric reasons, the model's time steps are generally much shorter than the time during which the storm makes visually interesting changes. Thus a storm simulation may calculate time steps for every two seconds of storm time but only store a time step in the accumulating data set for every minute of storm time. The user will visualize from the accumulating data set, using the same distributed software system that was described for visualizing the hurricane above. When the user wants to change the simulation, the system will enter a new operating mode allowing the initial conditions and model parameters to be changed and the model to be restarted at an earlier time step. This iteration will continue until the simulation behaves to the satisfaction of the scientist.

Almost all of the network traffic generated by this application is for visualiztion, and will be very similar to the traffic described above in Section 4.

6. Research issues for distributed interactive applications

The development of VIS-5D has included numerous experiments with the best ways to use the computing resources of the workstation to give the user fast and flexible interactions with large data sets, and how to control those interactions and resources through the user interface. The computing resources of the distributed version of VIS-SD are much more complex, involving two processors of unequal capabilities (workstation and supercomputer) and a communications network between them. Furthermore, the supercomputer and the communications network must usually be shared with other users, unlike the workstation.

In order to explore these issues, the simple operation sequence described above in Section 3 will be elaborated to provide more loosely coupled interactions between computational resources and to provide a more asynchronous user interface. Intermediate data structures, such as the subsectored grids on the supercomputer and the polygon lists on the workstation, can be cached according to various strategies, so that they need not be regenerated every time they are needed. The best caching strategies will depend on the patterns of user interactions and availability of computing resources, and will be determined experimentally.

Even where computing and communications rates are high there may be relatively long latencies. These can be addressed with a pipelined application design, so that the computations for several visual frames are simultaneously at different stages of the operation sequence. It may also be useful for the application to try to anticipate what the user will want to see.

Perhaps most important, we will structure different interactions to be done asynchronously, so that long computing and communications delays for one interaction do not block other interactions. For example, three-dimensional rotation which is implemented locally in the workstation should not be affected by delays in the communication network or the supercomputer. More complex examples arise when the application can cache intermediate data in the workstation.

Finally, the application should be able to provide useful service under a wide range of availablity of the shared resources. These issues will require numerous experiments with the design of the application and its user interface.

7. Conclusion

Understanding very large earth science data sets requires interactive visualization. Current high-performance workstations provide this for data sets of up to 5×10^7 points. High-bandwidth networking between supercomputer centers and scientists' workstations will make it possible to extend interactive visualization to data sets containing up to 10^{10} points for scientists in their own institutions. The network traffic for this application consists of very long messages at regular time intervals between a tenth and a half of a second. The total bandwidth is around 10^8 bits per second, depending on the speed of the computers and the use of compression. This application can also be adapted to interactive development of small numerical weather models.

Acknowledgements

We wish to thank Marie-Francoise Voidrot-Martinez, Dave Kamins, Jeff Vroom, Greg Wilson

and James Dodge for their help and support developing the VIS-5D system. We would also like to thank Francis Bretherton, Larry Landweber, Murray Thompson and Robert Kahn for their encouragement of the networking application. This work is being supported by the NASA Marshall Space Flight Center and the Gigabit Testbed Project being managed by the Corporation for National Research Initiatives.

References

[1] W. Hibbard and D. Santek, Visualizing large data sets in the earth sciences, *IEEE Comput.* **22** (8) (1989) 53–57.

[2] W. Hibbard and D. Santek, Interactive earth science visualization, *SIGGRAPH Video Rev.*, (43) (1989).

[3] G.J. Tripoli, A nonhydrostatic mesoscale model designed to simulate scale interaction, *Monthly Weather Rev.*, accepted for publication.

NIS

The VIS-5D System for Easy Interactive Visualization

Bill Hibbard and Dave Santek

Space Science and Engineering Center
University of Wisconsin - Madison

Abstract

The VIS-5D system provides highly interactive visual access to 5-dimensional data sets containing up to 50 million data points. The user has easy and intuitive control over animated 3-dimensional depictions of multiple interacting physical variables. VIS-5D is runs on the Stardent ST-1000 and ST-2000 workstations and is available as freeware from the Space Science and Engineering Center.

The VIS-5D System

We wrote the VIS-5D software system to help earth scientists understand their large and complex data sets. VIS-5D runs on the Stardent ST-1000 and ST-2000 workstations and generates animated 3-dimensional graphics from gridded data sets in real time. It provides a widget-based user interface and fast visual response which allows scientists to interactively explore their data sets. VIS-5D generates literal and intuitive depictions of data, has user controls which are data oriented rather than graphics oriented, and provides the WYSIWYG (what-you-see-is-what-you-get) response familiar to users of word processors and spread sheets. The result is a system which is easy for scientists to use, so that they can become the producers and directors of their own animations. VIS-5D can be applied to any data set in the McIDAS grid file format and containing up to 50 million grid points. Data sets containing hundreds of millions of grid points can be resampled to this 50 million point limit for interactive visualization.

We were motivated to write VIS-5D by our experiences using our 4-D McIDAS system running on IBM mainframe computers [1]. The 4-D McIDAS system generates 3-dimensional images in about 30 seconds each, with another 30 seconds each to load the images into a workstation for animation. We used this system to produce animated visualizations for many earth scientists. They were constantly wanting to change the animations and frustrated by the turnaround time. They found the video tapes we produced useful for public presentations and for teaching, but not for their own insight into their data sets, which they continued to get from the 2-dimensional graphics systems which they could use directly. Thus we wrote the highly interactive VIS-5D system which makes 3-dimensional graphics easy for scientists to use directly [2].

The VIS-5D system is available from the University of Wisconsin Space Science and Engineering Center as freeware. In addition to the visualization software, it includes tools for managing and analyzing large gridded data sets, a skeleton program for converting external data to the McIDAS grid file formats, documentation on how to use the software, and sample data sets to practice using the software.

Five-dimensional Data Sets

VIS-5D works with data in the form of a 5-dimensional rectangular grid of points. In a FORTRAN or C program these data sets could be declared as arrays with five dimensions. Three of the dimensions are spatial, one is time, and one is used to enumerate multiple physical variables. Thus these data sets sample a spatial volume at a regular lattice of points, sample dynamics at multiple steps over a time interval, and include multiple interacting physical variables.

Although a 5-dimensional grid may seem like a specialized format, it is the usual format for output from atmospheric simulations. It is also a common output

PRECEDING PAGE BLANK NOT FILMED

CH2913-2/90/0000/0028/$01.00 – 1990 IEEE

format for oceanography and hydrology simulations, and for some remote sensing instruments like radars which can scan quickly enough to produce time varying volumetric images. The most general setting for the earth's physical systems is multiple variables over three spatial dimensions plus time. Our 5-dimensional rectangles are just this setting, subject to discrete and uniform sampling of space and time. Thus the data format for VIS-5D is actually widely applicable to earth science data. For meteorological data the spatial dimensions are often latitude, longitude and altitude and the variables might be temperature, pressure, moisture and three wind vector components. For oceanography data the spatial dimensions are latitude, longitude and depth and the variables might be temperature, salinity, density and three ocean current vector components. For hydrology data the variables might be proportions for different rock and soil types, and three ground water flow vector components.

The VIS-5D system provides a high degree of interaction by storing the entire data set in the main memory of the workstation. Because of its compressed formats, this can be up to 50 million grid points. For example, these 50 million points can be factored as 50 latitudes by 50 longitudes by 20 altitudes by 100 time steps by 10 different physical variables. VIS-5D includes functions for managing much larger disk based data sets, and for resampling them down to smaller extents or to lower resolution in order to fit the size limit for interactive visualization. A factor of 2 reduction in resolution in time and space yields a 16 times reduction in data volume, which would allow a simulation data set of 800 million points to be reduced to the 50 million point limit.

The VIS-5D system supports a data format for trajectory paths, which are used to represent wind, ocean currents, ground water flow, and other motion fields, and a global topographical map data set, which is useful for large scale earth based data sets. The system includes a program for calculating trajectories from gridded motion vector fields, and management functions for listing, copying, merging and deleting data sets.

What Scientists Need

We accumulated considerable experience producing visualizations with scientists using our 4-D McIDAS system. Because 4D McIDAS requires an hour or more to produce each new animation sequence, this experience gave us an understanding of which types of changes to an animation sequence earth scientists really care about. These are:

A. change the viewpoint in three dimensions.

B. change the combination of simultaneously depicted variables.

C. change the depiction of a variable. For an iso-level contour surface this is a change to the defining value. For contour lines on a surface this is a change to the position of the surface and the density of the contour lines. For trajectory lines, this is a change to the density of the lines or placing trajectories through specific points.

D. change the time dynamics. This includes the choice of whether to enable time stepping, whether to step forward or backward (useful for tracing effects back to their causes), and how fast to step.

E. change the spatial extents of the depicted region.

F. calculate new variables from existing variables, including arithmetic, differential and integral operators.

G. make objects semi-transparent.

H. avoid depicting different variables using the same color.

It is worth noting that the types of controls the scientists care about relate to data rather than graphics. The value system of scientists is very different from

the value system of film and television producers. Scientists need to clearly perceive 3-dimensional geometry and time dynamics, and thus require some minimum standards of graphics quality. However, the benefits of advanced photorealistic techniques are often outweighed by the negative impact of their computational difficulty on system response time to user changes. Of course, many of the controls scientists care about are also computationally difficult, and this results in compromises in the user interface.

The VIS-5D User Interface

The goal of VIS-5D is to provide the scientist with an easy user interface for controlling the display, and fast visual response to changes. The workstation should feel like a steerable window which the scientist "flys" through a huge data set, hunting for interesting information hiding in the mass of data.

VIS-5D is able to present a simple and intuitive interface to the scientist because it deals specifically with the 5-dimensional rectangles of data produced by environmental simulations, because it generates very literal data depictions, and because it concentrates on the data oriented choices which scientists want.

The figure below shows a scene generated by VIS-5D which is part of a video depicting cold fronts moving across the North Atlantic. The data are taken from a forecast for February 4, 1988 by the European Center for Medium-range Weather Forecasts. The scene shows a topographical map, a transparent specific humidity contour surface at 6.38 grams per kilogram, and pressure contour lines at an altitude of 0.89 kilometers with a spacing of 5 millibars. The small clock hand in the upper left corner of the 3-D window shows where the scene is within the data set's time span. The highlighted widget buttons on the left show that the map, Q (specific humidity) and P (pressure) are enabled for display. The slider widgets show the level of the Q contour surface, the altitude and

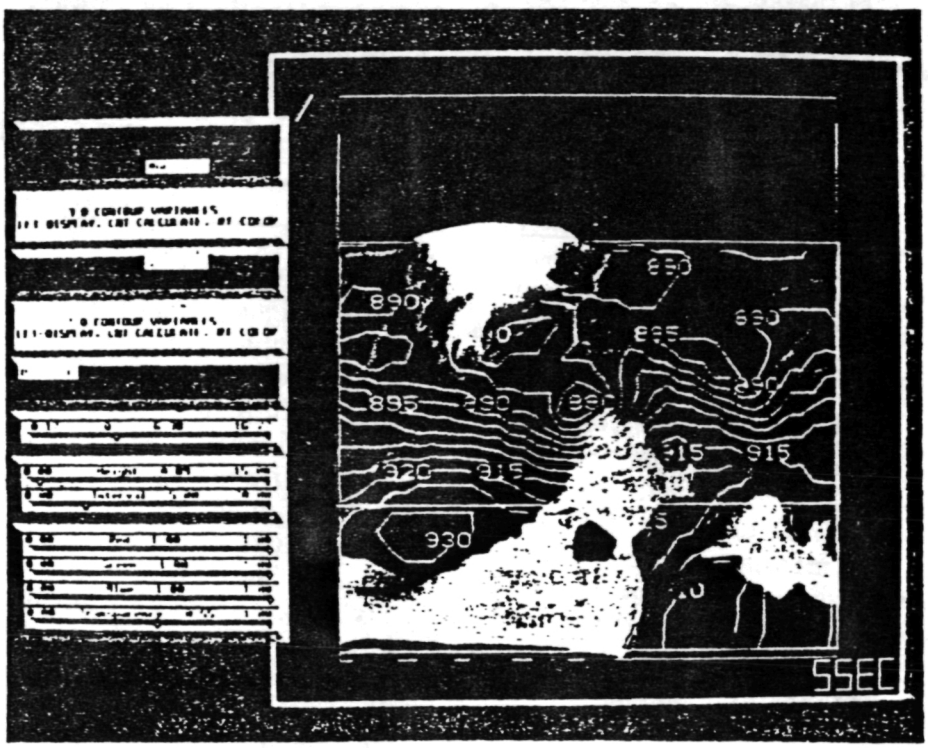

Figure 1 *(Color Plate 14, page 462)*

spacing of the P contour lines, and the color (white) and transparency (0.55) of the Q contour surface.

The three spatial dimensions of the data set rectangle are depicted with a single 3-dimensional box, and all the graphical elements depicting the data set are drawn in this common frame. The 3-D box contains either a static depiction of a single time step, or an animated depiction sequencing through the data set's time steps. The graphical elements of the depiction may include a topographical map, trajectory lines, and iso-level contour surfaces and contour lines for the data set's physical variables, all drawn in the common geometry of the 3-D box. The user can rotate, zoom and pan the 3-D box, control the time stepping, and independently enable or disable the depicted graphical elements. This provides a very literal representation of the 5-dimensional rectangle; the spatial dimensions are mapped into the 3-D box, the time dimension can be animated, and the physical variables can be viewed in arbitrary combinations.

The intuitive feel of the user interface is further enhanced by the fast response of VIS-5D to user controls. The 3-D box rotates, the time steps, and the depictions of variables appear and disappear, all within a fraction of a second of the appropriate mouse movement or button click. This immediate visual feedback is a critical element of a word processor's or spread sheet's user interface, and it is even more crucial for scientific visualization systems.

The McIDAS grid file format includes information specifying the time and location of each grid point, and the names of the physical variables in the data set. VIS-5D uses this information to generate a set of graphical widgets appropriate to the data set, and to automate the management of the components of the data set. For example, VIS-5D creates widgets for each physical variable, labelled with the names taken from the grid file, which are used to independently enable and disable graphical depictions of the variables. It also creates sliders for each variable used to change the values of their iso-level contour surfaces, and to change the altitude and density of their contour lines. A change to a slider value is applied to the appropriate variable for all the data set's time steps.

Changes to defining levels of contour surfaces and contour lines require a compromise in the user interface, because of the computational difficulty in computing new polygons and vectors for their graphical depictions. Ideally, the graphics would change as the user moved the slider, and VIS-5D does achieve this for changing altitude of contour lines on some data sets. However, new contour surfaces and complex contour line sets may require a couple of seconds for computation. When time dynamics are static, the new surface replaces the old surface as soon as it is computed. When time dynamics are animating, the new surfaces appear asynchronously with the animation sequence, gradually replacing the surfaces for all the data set's time steps.

VIS-5D provides pop up slider widgets which allow the user to change the color of contour surfaces and lines and the transparency of surfaces. When a data set includes ten different variables, each depicted by both surfaces and lines, it is hard to avoid multiple graphical elements with similar colors. With the color widgets the scientist can adjust the colors as variables are viewed in different combinations. The widget buttons used to enable graphical elements for display are highlighted with the color of the corresponding surface or lines, to help the scientist identify which graphics depict which variables.

We have avoided other graphical choices in our user interface. For example, there is a single light source which is always placed pointing along the view axis (actually there is a second light source pointing the other direction on the same axis to accommodate either sign of surface normals). The surfaces are drawn

according to a Gouraud shading model with fixed properties. In our experience these choices are less interesting to scientists and they tend to clutter up the user interface. Interactive rotation is a very powerful way to understand 3-D geometry, and these other controls offer only marginal improvement. Surface properties like specular highlights may actually be counterproductive by slowing the response time to interactive rotation. Surface property techniques like texture mapping can be useful when they are used to add data content to the display, although we have not yet included texture mapping into the VIS-5D system.

The Structure of VIS-5D

In order to maximize the size of the data set for visualization, VIS-5D uses a compressed format for the raw gridded data and the large polygon and vector lists used to represent contour surfaces and lines. The natural format of these data is 4 byte floating point, but they can be quickly compressed by a linear mapping into 1 or 2 byte integers, depending on the needed resolution. This compression allows a data set of 50 million grid points plus its associated polygon and vector lists to fit in 128MB of workstation memory.

The principal data structures of the VIS-5D visualization program include:

A. a 5-dimensional array of bytes, containing a compressed version of the raw gridded data. This array is organized into a series of 3-D spatial arrays indexed by time-step and variable. Each variable has a separate linear mapping for compression from its range of values to 1 byte integers.

B. a linear array which is dynamically allocated for the polygon and vector lists used to represent contour surfaces and contour lines. There is a polygon list and a vector list for each combination of time-step and variable (some lists may be empty), and an index by time-step and variable into the linear array. Vertex components are compressed by a linear mapping from the box extents to 2 byte integers, and normal components are compressed by a linear mapping from the interval (-1.0, 1.0) to 1 byte integers.

C. vector lists for trajectory lines. Each trajectory is stored as a single poly-vector with an index by time-step into the poly-vector.

D. a polygon mesh for the topographical map and vector lists for the map boundary lines.

E. a queue containing time-step and variable indices identifying 3-D grids for which contour surface polygon lists or contour line vector lists need to be computed.

F. arrays of values of iso-level contour surfaces and altitudes and densities of contour lines, indexed by variable.

G. colors and transparencies for contour surfaces and colors for contour lines, indexed by variable.

H. state information for the display, including the current time-step, whether animation is enabled, whether the map is enabled, whether the trajectories are enabled, and the transformation matrix for the 3-D to 2-D projection. This state information also includes arrays indicating whether contour surfaces and contour lines are enabled, indexed by variable.

I. ordered lists of variables recording which contour surfaces and lines have been most recently enabled for display.

J. intermediate structures used for computing contour surfaces and contour lines from 3-D grids.

VIS-5D runs under Stellix (UNIX System V with Berkeley extensions) and X Windows Version 11 Release 3. The top level pseudo-code for the visualization program is:

```
read the data set into the compressed 5-D byte array
create the linear array for polygon and vector lists
initialize the polygon and vector lists to empty
initialize the contour surface and line queue to empty
if the user specified a trajectory data set
  read the trajectory file
  build the trajectory vector lists and time-step indices
end if
if the user specified a map
  read the topography and map outline files
  resample these map data to a reasonable resolution
  build the map polygon mesh and vector lists
end if
set defaults for colors of contour surfaces and lines
set defaults for iso-levels of contour surfaces
set defaults for altitudes and densities of contour lines
create a window for the 3-D display
create widgets according to data set contents
initialize the display to the first time-step with no graphics
        enabled and nominal 3-D to 2-D projection
fork into 4 parallel threads
  thread 1
    do forever
      clear the display
      render a rectangular box
      render the map if enabled
      render the trajectories if enabled
      for each variable
        if the contour lines are enabled
          decompress the line vector list for the current time-step
          render vector list according to color for the variable
        end if
      end for
      for each variable (in order of decreasing opacity)
        if the contour surface is enabled
          decompress the surface polygon list for the current time-step
          render polygon list according to color and transparency
                for the variable
        end if
      end for
      check for X events and widget callbacks
        adjust the projection matrix according to mouse moves
        toggle map enable/disable if requested
        toggle trajectory enable/disable if requested
        toggle contour surface and line enable/disables if requested and re-order lists of
              variables recording which have been most recently displayed
        toggle time animation enable/disable if requested
        if time animation is disabled
          increment, decrement or reset time-step if requested
        end if
        change colors and transparencies if requested
        change contour surface levels if requested
        change contour line altitudes and densities if requested
```

```
      if a contour surface or line recompute is requested
       for each time-step
        if time animation is disabled and time-step=current
         add the selected variable and time-step to the head of the queue
        else
         add the selected variable and time-step to the tail of the queue
        end if
       end for
      end if
      exit visualization program if selected
     end check for X events and widget callbacks
     if time animation is enabled
      increment the time-step
     end if
    end do forever
  end thread
  threads 2, 3 and 4 (they are identical)
    do forever
     if the queue contains any contour line requests
      remove the first request for contour lines
      decompress the 3-D grid for time-step and variable
      compute contour lines at altitude and density for variable
      compress vector list for lines
      deallocate previous vector list for time-step and variable
      if there is not adequate free space in the linear array
       delete the least recently used vector and polygon lists
              until there is adequate space
      end if
      allocate space in linear array and insert vector list
      add index to vector list for time-step and variable
     else if the queue contains any contour surface requests
      remove the first surface request
      decompress the 3-D grid for time-step and variable
      compute contour surface at iso-level value for variable
      compress polygon list for surface
      deallocate previous polygon list for time-step and variable
      if there is not adequate free space in the linear array
       delete the least recently used vector and polygon lists
              until there is adequate space
      end if
      allocate space in linear array and insert polygon list
      add index to surface list for time-step and variable
     end if
    end do forever
  end thread
end fork
```

The ST-1000 and ST-2000 execute four instruction streams in parallel, so VIS-5D forks into four threads to take advantage of this parallelism. The X server is also a heavy computing load while VIS-5D is running and increases parallelism. Because fast response is important to VIS-5D, data should be accessed from main memory rather than disk. VIS-5D allocates a single large array from which to allocate polygon and vector lists in order to control the total use of main memory. This way it can

avoid paging delays which would occur if allocated memory became significantly larger than physical memory.

The parallel threads implement critical sections where simultaneous access to common data structures could cause interference. This true for insertion and deletion in the queues, the allocation and deallocation of space in the linear array, and reading and updating the polygon and vector lists and their associated index.

VIS-5D uses Stardent's XFDI library of 3-D extensions to X for rendering, using a Z-buffer and RGB true color. We also use a modified version of Stardent's LUI widget library, which is part of their Application Visualization System (AVS).

Future Developments

We have received numerous suggestions for additional functions for VIS-5D from scientists, as well as shortcomings which we recognize. Some of these are:

A. include contour lines drawn on vertical planes which can be arbitrarily positioned. This is currently being developed.

B. dynamically calculate trajectories through space-time points specified with a 3-D cursor. This is currently being developed.

C. represent planes through 3-D grids with pseudocolored images in addition to the current contour lines. This would be useful for radar data which are less smooth than model data.

D. texture map satellite images onto surfaces in the 3-D box.

E. render 3-D grids as transparent fogs, often referred to as volume images. This may be difficult to do with fast enough response for interactive rotation.

F. provide interactive analysis operations on the 5-D grid of data, including arithmetic, differential and integral operations. This is an open-ended area of development, often dependent on the particular source of the data set.

G. increase the size of the data sets which can be interactively visualized. This applies to the total 5-D rectangle and the number of grid points in the spatial 3-D box. Assuming the current level of interactivity, this depends on faster workstations, larger memories, and disks fast enough to support interactive access.

VIS-5D is aimed at 5-D data sets similar to those produced by weather models. We are also interested in developing systems for interactively visualizing and analyzing large image data sets. The same workstation technology which makes VIS-5D possible can also be exploited for radical new ways of processing image data, although the overall structure of such an application may be quite different from VIS-5D.

Acknowledgments

We wish to thank Dave Kamins and Jeff Vroom of Stardent Computer, Inc., and Marie-Francoise Voidrot-Martinez of the French Meteorology Office for their help. We also wish to thank the many scientists we have worked with and NASA Marshall Space Flight Center for their support (NAS8-36292).

References

1. Hibbard, W., and D. Santek, 1989: Visualizing large data sets in the earth sciences. IEEE Computer 22(8), 53-57.

2. Tripoli, G., W. Hibbard, and D. Santek, 1989: Four-dimensional interactive analysis: a tool for the efficient understanding of large data sets. Preprints, 12th Conference on weather analysis and forecasting. Monterey, American Meteorological Society, J10-J12.

N15

The VIS-AD Data Model: Integrating Metadata and Polymorphic Display with a Scientific Programming Language

William L. Hibbard[1&2], Charles R. Dyer[2] and Brian E. Paul[1]

[1]Space Science and Engineering Center
[2]Computer Sciences Department
University of Wisconsin - Madison
whibbard@macc.wisc.edu

Abstract. The VIS-AD data model integrates metadata about the precision of values, including missing data indicators and the way that arrays sample continuous functions, with the data objects of a scientific programming language. The data objects of this data model form a lattice, ordered by the precision with which they approximate mathematical objects. We define a similar lattice of displays and study visualization processes as functions from data lattices to display lattices. Such functions can be applied to visualize data objects of all data types and are thus polymorphic.

1. Introduction

Computers have become essential tools to scientists. Scientists formulate models of natural phenomena using mathematics, but in order to simulate complex events they must automate their models as computer algorithms. Similarly, scientists analyze their observations of nature in terms of mathematical models, but the volumes of observed data dictate that these analyses be automated as computer algorithms. Unlike hand computations, automated computations are invisible, and their sheer volume makes them difficult to comprehend. Thus scientists need tools to make their computations visible, and this has motivated active development of scientific visualization systems. Explicitly or implicitly, these systems are based on:

1. A data model - how scientific data are defined and organized.
2. A computational model - how computations are expressed and executed.
3. A display model - how data and information are communicated to a the user.
4. A user model - the tasks and capabilities (e.g., perceptual) of users.
5. A hardware model - characteristics of equipment used to store, compute with, and display data.

Robertson et. al. [11] describe the need for a foundation for visualization based on such formal models. The user and hardware models help define the context and requirements for a system design, whereas the data, computational and display models are actually high level components of a system design. Because

PRECEDING PAGE BLANK NOT FILMED

scientists explore into unknown areas of nature, they need models of data, computation, and display that can adapt to change.

2. Data Model Issues

2.1 Levels of Data Models

A data model defines and organizes a set of data objects. Data models can be defined at various levels of functionality [16]. Data models can describe:

1. The physical layout and implementation of data objects. At the lowest level, a data model may describe the physical layout of bits in data objects. It is widely acknowledged that this level should be hidden from users, and even hidden from systems developers as much as possible. At a slightly higher level, a data model may describe the data objects of a visualization system in terms of the data objects of the programming language(s) used to implement the system.
2. The logical structure of data. This level describes the mathematical and logical properties of primitive data values, how complex data objects are composed from simpler data objects, and relations between data objects.
3. The behavior of data in computational processes. This is a pure object-oriented view of data. The internal structure of data objects is invisible, and all that is specified is the behavior of functions operating on data objects.

While purely behavioral models of scientific data are possible, it is rare to see a behavioral data model that does not refer to the logical structural of data. That is, the behaviors of functions operating on objects are usually explained in terms like "returns a component object" or "follows a reference to another object." In particular, most data models that are described as "object oriented" are object oriented implementations of structural data models. In these cases, the internal structure of objects is hidden in the sense of programming language scope rules, but is not hidden in the user's understanding of object behavior. The idea of defining complex things in terms of simpler things is extremely natural and convenient, so it is not surprising that most data models are essentially structural. Furthermore, structural data models permit automated analysis of data syntax (e.g., for query optimization), but it is difficult to apply similar analyses to purely functional specifications of data.

2.2 Structural Data Models

The physical and implementation levels address issues that should not be visible to scientists using a system, and purely behavioral data models are rare. Thus we focus on the structural level. At this level a data model needs to address the following issues:

1. The types of primitive data values occurring in data objects. A primitive type defines a set of primitive values. It may also define an order relation, basic operations (e.g., addition, negation, string concatenation), and a topology (e.g., the discrete topology of integers, the continuous topology of real numbers) on the set of values.
2. The ways that primitive values are aggregated into data objects. These may be simple tuples of values, they may be functional relations between variables, or they may be complex networks of values.
3. Metadata about the relation between data and the things that they represent. For example, given a meteorological temperature, metadata includes the fact that it is a temperature, its scale (e.g., Fahrenheit, Kelvin), the location of the temperature and whether it is a point or volume sample, the time of the temperature, an estimate of its accuracy, how it was produced (e.g. by a simulation, by direct observation, or deduced from a satellite radiance), and whether the value is missing (e.g., in case of a sensor failure).

A structural data model defines behavior rather than implementation, but does so in terms of an underlying structure. That is, primitive types describe the operations that can be applied to primitive objects, but do so under the assumption that the state of a primitive object is a simple mathematical value. Similarly, aggregate types describe operations that return objects as functions of other objects, but do so in terms of hierarchical and network relations between objects. A purely behavioral model would not place such constraints on operations on objects.

2.3 Extensive Versus Intensive Models for Types of Data Aggregates

The way that a structural data model defines types of data aggregates is an important issue in the context of data visualization. Many visualization systems define data models that are essentially finite enumerations of those aggregate types for which the systems have implemented display functions. For example, a visualization system's data model may include images, 3-D scalar and vector fields, vector and polygon lists, and color maps. On the other hand, scientists writing programs require flexibility in defining aggregate types. Thus programming languages usually define data models in terms a set of techniques that let users define their own (potentially infinite) sets of aggregate types. That is, users are given language features like tuples (i.e., structures in C), arrays and pointers for building their own structures. Visualization systems stress aspects of their data models related to display models, whereas programming languages stress aspects of their data models related to computational models.

In set theory, a set may be defined *extensively* as a list of members, or defined *intensively* by a logical condition for membership. We borrow these terms, saying that an extensive data model is one that defines a finite enumeration of aggregate types, and saying that an intensive data model is one that defines a set of techniques for building a potentially infinite set of aggregate types. Systems designed for particular applications, including many scientific visualization

systems, tend to define extensive data models, while programming languages tend to define intensive data models.

Scientists need data models that can support general computational models and can also support general display models for all data objects. Object oriented techniques provide one approach to this need. Each aggregate type in an extensive data model can be defined as a different object class. Inheritance between classes simplifies the task of designing new types of aggregates, and polymorphism allows analysis and display functions to be applied uniformly to many different aggregate types. However, this approach still requires users to explicitly define new object classes and their display functions. An approach based on intensive data models may be easier for scientists to use.

2.4 Models for Metadata

Programming languages and visualization systems differ in their level of support for metadata. While programming languages offer users the flexibility to build their own logic for managing metadata, they have no intrinsic semantics for the relation between data and its metadata. For example, scientists may adopt the convention that -999 represents a missing value, but since the programming languages that they use do not implement any special semantics for this value, their programs must include explicit tests for this value. On the other hand, many scientific visualization systems do intelligently manage the relation between data and its metadata. For example, some systems implement missing data codes, some systems manage information about the spatial locations of data (sometimes called data navigation), and some systems manage information needed to normalize observations to common scales (sometimes called data calibration).

3. Data Lattices

Mathematical models define infinite precision real numbers and functions with infinite domains, whereas computer data objects contain finite amounts of information and must therefore be approximations to the mathematical objects that they represent. For example, a 32-bit floating point number represents real numbers by a set of roughly 2^32 different values in the range between -10^38 and +10^38, plus a few special codes for illegal and out-of-range values. Since most real numbers are not members of this set of 2^32 values, they can only be approximately represented by floating point numbers. As another example, a satellite image is a finite sampling of a continuous radiance field over the Earth. The image contains a finite number of pixels, and pixels sample radiance values in finite numbers of bits (8-bit values are common). Thus the satellite image can only approximate the continuous radiance field. Satellites and other sensor systems are fallible, so scientists usually define missing data codes to represent values where sensors failed. These missing data codes may be interpreted as approximations that contain no information about the mathematical values that they represent.

We can define an order relation between data objects based on the fact that some are better approximations than others. That is, if x and y are data objects, then we define $x \leq y$ to mean that y is consistent with x, and that y provides more precise information than x does. We illustrate this order relation using closed real intervals as approximations to real numbers. If w is a real number, and if $[a, b]$ is the real interval of numbers between a and b, then $[a, b]$ is an approximation to w if w belongs to the interval $[a, b]$ (i.e., if $a \leq w \leq b$). Given two intervals $[a, b]$ and $[c, d]$, we say that $[a, b] \leq [c, d]$ if $[c, d] \subseteq [a, b]$. This is because the smaller interval provides more precise information about a value than the containing interval does. Letting the symbol $\perp$ represent a missing data code, then $\perp$ provides less precise information about a real value than any interval, so we can say that $\perp <$ $[a, b]$ for any interval $[a, b]$. Figure 1 shows a few closed real intervals and the order relations among those intervals.

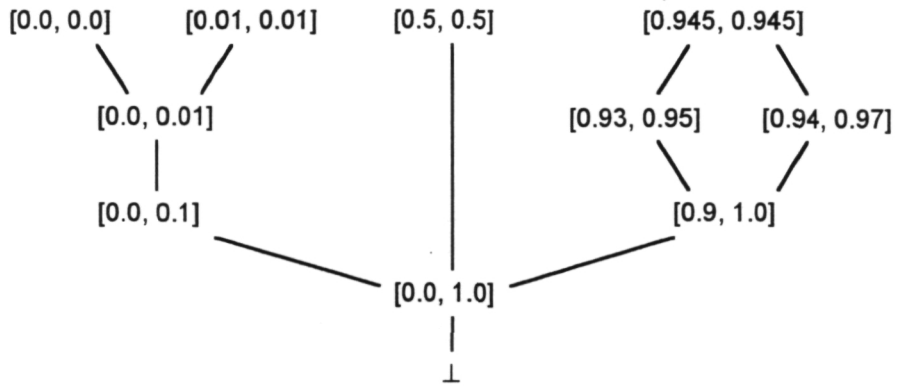

Figure 1. Closed real intervals are used as approximate representations of real numbers, ordered by the inverse of containment (i.e., containing intervals are "less than" contained intervals). We also include a least element $\perp$ that corresponds to a missing data indicator. This figure shows a few intervals, plus the order relations among those intervals. The intervals in the top row are all maximal, since they contain no smaller interval.

We interpret arrays as finite samplings of functions. For example, a function of a real variable may be represented by a set of 2-tuples that are (domain, range) pairs. The set {([1.1, 1.6], [3.1, 3.4]), ([3.6, 4.1], [5.0, 5.2]), ([6.1, 6.4], [6.2, 6.5])} contains three samples of a function. The domain value of a sample lies in the first interval of a pair and the range values lies in the second interval of a pair, as illustrated in Fig. 2. Adding more samples, or increasing the precision of samples, will create a more precise approximation to the function. Figure 3 shows the order relations between a few array data objects.

In general we can order arrays to reflect how precisely they approximate functions. If x and y are two array data objects that are both finite samplings of a function, and if, for each sample of x, there is a collection of samples of y that improve the resolution of the function's domain and range over the sample of x, then $x \leq y$. Intuitively, y contains more information about the function than x does.

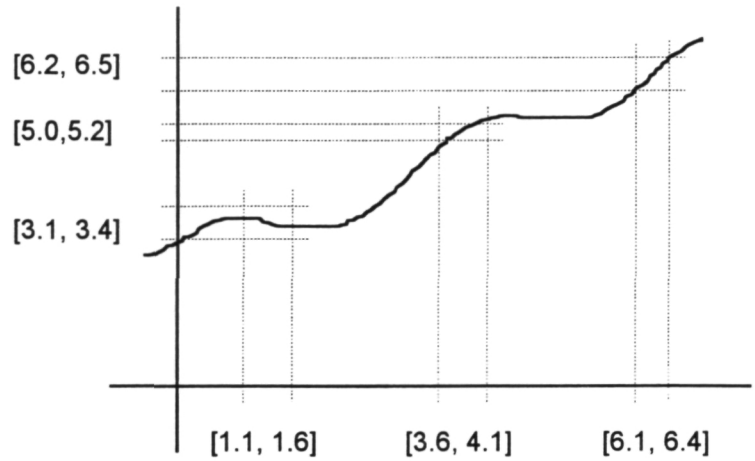

Figure 2. An approximate representation of a real function as a set of pairs of intervals.

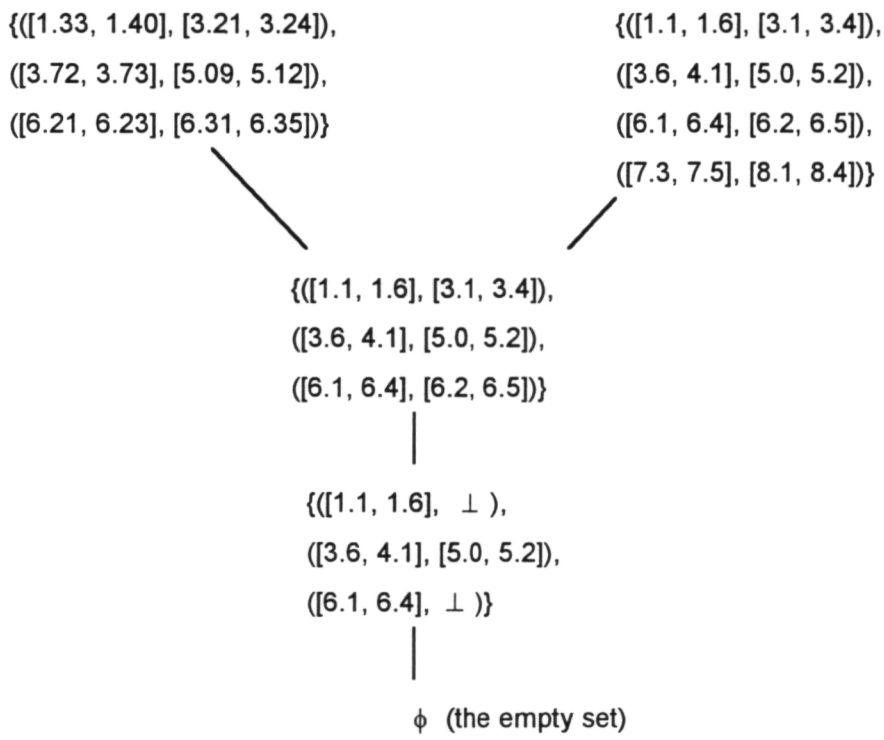

Figure 3. A few finite samplings of functions, and the order relations among them.

3.1 A Scientific Data Model

Now we will define a data model that is appropriate for scientific computations. We describe this data model in terms of the way it defines primitive values, how those values are aggregated into data objects, and metadata that describes the relation between data objects and the mathematical objects that they represent.

The data model defines two kinds of primitive values, appropriate for representing real numbers and integers. We call these two kinds of primitives *continuous scalars* and *discrete scalars*, reflecting the difference in topology between real numbers and integers. A continuous scalar takes the set of closed real intervals as values, ordered by the inverse of containment, as illustrated in Fig. 1. A discrete scalar takes any countable set as values, without any order relation between them (since no integer is more precise than any other). Figure 4 illustrates the order relations between values of a discrete scalar. Note that discrete scalars may represent text strings as well as integers. The value sets of continuous and discrete scalars always include a minimal value $\perp$ corresponding to missing data.

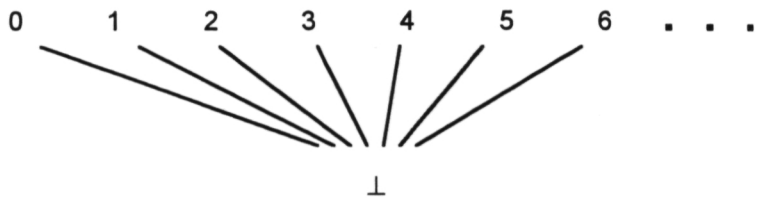

Figure 4. A discrete scalar is a countable (possibly finite) set of incomparable elements, plus a least element $\perp$.

Our data model defines a set T of data types as ways of aggregating primitive values into data objects. Rather than enumerating a list of data types in T, the data model starts with a finite set S of scalar types, representing the primitive variables of a mathematical model, and defines three rules by which data types in T can be defined. These rules are:

1. Any continuous or discrete scalar in S is a data type in T. A scientist using this data model typically defines one scalar type in S for each variable in his or her mathematical model.
2. If t_1, ..., t_n are types in T, then *struct*$\{t_1;...;t_n\}$ is a *tuple* type in T with *element* types t_i. Data objects of tuple types contain one data object of each of their element types.
3. If w is a scalar type in S and r is a type in T, then (*array* [w] *of* r) is an *array* type with *domain* type w and *range* type r. Data objects of array types are finite samplings of functions from the primitive variable represented by their domain type to the set of values represented by their range type. That is, they are sets of data objects of their range type, indexed by values of their domain type.

Each data type in T defines a set of data objects. Continuous and discrete scalars define sets of values as we have described previously. The set of objects of a tuple type is the cross product of the sets of objects of its element types. The set of objects of an array type is not quite the space of all functions from the value set of its domain type to the set of objects of its range type. Rather, it is the union of such function spaces, taken over all finite subsets of the domain's value set.

A tuple of data objects represent a tuple of mathematical objects, and the precision of the approximation depends on the precision of each element of the tuple. One tuple is more precise than another if each element is more precise. That is, $(x_1, ..., x_n) \leq (y_1, ..., y_n)$ if $x_i \leq y_i$ for each i. Figure 5 illustrates the order relations between a few tuples.

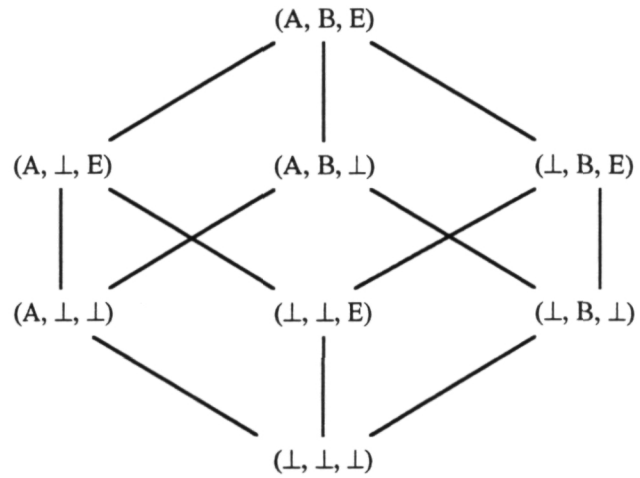

Figure 5. Defining an order relation on a cross product. Members of cross products are tuples. This figure shows a few elements in a cross product of three sets, plus the order relations among those elements. In a cross product, the least element is the tuple of least elements of the factor sets.

An array data object is a finite sampling of a function, and the precision of approximation depends on how precisely the function's domain is sampled and the precision of the array's range values. If an array is indexed by a continuous scalar, the interval values of the index indicate how precisely the function's domain is sampled, as illustrated in Figs. 2 and 3.

By building hierarchies of tuples and arrays, it is possible to define data types in T that represent virtually any mathematical model used in the physical sciences. For example, consider a set of data types appropriate for analyzing meteorological observations. The scalar types used to represent primitive variables for this analysis include:

temperature - thermometer reading (continuous)

```
dew_point       - wet bulb thermometer reading (continuous)
pressure        - barometer reading (continuous)
count           - frequency count of values in a histogram (discrete)
station_name    - name of observing station (discrete)
latitude        - latitude of observing station (continuous)
longitude       - longitude of observing station (continuous)
time            - time of observation (continuous)
```

The complex data types for this analysis include:

```
station_reading = struct{
                        .sta_temp = temperature;
                        .sta_dew = dew_point;
                        .sta_pres = pressure;
                }
station_series = (array [time] of station_reading)
station_set = (array [station_name] of
                struct{
                        .set_series = station_series;
                        .set_lat = latitude;
                        .set_lon = longitude;
                }
temperature_histogram   = (array [temperature] of count)
```

A data object of the *station_reading* type includes one value for each instrument at a weather observing station. A data object of type *station_series* contains a sequence of *station_reading* objects, that finitely sample continuous functions of meteorological fields over *time*. A data object of type *station_set* is an array that associates a time series of readings and *latitude* and *longitude* locations to each of a finite set of *station_names*. A data object of the *temperature_histogram* type contains frequency *counts* of intervals of *temperatures*. In this case, the interval values of the *temperature* represent the bins used for histogram calculation.

The lattice data model defines certain metadata about the relation between data objects and the mathematical objects that they represent, including:

1. Every primitive value in a data object is identified by the name of the primitive mathematical variable.
2. An array data object is a finite sampling of a mathematical function. The set of index values of the array specify how the array samples the function being represented.
3. The interval values of continuous scalars are approximations to real numbers in a mathematical model, and the sizes of intervals provide information about the accuracy of their approximations.

4. Any scalar data object may take the missing value (denoted by $\perp$) and this provides information about accuracy (i.e., the fact that the value has no accuracy).

3.2 Interpreting the Data Model as a Lattice

We view a data display process as a function from a set of data objects to a set of display objects. Our data model defines a different set of data objects for each different data type, suggesting that a different display function must be defined for each different data type. However, we can define a lattice U of data objects and natural embeddings of data objects of all data types into U. The lattice U provides us with a unified model for all of our scientific data objects, and enables us to define display functions that are applicable to all data types (i.e., these display functions are polymorphic). Our analysis of the properties of display functions will thus be independent of particular data types.

A *lattice* is an ordered set U in which every pair of elements x and y has a least upper bound $sup\{x, y\}$ [this is z such that $x \le z, y \le z$ and $\forall w \in U. (x \le w \ \& \ y \le w \Rightarrow z \le w)$] and a greatest lower bound $inf\{x, y\}$. A lattice U is *complete* if it contains the least upper bound $sup(A)$ and the greatest lower bound $inf(A)$ for any subset $A \subseteq U$.

We define a data lattice U whose members are sets of tuples. The primitive domains of this data lattice are defined by a finite set S of scalar types, and the tuple space is the cross product of the sets of values of the scalar types in S. Define I_s as the set of values of a scalar $s \in S$ and define $X = \mathbf{X}\{I_s \mid s \in S\}$ as the cross product of these scalar value sets. Members of our data lattice are subsets of X. Figures 1 and 4 illustrate the order relations on the scalar value sets I_s, and Fig. 5 illustrates the order relation on the set X of tuples.

Members of U are subsets of X. However, there is a problem with defining an order relation between subsets of X that is consistent with the order relation on X and is also consistent with set containment. For example, if $a, b \in X$ and $a < b$, we would expect that $\{a\} < \{b\}$. Thus we might define an order relation between subsets of X by:

$$\forall A, B \subseteq X. (A \le B \Leftrightarrow \forall a \in A. \exists b \in B. a \le b) \qquad (1)$$

However, given $a < b$, (1) implies that $\{b\} \le \{a, b\}$ and $\{a, b\} \le \{b\}$ are both true, which contradicts $\{b\} \ne \{a, b\}$. This problem can be resolved by restricting the lattice U to sets of tuples such every tuple is maximal in the set. That is, a set $A \subseteq X$ belongs to the lattice U if $a < b$ is not true for any pair $a, b \in A$. (Actually, the situation is a bit more complex - see [7] for the details.) The members of U are ordered by (1), as illustrated in Fig. 6, and form a complete lattice.

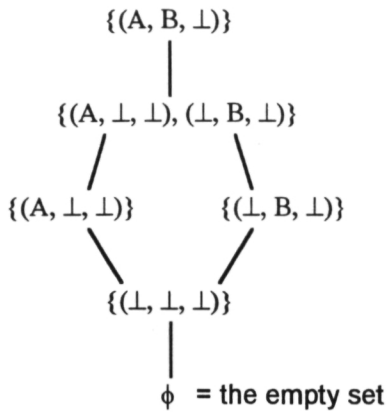

Figure 6. A few members of a data lattice U defined by three scalars, and the order relations between them.

To get an intuition of how data types are embedded in the lattices, consider a data lattice U defined from the three scalars *time*, *temperature* and *pressure*. Objects in the lattice U are sets of tuple of the form (*time*, *temperature*, *pressure*). We can define a tuple data type *struct*{*temperature*; *pressure*}. A data object of this type is a tuple of the form (*temperature*, *pressure*) and can be modeled as a set of tuples (actually, it is a set consisting of one tuple) in U with the form {(⊥, *temperature*, *pressure*)}. This embeds the tuple data type in the lattice U, and Fig. 7 illustrates this embedding.

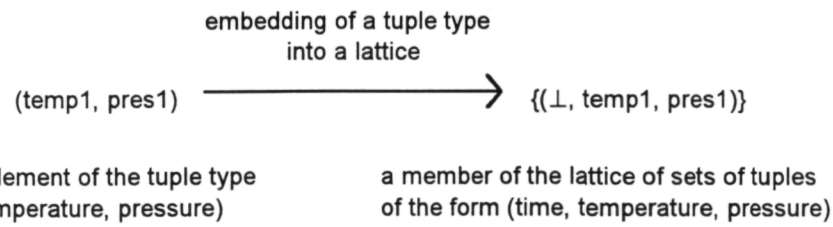

Figure 7. An embedding of a tuple type into a lattice of sets of tuples.

Similarly, we can embed array data types in the data lattice. For example, consider the same lattice U defined from the three scalars *time*, *temperature* and *pressure*, and consider an array data type (*array* [*time*] *of temperature*). A data object of this type consists of a set of pairs of (*time*, *temperature*). This array data object can be embedded in U as a set of tuples of the form (*time*, *temperature*, ⊥). Figure 8 illustrates this embedding. The basic ideas presented in Figs. 7 and 8 can be combined to embed complex data types, defined as hierarchies of tuples and arrays, in data lattices. The details are explained in [6] and [7].

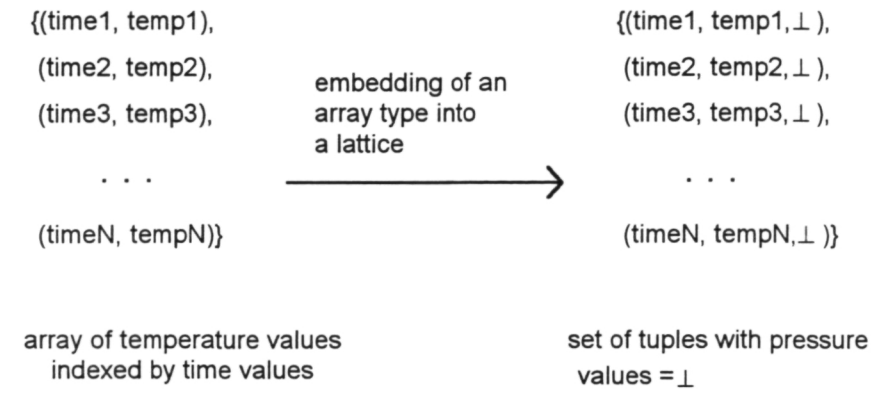

Figure 8. An embedding of an array type (a functional dependency between scalar types) into a lattice of sets of tuples.

If $x \in X$ is the embedding of a data object of a type $t \in T$, and if the scalar s does not occur in the definition of t, then the s values of all the tuples in x will be $\perp$. Also, in order to embed data objects in the data lattice U, we must restrict T to the set of data types t such that no scalar s occurs more than once in the definition of t. We note that, for each type in $t \in T$, the embedding of data objects of type t into U is an order embedding. This means that if a and b are objects of type t then $a \leq b$ if and only if $E_t(a) \leq E_t(b)$, where is E_t is the embedding of objects of type t.

Lattices and other kinds of ordered sets have played an important role in the denotational semantics of programming languages [2, 12, 13, 14, 15], and they can also play an important role in visualization.

3.3 Display Lattices

Our lattice structure can also be used to model displays. This is motivated by analogy with the display model of Bertin [1]. He defined a display as a set of graphical marks, and identified eight primitive variables of a graphical mark: two spatial coordinates of the mark in a graphical plane (he restricted his attention to static 2-D graphics), plus size, value, texture, color, orientation, and shape. Bertin defined diagrams, networks and maps as spatial aggregates of graphical marks. By defining a graphical mark as a tuple of its graphical primitive values, a display can be viewed as a set of tuples.

We define a finite set DS of display scalars that represent graphical primitives and we interpret a tuple of values of the display scalars as a graphical mark. Similar to the data lattice U, we define a display lattice V whose members are sets of tuples of values of display scalars.

We can define a display lattice for static 2-D displays using five continuous display scalars: two for image coordinates plus three for color components (e.g., red, green and blue). In this model, a display is a set of colored rectangles. The interval values of the image coordinate scalars in a tuple specify the size and location of the

rectangle on the screen, and the interval values of the color component scalars specify the range of colors used in the rectangle. This model can be extended to dynamic 3-D displays, by adding two more display scalars: one for a third image coordinate and another for indicating a graphical mark's location in an animation sequence. The three image coordinates then specify the locations and sizes of 3-D rectangles that must be projected onto a 2-D display screen (where multiple rectangles are projected to the same screen location, their colors must be combined according to some compositing algorithm). The values of the animation scalar are used to select tuples for display. At any instant during data display, an animation index takes an interval value, and only those tuples whose animation scalar intervals overlap this animation index value are displayed. By sequencing through values of the animation index, the display screen contents will change, providing a dynamic display. Figure 9 illustrates the role of the various display scalars in this display model.

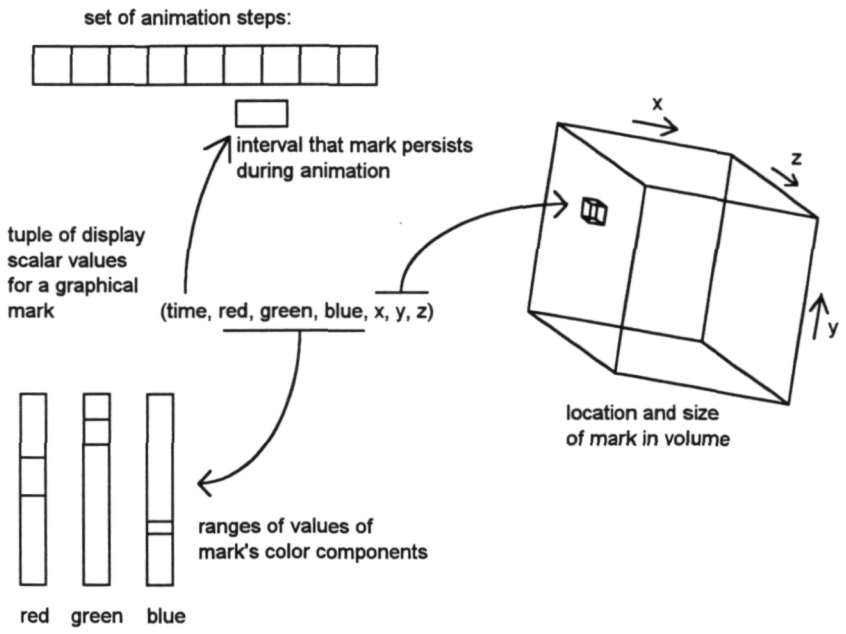

Figure 9. The roles of the display scalars in an animated 3-D display model.

Just like computer data objects, computer displays contain finite amounts of information. Pixels and voxels have limited resolution, colors are specified with limited precision, animation sequences consist of finite numbers of steps, etc. The lattice structure of V orders displays based on their information content.

Display models need not be limited to such primitive values as spatial coordinates, color components and animation indices. For example, consider a display model where a display consists of a set of graphical icons distributed at various locations in a display screen. This display model could be defined using

three display scalars: a horizontal screen coordinate, a vertical screen coordinate, and an icon identifier. Then a single value of the icon identifier display scalar would represent the potentially complex shape of a graphical icon. Or, a set of display scalars may form the parameters of a complex graphical shape. For example, 2-D ellipses may be used as graphical marks, parameterized by five display scalars for their center coordinates, orientations, and the lengths of their major and minor axes.

3.4 Data Display as a Mapping From a Data Lattice to a Display Lattice

We model a display process as a function $D:U \rightarrow V$ that generates a display in V from any data object in U. Rather than defining such functions constructively, in terms of algorithms for calculating a display $D(u)$ from a data object $u \in U$, we will define conditions on D and study the class of functions satisfying those conditions. For our conditions, we interpret Mackinlay's *expressiveness* conditions [8] in the lattice context. These conditions require that a display encode all the facts about a data object, and only those facts. As we show in [6] and [7], we can interpret these conditions as:

Condition 1. $\forall P \in MON(U \rightarrow \{\perp, 1\})$.
$\exists Q \in MON(\downarrow D(MAX(X)) \rightarrow \{\perp, 1\}).\ P = Q \circ D$

Condition 2. $\forall u \in U.\ D(u) \in \downarrow D(X)$ and $\forall Q \in MON(\downarrow D(MAX(X)) \rightarrow \{\perp, 1\})$.
$\exists P \in MON(U \rightarrow \{\perp, 1\}).\ Q = P \circ D^{-1}$

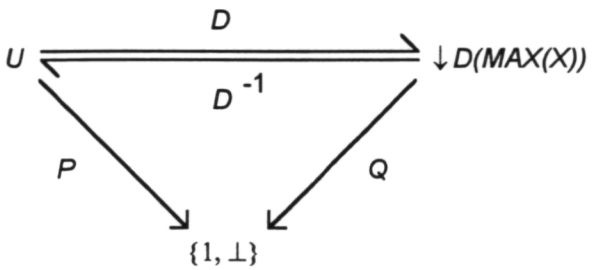

Figure 10. Expressiveness Conditions 1 and 2 interpreted as a commuting diagram. The conditions require that a display function D generate a one-to-one correspondence between the set of monotone functions P and the set of monotone functions Q, going both ways around the diagram.

Here $MON(U \rightarrow \{\perp, 1\})$ is the set of monotone functions from U to the set $\{\perp, 1\}$, $MAX(X)$ is the set of maximal tuples in X and thus the maximal element of U, and $\downarrow D(MAX(X))$ is the set of all displays in V less than the display of $MAX(X)$. A function P is monotone if $x \leq y$ implies $P(x) \leq P(y)$. We interpret facts about data objects as functions in $MON(U \rightarrow \{\perp, 1\})$ and we interpret fact about displays as functions in $MON(V \rightarrow \{\perp, 1\})$ (however, we limit this to displays less than the

display of the maximal data object). Condition 1 says that for every P there is a Q that makes the diagram in Fig. 10 commute, and Condition 2 says that for every Q there is a P that makes the diagram commute.

We say that a function $D{:}U{\rightarrow}V$ is a *display function* if it satisfies Conditions 1 and 2. In [7] we prove:

Proposition 1. $D{:}U{\rightarrow}V$ is a display function if and only if it is a lattice isomorphism from U onto $\downarrow D(MAX(X))$, which is a sub-lattice of V.

The definition of *display function*, and the proof of this proposition, do not refer to the construction of data and display lattices in terms of scalars (although that construction motivates some of the discussion). The set $MAX(X)$ plays a role in the definition of *display function* and in our proofs, but only as the maximal element of the lattice U. Since any complete lattice has a maximal element (i.e., the *sup* of all its elements), this result is true for any pair of complete lattices U and V.

In the special case that the lattice U and V are constructed from scalars and display scalars as described in Sects. 3.2 and 3.3, display functions can be characterized by simple mappings from scalars to display scalars. Specifically, for a scalar $s \in S$, define an embedding $E_S{:}I_S{\rightarrow}U$ by $E_S(b) = (\perp,...,b,...,\perp)$ (this notation indicates that all components of the tuple are $\perp$ except b) and define $U_S = E_S(I_S) \subseteq U$. Similarly, for a display scalar $d \in DS$, define an embedding $E_d{:}I_d \rightarrow V$ by $E_d(b) = (\perp,...,b,...,\perp)$ and define $V_d = E_d(I_d) \subseteq V$. These embedded scalar objects play a special role in the structure of display functions. In [7] we prove:

Proposition 2. If $D{:}U{\rightarrow}V$ is a display function, then we can define a mapping $MAP_D{:}S{\rightarrow}POWER(DS)$ such that for all scalars $s \in S$ and all for $a \in U_S$, there is $d \in MAP_D(s)$ such that $D(a) \in V_d$. The values of D on all of U are determined by its values on the scalar embeddings U_S (see [7] for the details). Furthermore,

(a) If s is discrete and $d \in MAP_D(s)$ then d is discrete,
(b) If s is continuous then $MAP_D(s)$ contains a single continuous display scalar.
(c) If $s \neq s'$ then $MAP_D(s) \cap MAP_D(s') = \phi$.

This tells us that display functions map scalars, which represent primitive variables like time and temperature, to display scalars, which represent graphical primitives like screen axes and color components. Most displays are already designed in this way, as, for example, a time series of temperatures may be displayed by mapping time to one axis and temperature to another as illustrated in Fig. 11. The remarkable thing is that Prop. 2 tells us that we don't have to take this way of designing displays as an assumption, but that it is a consequence of a more fundamental set of expressiveness conditions. Figure 12 in Sect. 4.4 provides a more detailed example of how a display function is defined by a set of mappings from scalars to display scalars.

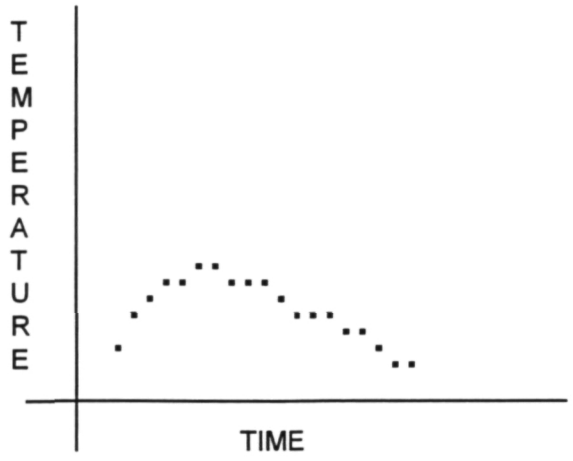

Figure 11. A time series displayed as a graph.

Display functions of the form $D:U \rightarrow V$ are polymorphic in that sense that they can be applied to data objects of any type in T. Furthermore, our lattice results show that we can define such functions in terms of a set of mappings from scalars to display scalars. Just as data flow systems define a user interface for controlling how data are displayed based on the abstraction of the rendering pipeline, we can define a user interface for controlling how data are displayed based on the abstraction of scalar mappings.

4. The VIS-AD Data Model

The VIS-AD (VISualization for Algorithm Development) system was designed to help scientists visualize their computations [5]. The system can be understood in terms of its data model, computational model and display model. The VIS-AD computational model is an imperative programming language of the type familiar to scientists (it is similar to C). The system's data model is based on the data lattice defined in Sect. 3.2. The data types and data objects of the lattice are just the types and objects of the VIS-AD programming language. Furthermore, metadata is integrated into this data model and plays a special role in the semantics of the programming language.

The biggest difference between the VIS-AD data model and the data lattice defined in Sect. 3.2 is the way that users define scalar types in S. A VIS-AD program defines scalar types as *real*, *real2d*, *real3d*, *int* or *string*. The *int* and *string* scalars are discrete scalars, and the *real* scalars are continuous scalars. The *real2d* and *real3d* scalars take pairs and triples of real intervals as values, and were included in the VIS-AD system to simplify the definition of spatial data types (e.g., the scalar *latitude_longitude* defined in the next section is a *real2d* scalar used as an index for 2-D image arrays).

4.1 Examples of User Defined Data Types

Users of VIS-AD can build types as arbitrary hierarchies of tuples and arrays, which provides the flexibility to adapt to scientific applications. VIS-AD's two- and three-dimensional real scalars make it easier to define types for spatial data like satellite images. The VIS-AD programming language provides a simple syntax for defining data types, as illustrated by the following examples taken from an algorithm for discriminating clouds in time series of multi-channel GOES (Geostationary Operational Environmental Satellite) images.

```
type ir_radiance = real;
type vis_radiance = real;
type latitude_longitude = real2d;
type time = real;
type image_region = int;
type count = int;
type goes_image =
  array [latitude_longitude] of
    structure {
      .im_ir = ir_radiance;
      .im_vis = vis_radiance;
    }
type goes_partition = array [image_region] of goes_image;
type goes_sequence = array [time] of goes_partition;
type histogram = array [ir_radiance] of count;
type histogram_partition =
  array [image_region] of
    structure {
      .hist_loc = latitude_longitude;
      .hist_hist = histogram;
    }
```

In these examples, a *goes_image* data object is an array of pixels indexed by *latitude_longitude* values, where each pixel is a structure containing infrared and visible radiances. A *goes_partition* object divides an image into regions, and includes a *goes_image* object for each value of *image_region*. A *goes_sequence* object is a *time* sequence of *goes_partition* objects. A *histogram* data object provides a frequency *count* of the number of occurrences of each *ir_radiance* value in an *image_region*, and a *histogram_partition* object associates a *histogram* object and a *latitude_longitude* value with each *image_region*.

4.2 Integrating Metadata with Programming Language Semantics

Unlike the situation in other programming languages, VIS-AD's arrays may be indexed by real values, or even by two- or three-dimensional real values. This is because VIS-AD's array types are defined as finite samplings of functional

relations from variables (i.e., from scalar types) to other data types. Thus metadata about the sampling of values is built into the semantics of the VIS-AD programming language. This has important consequences for the way that scientific data are manipulated and displayed. For example, an Earth satellite image is really a finite sampling of a continuous radiance field. If the pixels of an image are stored in an array in an ordinary language, the pixels are indexed in the array by integers, and the Earth locations of pixels must be managed separately. Thus the programming language has no information about the association between pixel values and their locations. However, if this satellite image is stored in a *goes_image* object, then the pixels are indexed with *latitude_longitude* values, and the programming language does have access to the locations of pixels. This enables the system to display a *goes_image* object in an Earth based frame of reference. If images from different sources (each with its own Earth projection) are overlaid in a display, the system can use the information about pixel locations to geographically register these images.

In the VIS-AD data model, all scalar values are managed in terms of finite samplings of infinite value sets. In addition to determining the values of array indices, this also determines the sampling and accuracy of values in arrays and tuples. For example, if a satellite sensor generates radiances as 8-bit quantities, then pixel values are really indices into a set of 256 samples of real radiance values. The scale of these real values may be a standard radiance, in which case the set of 256 values encodes the calibration of the satellite's sensor. Thus VIS-AD's management of sampling information can be used to encode satellite navigation and calibration information. Furthermore, sensor systems are fallible, so it is often the case the no value is defined for some pixels. In the VIS-AD data model, any data object or sub-object may take the special value *missing*, indicating the absence of information. Because missing values are part of the data model, they can be part of programming language semantics and display semantics.

We will use a satellite image example to illustrate how sampling information and missing data indicators are integrated with programming language semantics. In this example, we calculate the difference between images generated by different satellites. Let *goes_east*, *goes_west* and *goes_diff* be data objects of type *goes_image*, and let *loc* be a data object of type *latitude_longitude*. Assume that the *goes_east* and *goes_west* images were generated by GOES satellites at East and West stations over the U.S., so that they have different Earth perspectives. Then the following program calculates the difference between these images:

```
sample(goes_diff) = goes_east;
foreach (loc in goes_east) {
  goes_diff[loc] = goes_east[loc] - goes_west[loc];
}
```

The first line specifies that *goes_diff* will have the same sampling of array index values (i.e., of pixel locations) that *goes_east* has. The *foreach* statement provides a way to iterate over the elements of an array. In this case it iterates *loc* over the

pixel locations of the *goes_east* image. The expression *goes_west*[*loc*] resamples the *goes_west* image at the Earth location in *loc*. If *loc* falls in an area where there are no *goes_west* pixels, then *goes_west*[*loc*] evaluates to *missing*. VIS-AD's arithmetic operations evaluate to *missing* if any of their operands are *missing*, so if *goes_west*[*loc*] is *missing* then the difference *goes_diff*[*loc*] is also set to *missing*.

The VIS-AD programming language provides vector operations, so this little program can also be expressed as:

```
goes_diff = goes_east - goes_west;
```

The resampling of *goes_west* index values, and the evaluation to *missing* where there are no *goes_west* pixels, are implicit in this statement.

Users can access metadata about sampling and missing data explicitly. For example, the statement:

```
foreach (loc in goes_east) { ... }
```

iterates *loc* over the samples of the *goes_east* array. Missing data indicators may be explicitly accessed using these statements:

```
if (goes_east == missing) { ... }
goes_east[loc].im_ir = missing;
```

However, because of the special role of metadata in the semantics of the VIS-AD programming language, users rarely need to do explicit calculations with this metadata.

The integration of sampling information and *missing* data is generic, rather than specific to images. Thus the techniques illustrated in this satellite image example can be applied to any user-defined data types. As our simple programming example shows, this can relieve users of the need to explicitly keep track of missing data, the need to manage the mapping from array index values to physical values, and the need to check bounds on array accesses. The key to these advantages is that metadata is integrated into the data semantics of a programming language.

We can summarize the kinds of metadata that are integrated with the VIS-AD data model. They are:

1. Sampling information; every value in a data object is taken from a finite sampling of primitive values.
2. Missing data indicators; any value or sub-object in a data object may take the special value *missing* which indicates the lack of information.
3. Names for values; every primitive value occurring in a data object has a scalar type, and hence a name (i.e., the name of the scalar type).

Because these kinds of metadata are integrated with the data model, they are part of the computational and display semantics of the VIS-AD system. Note that the

VIS-AD programming language semantics do not integrate the accuracy information of interval values. However, this accuracy information could be integrated using interval arithmetic [10].

4.3 Other Types of Metadata

There are a great variety of kinds of metadata that scientists use to interpret their data. While some of these are integrated with the VIS-AD data model, the flexibility to define data types gives users a means to include other kinds of metadata in their data objects. For example, users of satellite images may want to manage the following kinds of information with their images:

1. Sensor identification. Satellites often have redundant sensors for measuring the same radiances, each with slightly different characteristics. Scientists sometimes need to know which sensor was used to generate a particular image.
2. Satellite sub-point. This is the Earth location (i.e., *latitude_longitude*) directly under the satellite, and is useful as a rough guide to image coverage.
3. Pixel scan rate. Images are often scanned over a significant time interval, and the scan rate in pixels per second can help assign precise times to pixel radiances.
4. Various measurements of the sensor systems, like voltages, temperatures and pressures. These are often used to diagnose problems with image quality.

We can create a new image type that includes these kinds of information, as follows:

```
type ir_radiance = real;
type vis_radiance = real;
type latitude_longitude = real2d;
type pixel_rate = real;
type sensor_id = string;
type temperature = real;
type voltage = real;
type annotated_goes_image =
  structure {
    .image_sensor = sensor_id;
    .image_subpoint = latitude_longitude;
    .image_pixel_rate = pixel_rate;
    .image_sensor_temp = temperature;
    .image_sensor_cathode = voltage;
```

```
    .image_data =
     array [latitude_longitude] of
      structure {
       .im_ir = ir_radiance;
       .im_vis = vis_radiance;
      }
}
```

While these kinds of metadata are not part of the semantics of the programming language, they are part of data objects and can be accessed by users' programs.

4.4 Data Display in VIS-AD

The VIS-AD display model is similar to the display lattice V described in Sect. 3.3, and is realized as a set of interactive, animated, 3-D voxel volumes. It is defined in terms of a set of display scalars that include:

- x, y and z coordinates of graphical marks in a 3-D volume
- *color* values of graphical marks
- a set of *contour* values; for each *contour* display scalar iso-surfaces and iso-lines are interpolated through the graphical marks in the 3-D volume
- an *animation* value; graphical marks whose *animation* value overlaps an animation index are selected for display
- a set of *selector* values, used to model abstract user control over display contents; the user selects a set of values for each *selector* display scalar, and only those graphical marks that overlap that set are displayed

Figure 12 illustrates the way that user's of VIS-AD control how their data types are displayed. An *image_sequence* data object is a time sequence of images with two spectral channels called *ir* (infrared) and *vis* (visible). Image pixels are indexed by pairs of real numbers specifying their Earth locations. Users define mappings from the scalar types of their application to the display scalar types that define the VIS-AD display model. The mappings indicated by arrows in Fig. 12 will cause an *image_sequence* data object to be displayed as an animated sequence of colored terrains, where the *ir* channel will determine the height of the terrain and the *vis* channel will determine its color. Users can interactively change the mappings from scalars to display scalars (e.g., change the mappings in Fig. 12 by mapping *ir* to *color* and mapping *time* to y - this will create a time series of images stacked up along the y axis). They can also interactively control the functions by which scalar values determine display scalar values (e.g., by adjusting color tables for the mapping of *vis* to *color* in Fig. 12). Data objects may be displayed according to multiple sets of mappings simultaneously.

C-2

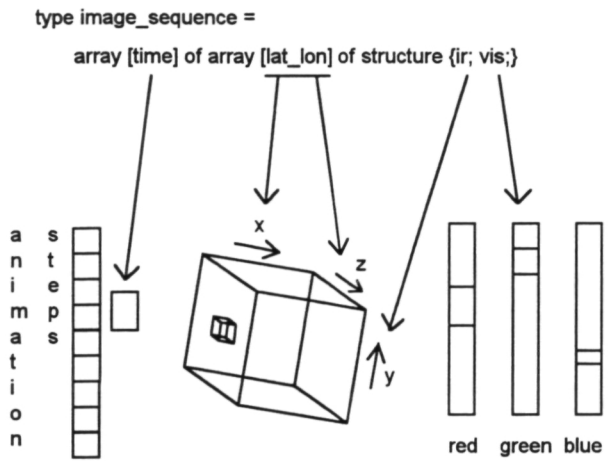

Figure 12. Users of VIS-AD control how data are displayed by defining mappings from the scalar types used to define complex data types to the display scalars used to define the VIS-AD display model.

The interface for controlling displays, consisting of the definitions of scalar mappings, is de-coupled from the VIS-AD programming language. This is important, because it allows users to control the display of their data objects without embedding explicit calls to display functions in their programs.

The VIS-AD system is available by anonymous ftp from iris.ssec.wisc.edu (144.92.108.63) in the pub/visad directory. The README file contains instructions for retrieving and installing the system.

5. Extending Data Lattices to More Complex Data Models

In Sect. 3.4 we described how a function $D:U \rightarrow V$ satisfying the expressiveness conditions must be a lattice isomorphism. Although we motivated this result in the context of a specific lattice structure for U and V (i.e., their members are sets of tuples of scalar values), the proof of this result only depends on U and V being complete lattices. Thus it is natural to seek to apply this result to other lattice structures for data and display models. The motive for new lattice structures must be new data models, since display models are themselves motivated by the need to visualize data. The data model defined in Sect. 3 includes tuples and arrays as ways of aggregating data. We will describe the issues involved in extending data lattices to data types defined by recursive domain equations, to abstract data types, and to the object classes of object-oriented programming languages. This discussions of this section are somewhat speculative.

5.1 Recursive Data Types Definitions

The denotational semantics of programming languages provides techniques for defining ordered sets whose members are the values of programming language expressions [4, 13, 14, 15]. An important topic of denotational semantics is the study of *recursive domain equations*, which define *cpos* (defined in the next paragraph) in terms of themselves.

First, we present some definitions used in denotational semantics. A *partially ordered set* (*poset*) is a set D with a binary relation $\leq$ on D such that, $\forall x, y, z \in D$

$x \leq x$	"reflexive"
$x \leq y \,\&\, y \leq x \Rightarrow x = y$	"anti-symmetric"
$x \leq y \,\&\, y \leq z \Rightarrow x \leq z$	"transitive"

A subset $M \subseteq D$ is *directed* if, for every finite subset $A \subseteq M$, there is an $x \in M$ such that $\forall y \in A.\ y \leq x$. A *poset* D is *complete* (and called a *cpo*) if every directed subset $M \subseteq D$ has a least upper bound *sup*(M) and if there is a least element $\perp \in D$ (i.e., $\forall y \in D.\ \perp \leq y$). If D and E are *posets*, a function $f{:}D \rightarrow E$ is *monotone* if $\forall x, y \in D.$ $x \leq y \Rightarrow f(x) \leq f(y)$. A function $f{:}D \rightarrow E$ is *continuous* if it is monotone and if $f(inf(M)) = inf(f(M))$ for all directed $M \subseteq D$. If D and E are *cpos*, a pair of continuous functions $f{:}D \rightarrow E$ and $g{:}E \rightarrow D$ are a *retraction pair* if $g \circ f \leq \mathbf{id}_D$ and $f \circ g = \mathbf{id}_E$. The function g is called an *embedding*, and f is called a *projection*.

We take the following example of a recursive domain equation from [12]. A data type for a binary tree may be defined by:

$$Bintree = (Data + (Data \times Bintree \times Bintree))_{\perp} \qquad (2)$$

Here "+", "×" and "$(.)_{\perp}$" are type construction operators similar to the tuple and array operators we discussed in Sect. 3.1. The "+" operator denotes a type that is a choice between two other types (this is similar to the *union* type constructor in the C language), "×" denotes a type that is a cross product of other types (this is essentially the same as our tuple operator, so that (*Data* × *Bintree* × *Bintree*) is a 3-tuple), and the "$\perp$" subscript indicates a type that adds a new least element $\perp$ to the values of another type. Equation (2) defines a data type called *Bintree*, and says that a *Bintree* data object is either $\perp$, a data object of type *Data*, or a 3-tuple consisting of a data object of type *Data* and two data objects of type *Bintree*. Intuitively, a data object of type *Bintree* is either missing, a leaf node with a data value, or a non-leaf node with a data value and two child nodes.

The obvious way to implement binary trees in a common programming language is to define a record or structure for a node of the tree, and to include two pointers to other tree nodes in that record or structure. In general, the self references in recursive type definitions can be implemented as pointers. Thus, recursive domain equations correspond to defining data types with pointers.

5.1.1 The Inverse Limit Construction

The equality in a recursive domain equation is really an isomorphism. As explained quite clearly by Schmidt in [12], these equation may be solved by the *inverse limit construction*. This construction starts with $Bintree_0 = \{\perp\}$, then applies (2) repeatedly to get

$$Bintree_1 = (Data + (Data \times Bintree_0 \times Bintree_0))_\perp$$
$$Bintree_2 = (Data + (Data \times Bintree_1 \times Bintree_1))_\perp$$
etc.

The construction also specifies a retraction pair $(g_i, f_i):Bintree_i \leftrightarrow Bintree_{i+1}$ for all i, such that g_i embeds $Bintree_i$ into $Bintree_{i+1}$ and f_i projects $Bintree_{i+1}$ onto $Bintree_i$. Then *Bintree* is the set of all infinite tuples of the form $(t_0, t_1, t_2, ...)$ such that $t_i = f_i(t_{i+1})$ for all i. It can be shown that *Bintree* is isomorphic with $(Data + (Data \times Bintree \times Bintree))_\perp$, and thus "solves" the recursive domain equation. The order relation on the infinite tuples in *Bintree* is defined element-wise, just like the order relation on finite tuples defined in Sect. 3.1, and *Bintree* is a *cpo*. We note that the inverse limit construction can also be applied to solve sets of simultaneous domain equations.

One way of extending our data lattices would be to show how to apply the inverse limit construction to recursive equations involving our tuple and array type constructors. Our tuple constructor is equivalent to the cross product operator "×". While our array constructor is similar to the function space operator "→" used in denotational semantics, it is not the same. $(A \rightarrow B)$ defines the set of all functions from A to B, while our array constructor (*array* [A] *of* B) defines the set of functions from finite subsets of A to B. Thus we would need to show how to apply the inverse limit construction to equations involving the constructor (*array* [A] *of* B). The *cpos* defined by the inverse limit construction are generally not lattices, but can always be embedded in complete lattices. Specifically, the Dedekind-MacNeille completion, described in [2], shows that for any partially ordered set A, there is always a complete lattice U such that there is an order embedding of A into U.

Note that the set of *Bintree* objects defined by the inverse limit construction includes infinite trees. This is because this set is complete and infinite trees are limits of infinite sequences of finite trees. The development of denotational semantics was largely motivated by the need to address non-terminating computations (the unsolvability of the halting problem showed that there was no way to separate terminating from non-terminating computations), and non-terminating computations may produce infinite trees as their values. Since our result that display functions are lattice isomorphisms depends on the assumption that data and display lattices are complete, it is likely that any extension of our data lattice to include solutions of recursive domain equations must include infinite data objects.

The inverse limit construction defines the set of data objects of a particular data type that solves a particular recursive domain equation. However, our approach in Sect. 3.2 was to define a large lattice that contained data objects of many different data types. It would be useful to continue this approach, by defining a lattice that includes all data types that can be constructed from our scalar types as tuples, arrays, and solutions of recursive domain equations. This is the subject of Sect. 5.1.2.

5.1.2 Universal Domains

A fundamental result of the theory of ordered sets is the *fixed point theorem*, which says that, for any *cpo* D and any continuous function $f{:}D{\rightarrow}D$, there is $fix(f) \in D$ such that $f(fix(f)) = fix(f)$ (i.e., $fix(f)$ is a fixed point of f) and such that $fix(f)$ is less than any other fixed point of f.

Scott developed an elegant way to solve recursive domain equations by applying the fixed point theorem [4, 14]. In a sense, the solution of a recursive domain equation is just a fixed point of a function that operates on *cpos*. Scott defined a *universal domain* U and a set R of retracts of U (this may be the set of all retracts on U, the set of projections, the set of finitary projections, the set of closures, or the set of finitary closures - note that these terms are defined in [7]). Then he showed that a set OP of type construction operators (these operators build *cpo*'s from other *cpo*'s) can be represented by continuous functions over R, in the sense that for $o \in OP$ there is a continuous function f on R that makes the diagram in Fig. 13 commute.

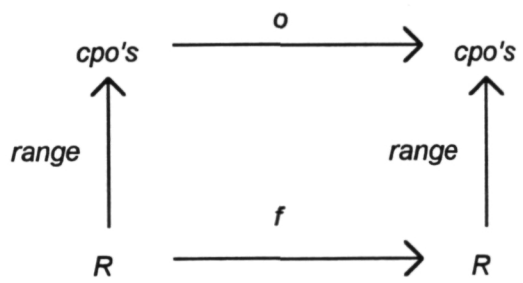

Figure 13. The type construction operator o is represented by function f.

Note that $range(w) = \{w(u) \mid u \in U\}$. For unary $o \in OP$ this is $range(f(w)) = o(range(w))$. Similar expressions hold for multiary operators in OP. Then, for any recursive domain equation $D = O(D)$ where O is composed from operators in OP, there is a continuous function $F{:}R{\rightarrow}R$ that represents O. By the fixed point theorem, F will have a least fixed point $fix(F)$, and $O(range(fix(F))) = range(F(fix(F))) = range(fix(F))$, so $range(fix(F))$ is a *cpo* satisfying the recursive domain equation $D = O(D)$. The solution of any domain equation (or any set of simultaneous domain equations) involving the type construction operators in OP

will be a *cpo* that is a subset of the universal domain U. Thus this approach is similar to the way that the data types of our data model define sets of data objects that are embedded in a single data lattice.

Universal domains and representations have been defined for sets OP that include most of the type constructors used in denotational semantics, including "+", "×", "→", and "$(.)_\perp$". In order to apply universal domains to extend our data model to include recursively defined types, we would need to show how our tuple and array type constructors can be represented over some universal domain.

A common example of a universal domain is the set *POWER*(**N**), which is just the set of all subsets of the natural numbers **N** (i.e., non-negative integers). *POWER*(**N**) is a complete lattice. However, it does not include natural embeddings of our scalar data objects. Furthermore, the embeddings of mathematical types into universal domains, as defined by papers in denotational semantics, are not suitable for our display theory. For example, a simple integer and a function from integers to integers are embedded to the same member of *POWER*(**N**). A display function applied to the lattice *POWER*(**N**), with these embeddings, would produce the same display for the integer and the function from integers to integers. Since the goal of visualization is to communicate information rather than to make it obscure, other embeddings of types into universal domains must be developed. Specifically, an extension of our display theory to recursively defined data types should include a universal domain with natural embeddings of our scalar data types, and should include representations of our tuple and array type constructors that will produce natural embeddings of constructed types.

5.1.3 Display of Recursively Defined Data Types

Since the goal of visualization is to communicate the information content of data to users, an extension of our theory must focus on the data lattice U. However, since a display function D is a lattice isomorphism of U onto a sub-lattice V, we should be able to say some things about the structure of V. If a subset $A \subseteq U$ is the solution of a recursive domain equation, then $D(A) \subseteq V$ is isomorphic to A and must itself be a solution of the recursive domain equation.

For example, if the set A is the solution of (2) for *Bintree*, then the set $D(A)$ must also solve this equation. The isomorphism provides a definition of the operators "+", "×" and "$(.)_\perp$" in $D(A)$ and thus also defines a relation between objects and their "subtree" objects in $D(A)$. The isomorphism does not tell us how to interpret these operators and relations in a graphical display, but it does tell us that such a logical structure exists. Given the complexity of this structure, it seems likely that display objects in $D(A)$ will be interpreted using some graphical equivalent of the pointers that we use to implement data objects in A.

Two graphical analogs of pointers come to mind immediately:

1. Diagrams. Here icons represent nodes in data objects, and lines between icons represent pointers.

2. Hypertext links. Here the contents of a window represents one or more nodes in a data object, and an icon embedded in that window represents an interactive link to another node or set of nodes. That is, if the user selects the icon (say by a mouse point and click), new window contents appear depicting whatever the icon points at.

In order to extend our display theory to data types defined with recursive domain equations, we need to extend our display lattice V to include these graphical interpretations of pointers. The most interesting problem is to find a way to do this that produces a display lattice complex enough to be isomorphic to a universal domain as described in Sect. 5.1.2.

5.2 Abstract Data Types and Object Classes

Abstract data types and the object classes of object-oriented programming are ways of defining data types that hide the internal structures of data objects from the programs that use those data objects. Definitions of abstract data types and object classes include definitions of *member functions* for basic operations on data objects. Data objects are accessed by applying these member functions, rather than by selecting their primitive sub-objects. In fact, the hidden implementation of data objects may not include sub-objects at all, but may be purely functional. For example, an array data object may be implemented either by explicitly storing elements of the array, or by a function for computing the elements of the array as they are accessed.

5.2.1 Abstract Data Types

In an algebraic setting [17], abstract data types are specified by a signature $\Sigma = (T, F)$ and a set of logical conditions E. T is a set of types and F is a set of member functions among the types in T (that is, the types of the member functions, and the numbers and types of their arguments, are specified). E is a set of first order statements involving quantifiers, equality, the member functions of F, and variables with types in T. It is undecidable whether the statements in E are satisfiable (i.e., no algorithm exists which can tell, for given E, whether any set of data objects and functions satisfy E), so there are no compilers that produce implementations from T, F and E. However, abstract data types are used as the basis of programming methods. System designers use heuristic methods to derive conditions in E by analyzing system requirements, and use these conditions as a guide for implementing the functions in F [9].

Because of the generality of the abstract data type formalism, it can be used to express our lattice theory of display. To see this, define $T_{\mathrm{LAT}} = \{U, V\}$, $F_{\mathrm{LAT}} = \{inf_U: U\times U\rightarrow U, sup_U: U\times U\rightarrow U, inf_V: V\times V\rightarrow V, sup_V: V\times V\rightarrow V, D: U\rightarrow V\}$ and $E_{\mathrm{LAT}} =$ {lattice axioms for U and V, expressiveness conditions on D}. That is, the data types of T_{LAT} are the data and display lattices U and V, the functions in F_{LAT} are the lattice operations on U and V plus the display function D, and the logical

conditions in E_{LAT} are the axioms defining U and V as lattices plus the expressiveness conditions. Expressiveness Conditions 1 and 2, as defined in Sect. 3.4, quantify over $MON(U \rightarrow \{\perp , 1\})$ and are thus second order statements whereas E_{LAT} is supposed to consist of first order statements. However, as shown by Prop. 1, Conditions 1 and 2 are equivalent to conditions that can be expressed as first order statements. There are obviously many different sets of lattice U and V satisfying the abstract data type definition in T_{LAT}, F_{LAT} and E_{LAT}.

In Sect. 5.2.3 we will discuss the issues involved with extending our lattice theory to display the data objects of abstract data types.

5.2.2 Object Classes

Like abstract data types, the object classes of object-oriented programming languages define access to data objects in terms of a set of member functions. However, rather than defining logical conditions that the member functions must satisfy, object classes define these functions explicitly as programs. An *object class* in C++ defines members as both data structures and functions, which are divided into private and public parts [3]. The private members are only accessible from member functions defined as part of the class (i.e., they are not accessible from outside the class definition).

In addition to hiding the implementation of object classes, object-oriented languages provide two mechanisms called polymorphism and inheritance that provide a novel style of programming compared to traditional procedural languages. *Polymorphism* means that the same member function name may be defined in the public parts of multiple object classes. Calls to member functions are bound to the appropriate function definitions at run time, determined by the classes of the objects passed as arguments to the functions. *Inheritance* allows an object class to be defined in terms of another. The new class "inherits" the members of the old class, except where those data members are explicitly redefined.

Object-oriented visualization systems define polymorphic display functions. These systems partition their data models into a number of object classes, and their polymorphic display functions may be called to display any data object in their data models. Thus the object oriented approach and our lattice approach both define display functions that can be applied to data objects of any type. However, where the object oriented visualization systems require constructive definitions of display functions as programs, our approach defines data and display lattices and defines display functions as any function satisfying the expressiveness conditions. Thus it is still interesting to investigate how our lattice theory can be extended to the object classes of an object-oriented language.

The private and public members of class definitions include data structures. These may be displayed using the techniques that we developed in Sect. 3, and the techniques for displaying recursively defined data types suggested in Sect. 5.1. However, this approach does not provide a systematic way to display data objects defined by classes, since class definitions include functions as well as data

structures. In the next section we will discuss issues involved in extending our lattice model of display to the member functions of object classes.

5.2.3 Lattice Models for Abstract Data Types and Object Classes

Because of the similarity between abstract data types and object classes, we will discuss them together, using the notation of abstract data types. We let T denote a set of types and let F denote a set of member functions. The types in T may be abstract data types or may be object classes, and the functions in F may be defined by a set E of logical conditions or by a set of programs. The important point is that the functions in F define relations among data objects, and that these relations take the place of a definition of data objects in terms of their internal structure. In fact, we could say that the relations defined by functions in F are a generalization of the relations between objects and their sub-objects that define the internal structures of data objects. In Sect. 3.1 we defined tuple objects in terms of their element sub-objects, and we defined array objects in terms of their domain and range sub-objects. These relations between objects and their sub-objects are special cases of relations between objects expressed by member functions.

Let A be the union of the sets of data objects of all types in T. We could give A the discrete order (i.e., no object is greater than any other object), but this would lead to a very boring theory of data display. In order to define a more interesting order relation on A, we can start with the data lattice that we defined in Sect. 3.2 (here we call it U_0). If T is a set of abstract data types, then we let the objects of U_0 serve as constants in the logical conditions of E. If T is a set of object classes, then we take the continuous and discrete scalar types of U_0 as the primitive values of an object oriented programming language.

If we also assume that the member function in F are monotone, then we can use these functions to derive order relations between objects in A from the order relations between objects in U_0. However, there is no guarantee that there is a order relation on A that is consistent with the assumption that the functions in F are monotone. For example, while it is easy to define monotone arithmetic operators on the scalar types of U_0, there is no reasonable way to define monotone logical and comparison operators on the scalar types of U_0 (we run into inconsistencies assuming either that *false* $\leq$ *true* or that *true* and *false* are not ordered). This suggests that a monotonicity assumption is a severe restriction on the member functions in F.

However, in order to define an interesting order condition on A we may assume that member functions are monotone. In this case, we need to verify that the monotone functions of F are consistent with an order relation on A, although this appears to be a difficult problem. If T is a set of object classes, and if member functions are implemented in a programming language that includes logical and comparison operators, then it is generally undecidable whether functions defined among the objects of U_0 are monotone. However, we may be able to design a restricted programming language for member functions that allows us to verify that monotone member functions are consistent with an order relation on A. If T is a set

of abstract data types, then the monotonicity requirement must be added to E as a set of logical conditions on the member functions in F (along with conditions that define order relations on the types in T). This may cause a set of satisfiable conditions to become unsatisfiable, and it is generally undecidable whether the addition of monotonicity conditions causes a set E of conditions to become unsatisfiable. Of course, this situation is no worse than without monotonicity assumption, since the question of whether a set of logical conditions is satisfiable is generally undecidable.

Given an ordered set A of data objects (the union of the sets of data objects of each type in T), we can use the Dedekind-MacNeille completion to embed A in a complete lattice U. In order to apply our display theory we would need to construct a display lattice V such that isomorphisms from U onto sub-lattices of V exist, and develop interpretations of display objects in V in terms of a physical display device. Since the display function $D{:}U{\rightarrow}V$ is an isomorphism between the set of data objects and a subset of the set of display objects, and since the relations between data objects expressed by the member functions in F (and subject to the logical conditions in E) are a generalization of the hierarchical relations between objects and sub-objects in the data model defined in Sect. 3.1, it is natural to seek an interpretation of display objects in terms of relations between display objects that generalizes the relation between display objects and graphical marks as described in Sect. 3.3. For example, we may represent data objects by icons in a display, and let users interactively explore the relations between those icons as defined by the functions in F. Finding a systematic way to interpret displays of abstract data types seems like a very open ended and interesting problem.

It is interesting to note that in the case of abstract data types, we can use the generality of the framework to add our display model to an existing set of abstract data types defined by T, F and E. Take T_{LAT}, F_{LAT} and E_{LAT} as defined in 5.2.1, and define

$$
\begin{aligned}
T' &= T \cup T_{\mathrm{LAT}} \\
F' &= F \cup F_{\mathrm{LAT}} \cup \{\text{embeddings of the types in } T \text{ into } U\} \\
E' &= E \cup E_{\mathrm{LAT}} \cup \{\text{monotonicity conditions on the embeddings} \\
&\qquad\qquad \text{from } T \text{ into } U\}
\end{aligned}
$$

Of course, there is no algorithm for deciding if the conditions in E' are satisfiable or for constructing the lattice U and V if they are. Furthermore, this tells us nothing about how display objects in V are interpreted in terms of a physical display device.

6. Conclusions

The design of the VIS-AD data model is tightly integrated with a computational model and a display model. The result is a data model whose data objects can be uniformly visualized using polymorphic display functions, and which has the flexibility to adapt to scientists' computations. Several kinds of metadata are integrated with this data model, providing a novel and useful programming

language semantics, and also providing the capability to display multiple data objects in common frames of reference.

The VIS-AD data model is based on lattices. These lattices may be applied to models of both data and displays. This provides an interesting context for analyzing visualization processes as functions from data lattices to display lattices. There are also interesting prospects for extending the lattice theory of visualization to more complex data models that involve recursively defined data types, abstract data types, and the object classes of object oriented programming languages.

References

[1] Bertin, J., 1983; Semiology of Graphics. W. J. Berg, Jr. University of Wisconsin Press.

[2] Davey, B. A. and H. A. Priestly, 1990; Introduction to Lattices and Order. Cambridge University Press.

[3] Gorlen, K. E., S. M. Orlow and P. S. Plexico, 1990; Data Abstraction and Object-Oriented Programming in C++. John Wiley & Sons.

[4] Gunter, C. A. and Scott, D. S., 1990; Semantic domains. In the Handbook of Theoretical Computer Science, Vol. B., J. van Leeuwen ed., The MIT Press/Elsevier, 633-674.

[5] Hibbard, W., C. Dyer and B. Paul, 1992; Display of scientific data structures for algorithm visualization. Visualization '92, Boston, IEEE, 139-146.

[6] Hibbard, W., C. Dyer and B. Paul, 1993; A lattice theory of data display. Submitted to IEEE Visualization '94.

[7] Hibbard, W. L., and C. R. Dyer, 1994; A lattice theory of data display. Tech. Rep. # 1226, Computer Sciences Department, University of Wisconsin-Madison. Also available as compressed postscript files by anonymous ftp from iris.ssec.wisc.edu (144.92.108.63) in the pub/lattice directory.

[8] Mackinlay, J., 1986; Automating the design of graphical presentations of relational information; ACM Transactions on Graphics, 5(2), 110-141.

[9] Mitchell, R., 1992; Abstract Data Types and Modula-2: a Worked Example of Design Using Data Abstraction. Prentice Hall.

[10] Moore, R. E., 1966; Interval Analysis. Prentice Hall.

[11] Robertson, P. K., R. A. Earnshaw, D. Thalman, M. Grave, J. Gallup and E. M. De Jong, 1994; Research issues in the foundations of visualization. Computer Graphics and Applications 14(2), 73-76.

[12] Schmidt, D. A., 1986; Denotational Semantics. Wm.C.Brown.

[13] Scott, D. S., 1971; The lattice of flow diagrams. In Symposium on Semantics of Algorithmic Languages, E. Engler. ed. Springer-Verlag, 311-366.

[14] Scott, D. S., 1976; Data types as lattices. Siam J. Comput., 5(3), 522-587.

[15] Scott, D. S., 1982; Lectures on a mathematical theory of computation, in: M. Broy and G. Schmidt, eds., *Theoretical Foundations of Programming Methodology*, NATO Advanced Study Institutes Series (Reidel, Dordrecht, 1982) 145-292.

[16] Treinish, L. A., 1991; SIGGRAPH '90 workshop report: data structure and access software for scientific visualization. Computer Graphics 25(2), 104-118.

[17] Wirsig, M., 1990; Algebraic specification. In the Handbook of Theoretical Computer Science, Vol. B., J. van Leeuwen ed., The MIT Press/Elsevier, 675-788.

N1 S

A Highly Parallel Approach for Satellite Archive Processing

William Hibbard

Space Science and Engineering Center
University Of Wisconsin - Madison

1. INTRODUCTION

The Space Science and Engineering Center (SSEC) has an archive of almost all the GOES (Geostationary Operational Environmental Satellite) data produced since 1978, stored on roughly 26,000 videocassettes containing 120 trillion bytes (Suomi, 1982). SSEC is investigating a system for periodically processing this entire data set to derive products such as cloud heights and effective emissivity, cloud classification, cloud motion winds, humidity fields, fire frequency in the tropics, and diurnal variability of cloudiness. SSEC has set the goal of processing the entire archive once per year. This paper describes a highly parallel approach to implementing a climate archive processing system. This approach has reduced projected hardware costs for the GOES archive system to a fraction of the total cost of archive processing, compared to hardware costs that dominate total costs for other approaches.

2. SOLUTION BY PARTITION

Our system design concept is based on the assumption that the archive can be partitioned into sections, that the data in these sections can be independently processed into intermediate products reduced in volume by two or three orders of magnitude, and that any necessary combinations of data between sections can be done using these intermediate products. In the case of the GOES archive, the sections would consist of the data from one satellite over a time period such as a week or a month. Existing algorithms for most GOES products can be applied independently within these sections. Cloud classification (Garand, 1982), percentage of cloud cover (Coakley and Bretherton, 1982), cloud heights, effective emissivity, and humidity fields are derived from a single GOES image. Cloud drift winds (Merrill, 1989) are derived from short time sequences of visible or infrared images from a single GOES satellite.

The assumption of partitioning the archive into sections allows the archive processing system to be partitioned into many loosely coupled identical subsystems, each processing one or more sections of the archive. The raw archive data would not flow between subsystems. Instead, the much smaller volume of processed data would flow from the subsystems into a central product store. Because the raw data do not need to flow through any single system component, very high bandwidth components can be avoided, reducing cost and risk and increasing maintainability.

The GOES climate archive system would be implemented as a set of subsystem units, where each unit consists of a RISC (Reduced Instruction Set Computer - often used to mean fast and inexpensive computer) processor, an archive tape playback unit, a disk store, and a network connection. The units would be connected over a network to a larger central processor and store for accumulating intermediate and final products. If the number of units is very large, then they may be divided among several networks, each connected to the central processor. We estimate that each unit will cost between $20K and $40K, so the number of units may be large.

Dividing the 120 trillion bytes in the GOES archive by the 30 million seconds in a year gives an average processing rate of 4 million bytes per second. Assuming playback at the normal GOES stretched data transmission rate of 0.25 million bytes per second, this works out to about 17 processing units, including a spare. A 40 MIPS (Million Instructions Per Second) RISC processor can allocate 188 instructions to processing each visible pixel or 3150 instructions to processing each IR pixel in the GOES stream. Thus it should be possible to generate many types of products "on the fly" as the data are being ingested from the archive tape into the RISC processor. In one experiment run on the IBM RISC 6000 Model 320, a program written in C was able to unpack packed 6 bit GOES visible pixels into one pixel per byte, at a rate 17 times faster than the normal GOES ingest rate.

Where it is not possible to process data at the ingest rate, the data can be ingested onto local disk store and processed after ingest. In this case the system would periodically stop ingesting data and allow processing to catch up, resulting in a slower average rate of processing and a need for a larger number of processing units. As the system work load increases through the addition of new products and the modification of product algorithms, a system composed of many identical units may be expanded gradually, by increasing the number of units.

PRECEDING PAGE BLANK NOT FILMED

System management is potentially a large problem for the highly parallel approach. The system must:

Manage the flow of archive tapes through processing and keep track of the status of each tape.

Manage the product algorithms running in each unit, making sure that they are consistent, or that variations are intentional and understood.

Monitor the correct functioning of the units, and manage overall operations on a changing subset of operational units (with a large number of units, some will normally be down).

3. A SCALABLE APPROACH

The Earth Observing System (EOS) is expected to generate much larger volumes of data than GOES and other current instruments. A good processing system design should be able to scale to handle larger data volumes. System designs with single point data bottlenecks can only scale by increasing the bandwidth of their components. However, a system composed of many units can scale by increasing the number of units. Ultimately the scalability of our design concept is limited by the central product store, which is a single point for products, and by the complexity of managing and operating a large number of units. Given a reduction of volume from raw data to products by a factor of 100, and a central product store with 10 times the bandwidth of each replicated unit, the system design may scale to 1000 units. However, operations would be extremely complex for a system of 1000 units. As the number of units increases, the system should probably evolve into a hierarchical organization, with multiple physical networks and multiple central product stores. This would prevent any single point failure from bringing down a large and expensive system, and it would allow a partitioning of system management functions into a number of smaller systems. At such large scales it would also be very desirable to avoid replicating the tape playback function for each unit. 1000 tape units and a correspondingly large tape archive would require a large number of people for operations and maintenance.

Bandwidths of tape media, processors, networks and disk stores will increase. Nevertheless, the assumption of a partitioned archive, and the consequent highly parallel approach to processing system design, can help us avoid the cost and risk of using the highest bandwidth components.

4. ACKNOWLEDGEMENTS

The ideas presented here grew out of collective discussions with Francis Bretherton, John Anderson, Robert Fox, J. T. Young, Joe Rueden, Tom Whittaker, John Benson and others at the Space Science and Engineering Center.

5. REFERENCES

Coakley, J., and F. Bretherton, 1982; Cloud cover from high resolution scanner data: detecting and allowing for partially filled fields of view; J. Geophys. Res., 87, 4917-4932.

Garand, L., 1987; Automated classification of oceanic cloud patterns with applications to cloud type dependent retrievals of meteorological parameters; Digital Image Proc. and Visual Comm. Technologies in Meteor., Cambridge, SPIE, 47-53.

Merrill, R., 1989; Advances in the automated production of wind estimates from geostationary satellite imagery. 4th Conf. on Satellite Meteor., San Diego, AMS, 246-249.

Suomi, E., 1982: The Videocassette GOES Archive System - 21 Billion Bits on a Videocassette. IEEE Trans. on Geoscience and Remote Sensing, GE-21(1), 119-121.

N15

PROGRESS WITH VIS-5D / DISTRIBUTED VIS-5D

Brian Paul [1], Andre Battaiola[2] and Bill Hibbard[1]
[1]Space Science and Engineering Center
University of Wisconsin - Madison
[2]INPE/CPTEC, Brazil

1. INTRODUCTION

VIS-5D is a system for interactive visualization of large atmospheric data sets such as those generated by numeric weather simulations. It is a simple yet powerful tool used by scientists for both analysis and presentation of their data. VIS-5D is freeware; one can obtain the software at no charge, evaluate it, and use it if it fits one's needs.

VIS-5D was first demonstrated at the AMS annual meeting in January 1989 and described in a paper (Hibbard and Santek 1990). Originally, VIS-5D was only available on Stellar workstations. Since then we have ported the software to SGI and IBM workstations. Coupled with the dramatic drop in prices for 3-D graphic workstations, the number of people using VIS-5D has increased greatly.

Distributed VIS-5D is a variation of VIS-5D designed for visualization in a network environment. With this system, the data do not have to be on the workstation to visualize it. By splitting the program into two parts which run on a supercomputer and graphics workstation, larger data sets can be visualized than practical with a single workstation.

2. OVERVIEW OF VIS-5D

VIS-5D works with multi-variable, time varying data in the form of a five-dimensional grid. Three dimensions correspond to space, one dimension is time, and another data dimension enumerates multiple physical variables. For example, a thunderstorm data set may contain physical variables for potential temperature, pressure, cloud water, and wind speed spanning 50 time steps and a spatial volume of size 30 by 30 by 25 grid points. In this case, the data set would contain 4.5×10^6 data points (4*50*30*30*25). On a workstation with 256 MB of memory, VIS-5D can visualize data sets containing as many as 1.25×10^8 points.

The three space dimensions form a regular-spaced, rectangular domain displayed as a 3-D box on the screen. The physical variables which comprise the data set can be depicted inside the box in the following formats:

- Iso-level contour surfaces with optional transparency.
- Horizontal and vertical contour line slices with selectable contour interval.
- Horizontal and vertical color slices with interactive data-to-color mapping.
- Interactive wind trajectory tracing using a 3-D cursor.
- Horizontal and vertical wind vector slices with adjustable scaling and density.

VIS-5D is completely controlled using a graphical user interface. The mouse is used to rotate, move, and animate the 3-D volume while buttons, sliders, and type-in widgets control program options.

Most data sets visualized with VIS-5D are time varying. The user controls animation and single stepping, either forward and backward in time. A combination analog/digital clock in the 3-D viewing window shows the current time.

To manage the display of the physical variables in the various formats there is a widget button for every possible combination of variable and depiction format. As implemented, the control panel window contains a matrix of buttons in which each physical variable is represented by a row and each depiction format is represented by a column. To display a vertical contour slice of the variable TEMP only requires clicking on the button in the TEMP row and vertical contour slice column. Similarly, if U, V, and W wind components are available, extra buttons will be present for wind vector slices and trajectories.

When the display of any variable is activated, a pop-up window will appear which contains any applicable options. For iso-level contour surfaces, the pop-up window contains a slider to change the iso-level. For wind vector slices, the pop-up window contains type-in widgets to change the vector lengths and density. Only one options window for each type of graphic is displayed at a time to prevent screen clutter.

In addition to graphical widgets, direct manipulation techniques are used in VIS-5D. For example, the default location of a horizontal or vertical slice is through the center of the box. To move the slice to a different location we let the user drag it directly with

 PRECEDING PAGE BLANK NOT FILMED

the mouse. Similarly, when tracing wind trajectory motions an initial location must be selected in the 3-D volume. This is done by dragging a 3-D cursor inside the box using the mouse. We have found that direct manipulation is the best way to handle these interactions because it is simple to use and it eliminates extra slider or dial widgets.

Other features of VIS-5D include:

- User-definable topography and map lines.
- Text annotations with any selected font.
- Antialiasing, transparency, line width, and full-screen rendering options.
- Save and restore of current graphics.
- Graceful handling of missing data.

Because VIS-5D was designed to deal with a singe type of data it works very efficiently. The 5-D grid data is compressed and the internal representation of iso-level contour surfaces, slices, etc. is in a compressed format as well. VIS-5D also utilizes multiple processors to compute the iso-level contour surfaces or slices for each time step in parallel. By using coarse grained parallel programming, near linear speedups can be obtained by increasing the number of processors. For these reasons a number of people have turned to VIS-5D after finding that other visualization systems are too slow or limit data capacity.

3. VIS-5D IN USE

VIS-5D has been used by atmospheric scientists at the University of Wisconsin and other sites for several years now. During that time it has evolved to serve three distinct purposes:

- As a debugging/analysis tool: Those involved in atmospheric modeling use VIS-5D as a diagnostic tool. Typically, after a number of time steps have accumulated, VIS-5D will be used to visually inspect the progress of the model. If there is a problem it can be quickly discovered and diagnosed.

- As a teaching tool: It is not uncommon for a teacher or researcher to bring groups of people into the lab for a VIS-5D session. Concepts which are difficult to explain verbally or display in 2-D can be naturally illustrated with VIS-5D.

- As a presentation tool: It has become common practice for scientists to produce a VIS-5D video to accompany a research paper for presentation at conferences. During only an afternoon's time, some very informative and interesting videos have been made.

Overall, users seem to be very happy with VIS-5D's user interface. It has been our experience that people become proficient with the program after only one or two sessions and minimum instruction. This is very important because the average scientist usually does not have time to spend learning a complicated new software system.

4. DISTRIBUTED VIS-5D

The purpose of Distributed VIS-5D is to allow visualization of very large data sets (1.0×10^{10} data points). This is accomplished by dividing the visualization workload between two computers. Essentially, the VIS-5D program is split into two parts: a client and a server. The client program runs on the user's graphic workstation. It provides the user interface and displays the 3-D graphics. The server runs on the computer where the user's data are stored, typically a supercomputer, and is responsible for converting data into geometric primitives. When the user wants to view an iso-level contour surface, slice, etc., a request is sent from the client to the server. The server receives the request, fetches the appropriate data, computes the iso-level contour surface or slice, and sends the resulting geometric description back to the client where it is displayed.

The requests from the client to the server require little network bandwidth. However, the resulting geometric descriptions sent back may be quite large. Descriptions of iso-level contour surfaces computed from a 50 by 50 by 25 point grid are often 100KB in size. A high speed network is required to sustain interactive rates. In fact this project is in part funded by the Gigabit Network Testbed project and intended to be a test application of the network.

Our testing was done using an SGI 340VGX workstation at the UW Madison and a CRAY-2 supercomputer at the NCSA in Urbana, IL connected by a T-1 (1.5 Mbps) network. We discovered two main problems: network congestion and supercomputer process scheduling. Since we were certainly not the only users of the network it was inevitable that we would be competing for bandwidth with other users. When the network was congested we experienced a dramatic decrease in bandwidth but only moderate increases in latency. We found just the opposite problem while using the CRAY-2; the server process would get alternating short periods of high throughput followed by long periods of low throughput. The combination resulted in the user observing wildly varying performance.

For example, if the user selected an iso-level contour surface for viewing and then turned on the

animate option he would see a very uneven frame rate. This is because Distributed VIS-5D only sends requests to the server on an as-needed basis. When animation is selected, a sequence of iso-level contour surface requests are sent to the server. Since the server is a parallel program and iso-level contour surfaces may require different amounts of time to be computed they may be send back to the client out of order. This, compounded with the network and scheduling problems, results in an uneven frame rate. In an attempt to solve the problem, we experimented with various techniques to adaptively control the animation rate depending on request/result turn around time. While we were mostly successful, we found no way to effectively deal with unexpected periods of zero throughput.

What does this mean for the user? During hours of heavy network or supercomputer use, one encounters a cycle of delays and bursts in responsiveness. This can be very distracting because slowness and delays make the user wonder if the system has crashed. The solution is to increase the network speed and to change the process scheduling on the supercomputer to give a more even throughput. We may experiment in both of these areas when our network is upgraded to T3 (45 Mbps) and by porting the server to another supercomputer.

Despite the complications involved in distributed visualization, it can be useful as long as it allows you to do something not possible with a single workstation. In our case Distributed VIS-5D allows visualization of much larger data sets than possible previously.

5. SUMMARY

The developers of VIS-5D have worked closely with its users resulting in an exceptional balance between ease-of-use and features.

Three important themes run throughout VIS-5D:

- Interactivity: VIS-5D gives quick responses to user input. When the user does not have to wait for results, he is more likely to try new ideas and explore data in more detail.

- Ease of use: Having a graphical user interface is not sufficient to qualify a program as easy to use. Our philosophy has been "less is more" when it comes to windows and widgets - we avoid presenting our users with unnecessary and confusing choices.

- Versatility: While VIS-5D has a rich variety of visualization methods, the program itself can be used in several unique ways.

There should be no question about the value of scientific visualization. We are now concerned with how to best apply the technology and make it accessible to the people who need it.

6. OBTAINING VIS-5D

VIS-5D is available via anonymous ftp:

```
% ftp vis5d.ssec.wisc.edu
(or % ftp 144.92.108.66)
login:  anonymous
password:  myname@mylocation
ftp> cd pub/vis5d
ftp> ascii
ftp> get README
ftp> bye
```

See section 2 of the README file for complete installation instructions. Since VIS-5D is freeware there is no warranty of any kind. However, we will try to help with any questions or problems addressed to us.

7. ACKNOWLEDGEMENTS

This work was supported by NASA (NAS8-36292 and NAG8-828). This work was also supported by the National Science Foundation and the Defense Advanced Research Projects Agency under the Cooperative Agreement NCR-8919038 with the Corporation for National Research Initiatives.

8. REFERENCES

Hibbard, W., and D. Santek, 1990; The VIS-5D system for easy interactive visualization. Visualization '90, San Francisco, IEEE. 28-35. Also in 1991 Preprints, Conf. Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology. New Orleans, American Meteorology Society. 129-134.

Hibbard, W., D. Santek, and G. Tripoli, 1991; Interactive atmospheric data access via high speed networks. Computer Networks and ISDN Systems, 22, 103-109.

N/S

VIS-AD DATA MANAGEMENT

William Hibbard[1&2], Charles R. Dyer[2] and Brian Paul[1]

[1]Space Science and Engineering Center
[2]Department of Computer Sciences
University of Wisconsin - Madison

1. INTRODUCTION

The VIS-AD system was designed as an algorithm development system, enabling scientists to visualize the results of experiments with their data analysis algorithms. However, VIS-AD could also become a very powerful data management system. VIS-AD provides a programming language similar to C for expressing scientific algorithms. The language provides a mechansim for users to define complex data types for the data objects of their algorithms. Data types can be defined for images, multi-spectral images, image sequences, gridded data, randomly located data, geometrical data such as boundary lines, and virtually any other data structures used in earth science. Furthermore, VIS-AD manages all these data types in a uniform way, and provides access to data through its programming language. These features could form the basis for a new way to manage environmental data.

2. DATA TYPES

VIS-AD enables its users to define data types for the data objects of algorithms, as follows. Define T as the set of types for the data objects in an algorithm. It is common for a programming language to define a set of primitive types (e.g. *int*, *real*), and to define a set of type constructors for building the types in T from the primitive types. We modify this by interposing a finite set S of scalar types between T and the primitive types. We define the primitive types as:

$$PRIM = \{int, string, real, real2d, real3d\}$$

where *real2d* and *real3d* are pairs and triples of real numbers. The user defines a finite set S of scalar types, and binds them to the primitive types by a function $P: S \rightarrow PRIM$. An infinite set T of types can be defined from S by:

$$S \subset T$$
$$(\text{for } i = 1,\ldots,n. t_i \in T) \Rightarrow (t_1,\ldots,t_n) \in T$$
$$(s \in S \wedge t \in T) \Rightarrow (s \rightarrow t) \in T$$

where $(t_1,\ldots,t_n)$ is a tuple type constructor with element types t_i, and $(s \rightarrow t)$ is an array type constructor with value type t and index type s.

The important consequence of the use of scalar types is that every primitive value, including an array index value, occurring in a data object of type t [illegible], has a scalar type in S. The names of primitive values are a form of ancillary information, and the scalar types are a way of requiring the user to supply this ancillary information with data objects. The VIS-AD system uses this ancillary information to intelligently generate graphical depictions of data objects, but it may also be useful for supporting other intelligent data management functions.

3. EXAMPLES OF DATA TYPES

The VIS-AD system provides a simple syntax for data type definitions. We offer examples from an algorithm for discriminating clouds in GOES images. The following are examples of how the user defines scalar types in S and the function $P: S \rightarrow PRIM$:

```
type brightness = real;
type temperature = real;
type earth_location = real2d;
type image_region = int;
type time = real;
type count = int;
```

Here *brightness* and *temperature* are the visible and infrared radiance values of pixels in satellite images, *earth_location* is a pair of values for the latitude and longitude of pixel locations, *image_region* is an index into rectangular sub-images, *time* is an index for image sequences, and *count* is used for histograms.

The following are examples of how the user defines complex types in T (the keyword *structure* is used to indicate the tuple constructor):

```
type visir_image =
  array [earth_location] of
    structure {
      .visir_ir = temperature;
      .visir_vis = brightness;
    };

type visir_set = array [image_region] of visir_image;

type visir_set_sequence = array [time] of visir_set;

type histogram = array [temperature] of count;
```


PRECEDING PAGE BLANK NOT FILMED

```
type histogram_set =
  array [image_region] of
    structure {
      .hist_location = earth_location;
      .hist_histogram = histogram;
    };
```

Data objects of type *visir_image* are two-dimensional images of *temperature* and *brightness* values, indexed by *earth_location* values. The cloud discrimination algorithm partitions images into regions, and a data object of type *visir_set* is an image with partitions indexed by *image_region* values. A data object of type *visir_set_sequence* is a *time* sequence of partitioned images. A *histogram* data object attaches a frequency *count* to a set of *temperature* values, and a *histogram_set* object contains a *histogram* and an *earth_location* value for each *image_region* value.

Type definitions can be used to attach ancillary values to data objects. For example, the following type defintions provide a way to attach a sensor name, a satellite sub-point, and a table of errors as a function of temperature, to each image in a time sequence:

```
type sensor_name = string;
type rms_error = real;

type visir_sequence =
  array [time] of
    structure {
      .vs_sensor = sensor_name;
      .vs_sub_point = earth_location;
      .vs_error = array [temperature] of rms_error;
      .vs_visir = visir_image;
    };
```

4. DATA OBJECTS

Define $D(t)$ as the set of data objects of a type $t \in T$, sometimes called the "domain" of a data type. The domains of scalar types are determined from the domains of their primitive types, by $D(s)=D(P(s))$. The domain of the primitive type *int* is the union of a set of finite sub-domains, each an interval of integers, as follows:

$$D(\mathrm{int}_{i,j}) = \{k | i \le k \le j\}$$

$$D(\mathrm{int}) = \{missing\} \cup \bigcup_{i \le j} D(\mathrm{int}_{i,j})$$

where i, j and k are integers and the *missing* value indicates the lack of information (the use of special "missing data" codes is common in remote sensing algorithms). The domain of the primitive type *real* is the union of a set of finite sub-domains, each a set of half-open intervals, as follows:

$$D(real_{f,i,j,n}) = \{f(k/2^n), f((k+1)/2^n)) | i \le k \le j\}$$

$$D(real) = \{missing\} \cup \bigcup_{f \in Fld} \bigcup_{i \le j, n \ge 0} D(real_{f,i,j,n})$$

where i, j, k and n are integers and *Fld* is a set of increasing continuous bijections from **R** (the set of real numbers) to **R**; the functions in *Fld* provide non-uniform sampling of *real* values. The domains $D(real2d)$, $D(real3d)$ and $D(string)$ are similarly defined as the unions of finite sub-domains.

$D((s \to t))$ is defined as the union of a set of function spaces, rather than as the single space of functions from $D(s)$ to $D(t)$, as follows:

$$D((s \to t)) = \{missing\} \cup \bigcup_{subs} (D(s_{subs}) \to D(t))$$

where *subs* ranges over the finite sub-domains of the scalar domain $D(s)$, and $(D(s_{subs}) \to D(t))$ denotes the set of all functions from the set $D(s_{subs})$ to the set $D(t)$. Every array object in $D((s \to t))$ contains a finite set of values from $D(t)$, indexed by values from one of the finite sub-domains of $D(s)$.

The domains of tuple types are defined by:

$$D((t_1,\ldots,t_n)) = \{missing\} \cup D(t_1) \times \ldots \times D(t_n)$$

Each domain $D(t)$ has a lattice structure, with the *missing* value as its least element. The half-open intervals in $D(real)$ are approximations to values in **R** and are ordered by the inverse of set inclusion; that is, in the lattice structure, an interval is "less" than its sub-intervals. Values in $D(real2d)$ and $D(real3d)$ form similar lattices and are approximations to values in $\mathbf{R}^2$ and $\mathbf{R}^3$. The lattice structure can be extended to array and tuple types.

The lattice structure of domains, and the definition of array domains as unions of function spaces, provide a formal basis for interpreting array data objects whose indices have primitive types *real*, *real2d* or *real3d* as finite samplings of functions over **R**, $\mathbf{R}^2$ or $\mathbf{R}^3$. For example, a satellite image is a finite sampling of a continuous radiance field. The VIS-AD programming language allows arrays to be indexed by *real*, *real2d* and *real3d* values. Navigation (earth alignment) and calibration (radiance normalization) for satellite images can be implemented by appropriately defined sub-domains of $D(real2d)$ and $D(real)$, so that raw satellite images can be accessed directly in terms of latitude, longitude and temperature.

Physical variables range over infinite sets of values, such as the set **R** of real numbers. However, values must be stored in computers using a finite number of bits, and thus are constrained to range over finite sets of values, such as the set of 32-bit floating point numbers. These are finite samplings of infinite value sets. In most programming languages, the finite samplings for variables

are determined by the type of a variable (for example, *real* or *double* in C). In the VIS-AD programming language, however, the finite samplings are specified as part of the data object. A scalar domain $D(s)$ is a union of a set of finite sub-domains, and each sub-domain is a different finite sampling of the infinite set that is the completion of the lattice $D(s)$ (for example, R is the completion of D(*real*)).

The pixel locations in a satellite image form a finite sampling of an infinite set of points on the earth. It is usual to store image data as arrays. Since arrays indices are identified as scalar types in VIS-AD, and since the finite sampling of the array index is part of an array data object, the navigation information for the satellite image is part of the image data object. Similarly the 256 radiance values of an 8-bit pixel are a finite sampling of temperatures, so the calibration information for a satellite image is also part of the image data object. Thus the domains for VIS-AD data objects provide a uniform way to manage these ancillary information as part of the data objects themselves.

One simple consequence of the VIS-AD data domain structure is support for variable length arrays. That is, the sizes of arrays are not fixed in their declarations, but may vary. Thus arrays can be used to model list structures. For example, a map boundary can be defined with the following data types:

```
type list_index = int;

type map_boundary =
  array [list_index] of earth_location;
```

A *map_boundary* data object is a variable length array of *earth_location* points. Thus, although VIS-AD does not explicitly support linked data structures, it can easily model simple list structures.

The VIS-AD support for *missing* data is motivated by its use for managing remote sensing data. However, *missing* data can also be used as a data structuring tool. For example, a data object *image_area* of type *visir_image* can be used to represent an arbitrarily shaped image region simple by setting

```
image_area[earth_location] = missing;
```

for all values of *earth_location* not in the image region.

Thus navigation, calibration and missing data indicators can be built into the values of data objects, variable length arrays can be used to model list structures, and missing data can be used as a data structuring tool. Combined with the flexibility of type definitions illustrated in Section 4, VIS-AD provides very powerful tools for managing earth science data.

5. ACCESS TO DATA OBJECTS

The VIS-AD programming language provides a transparent way to access the finite sampling information of data objects. For example, if *goes* is an object of type *visir_image* and *loc* is an object of type *earth_location*, then VIS-AD evaluates the expression *goes[loc]* as:

> if *loc* is outside the range of the finite sampling of index values of *goes*, then evaluate *goes[loc] = missing*
> otherwise, resample *loc* to the actual index value *loc'* of the *goes* array closest to *loc*, and evaluate *goes[loc] = goes[loc']*

Thus array accesses may evaluate to *missing*. VIS-AD provides a transparent way to manage operations on *missing* values. If OP is a binary arithmetical operator (+, -, * or /) and *val1* and *val2* are expressions with scalar values, then VIS-AD evaluates the expression *val1* OP *val2* to *missing*:

> if *val1* = *missing* or
> if *val2* = *missing* or
> if OP == / and *val2* == 0

These evaluation rules make it easy to combine data from different sources, with out the need to explicitly remap one set of data to the projection of the other. For example, let *goes_west* and *goes_east* be two data objects of type *visir_image*, from the west and east GOES satellites respectively. The pixels in these images are not co-located, but the difference of these images can be calculated quite simply by:

```
foreach (loc in goes_west) {
  goes_west[loc] = goes_west[loc] - goes_east[loc];
};
```

where *loc* is a data object of type *earth_location*. Inside the *foreach* loop, the value of *loc* varies over all the array index values of the array *goes_west*. The values of *goes_east* are resampled to the index locations of *goes_west*, and the difference of these image arrays evaluates to *missing* wherever they do not overlap.

VIS-AD also provides a simple means to access sub-objects of data objects. For example, if *hset* is a data object of type *histogram_set* and if *reg* is a data object of type *image_region*, then the expression *hset[reg].hist_histogram* evaluates to a data object of type *histogram*.

VIS-AD includes functions for converting objects between their internal storage formats and external formats suitable for storage in disk files and for transmission to other processes or across computer networks. Both the internal and external object formats use memory efficiently. Numerical values are stored as scaled integers rather than as floating point numbers, and use 8-bit or 16-bit integers wherever possible. These formats minimize use of disk storage and communications bandwidth. The absence of any floating point values eliminates the need for converting

data objects between machine architectures (except for possible problems with big-endian versus small-endian machines, and machines that use non-ascii text).

6. HIGH-LEVEL FUNCTIONS

VIS-AD supports calls to three kinds of functions. Internal functions are implemented in the VIS-AD programming language, and users writing VIS-AD programs are free to define as many internal functions as they need. Intrinsic functions are implemented as part of the VIS-AD system and should be viewed by users as part of the language (like the MAX function in FORTRAN). External functions are implemented in C or FORTRAN, and give users a way to link their existing programs to VIS-AD.

The VIS-AD system includes a variety of intrinsic functions for transferring McIDAS data structures (for example, image and grid files) into VIS-AD data objects, and for analyzing data. We are constantly adding new intrinsic functions to the system. Analysis functions are currently defined for:

- Remapping two- and three-dimensional images and grids.
- Low-pass filtering one-, two- and three-dimensional data.
- Calculating histograms of data arrays.
- Finding clusters in histograms.
- Finding percentiles of histograms.
- Selecting regions of arrays with values in selected ranges.
- Boolean operations on regions of arrays.

VIS-AD external functions, written by the user in C or FORTRAN, provide a way for users to access data sets stored in any format, and a way for users to link to their existing analysis functions.

7. CONCLUSION

The purpose of this paper is to point out that VIS-AD contains many powerful functions for managing earth science data. These include:

- An easy way for users to define new data structures, such as images, multi-spectral images, image sequences, histograms, spatial regions, region boundaries, etc. These flexible data types also let users define a variety of ancillary data as part of their data types.
- An easy way to access data objects and their sub-components a programming language, and an easy way to write functions for analyzing those data.
- Uniform mechanisms for management of all data structures.
- A special *missing* data indicator that can be used for the value of any data object or sub-object.
- A uniform mechanism for managing finite samplings of continuous quantities, as for example, the way that satellite navigation and calibration finitely sample earth locations and temperatures.
- A easy way for users to write C or FORTRAN programs for converting data between VIS-AD data objects and other systems.
- A mechanism for storing data objects in disk files or for transmitting data objects across networks.

Thus, the most difficult functions for managing complex earth science data already exist in VIS-AD. In order for VIS-AD to function as a true data management system, it needs to include functions for:

- Immediate mode commands. Currently, all functions are called by a running VIS-AD program. Users need a way to invoke functions one at a time by typing commands.
- Managing data objects in disk files. There should be commands for transfering data objects between disk files and memory, for listing objects in disk files, and possibly for retrieving sub-objects of objects stored in disk files (since data objects may be large).

After these functions are implemented, VIS-AD will be a powerful earth science data manager, in addition to its original role for visualizing the behavior of scientific algorithms.

8. REFERENCES

Hibbard, W., C. Dyer and B. Paul, 1992a; A development environment for data analysis algorithms. Preprints, Conf. Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology. Atlanta, American Meteorology Society. 101-107.

Hibbard, W., C. Dyer, and B. Paul, 1992b; Display of scientific data structures for algorithm visualization. Accepted for Visualization '92, IEEE.

N 15

A DEVELOPMENT ENVIRONMENT FOR DATA ANALYSIS ALGORITHMS

William Hibbard[1], Charles R. Dyer[2], and Brian Paul[1]

[1]Space Science and Engineering Center
[2]Computer Science Department
University of Wisconsin-Madison

1. INTRODUCTION

Data analysis algorithms encode much of our scientific understanding of the environment. They extract information from images and from other observations, and they produce diagnostic fields from the output of numerical simulations. A primary activity of the Space Science and Engineering Center has been the creation of algorithms for analyzing GOES (Geostationary Operational Environmental Satellite) images. These algorithms estimate winds from cloud motions, estimate cloud top height and thickness, classify cloud types, and estimate vertical temperature and moisture profiles from multi-spectral images. Analysis algorithms are also applied to a wide variety of non-satellite data, including two- and three-dimensional radar and lidar images, model output grids, irregularly located observations, and time sequences of all of these.

There is currently a great need and a great opportunity to create new visualization tools to help the developers of data analysis algorithms. There is a need due to the increasing volumes of environmental data, the difficulty of many data analysis problems, and the urgency of environmental understanding. There is opportunity because of the increasing power of visualization workstations, and because most environmental data are inherently geometrical and thus can be understood visually.

Two particularly interesting new tools for image processing are the Khoros system (Rasure, Argiro, Sauer and Williams, 1990) and the Interactive Image Spreadsheet (Hasler, Dodge and Woodward, 1991). The Khoros system allows its user to compose image processing algorithms as data flow diagrams whose nodes are basic image processing modules. The user may interactively execute the flow algorithm, and display any of its intermediate image products, in order to visualize the functioning of the algorithm. The Image Spreadsheet allows its user to build and display a matrix of images, where the images are raw satellite channels, or computed from combinations of other images by standard image processing functions. This allows the user to experiment with combinations of satellite channels and image processing operators to extract useful information.

We are developing the VIS-AD (VISualization for Algorithm Development) system to extend this experimentation with data analysis algorithms beyond image data. Even when the data being analyzed consists of images, many of the intermediate data structures are not images. For example, during the development of an algorithm for extracting clouds from GOES images, we utilized ad hoc graphics of multiple intermediate images, image section boundaries, cloud boundaries, scatter diagrams, histograms, and histogram clusters, all in a common frame of reference (Hibbard, 1987). These graphics helped in understanding the relation between these data structures and therefore the functioning of the cloud extraction algorithm.

2. INTERACTIVE DEBUGGING PLUS VISUALIZATION

From a programmer's point of view, VIS-AD is an interactive debugger that provides visualization of program data structures instead of just printing the values of variables. Data Analysis algorithms are expressed in an ordinary programming language which is similar to C with high level data structures. The VIS-AD user edits an analysis program, interactively executes it, and controls the display of its data structures. Interactive debuggers greatly increase the productivity of programmers, but they have not been appropriate for most scientific algorithms because they can only print numerical values of variables. VIS-AD will make this productivity tool available to scientists by including visualization of high level data structures.

The central focus of VIS-AD is a programming window where the user edits programs and controls execution and display. The window provides an interactive screen editor for program text, including saving and retrieving programs. The user controls execution with the mouse, selecting and highlighting program lines as breakpoints, toggling a button to start and stop program execution, and hitting a "single step" button to execute one program step. The user controls data display by selecting and highlighting program variables for display.

PRECEDING PAGE BLANK NOT FILMED

VIS-AD also includes one or more display windows, where program variables are depicted as graphics and images. Each window can display two- and three-dimensional color graphics. The window also supplies widgets for user interaction with the window contents, and mouse controls for interactive pan, zoom, and rotation of the 3-D window contents. Each display window defines a different frame of reference for the display of an algorithm's data structures. The user defines a frame of reference by a set of mappings from the physical scalar quantities of the algorithm to the coordinates of the display, as described in Section 4. These mappings are edited in a text window attached to the display window.

In a typical interactive scenario the user would:

1) compose an algorithm in the program window, or retrieve a saved algorithm
2) create one or more display windows, each defining a frame of reference for the algorithm's data structures
3) select an input data sets, and interactively execute the algorithm, either one step at a time, or by setting breakpoints
4) during execution, select various combinations of the algorithm's data structures for display in the display windows
5) alter the algorithm based on the visual understanding of its actions
6) repeat the steps above during a development session
7) save the developed algorithm.

3. DATA TYPES

The key to VIS-AD is the fact that every algorithm data structure can be displayed. We have developed a technique for automating the display of scientific data structures (Hibbard and Dyer, 1991), in terms of a data type definition language and a way of defining a frame of reference by a set of mappings. We have made this data type definition language part of the VIS-AD programming language.

VIS-AD allows scalar types to be defined as integers, as one-, two- and three-dimensional real numbers, as strings, and as list indices. For example, the following are VIS-AD scalar type definitions:

```
type time = float;
type temperature = float;
type earth_location = float2d;
type wind_velocity = float3d;
type data_set_name = string;
type count = int;
type line_list_index = list;
```

Here *time* and *temperature* take real values, *earth_location* takes a pair of real values for latitude and longitude, and *wind_velocity* takes a triple of real values for wind components. *Data_set_name* takes a text string value, and *count* and *line_list_index* take integer values. *List* and *int* scalars are very similar, but *list* scalars generally have no physical meaning and their use as an array index indicates that the array is used as a list.

VIS-AD also allows complex data structures to be built up as arrays and structures from simpler types. The following are VIS-AD type definitions for complex data structures:

```
type image =
  array [earth_location] of temperature;
type image_sequence =
  array [time] of image;
type border =
  array [line_list_index] of
    structure {
      .first_end_point = earth_location;
      .second_end_point = earth_location;
    };
type histogram =
  array [temperature] of count;
```

Here an *image* is an array of *temperature*'s indexed by a set of *earth_location*'s, an *image_sequence* is an array of *image*'s indexed by *time*, a *border* is an array of *earth_location* pairs (the endpoints of line segments) indexed as a list, and a *histogram* is an array of *count*'s indexed by *temperature*.

Arrays may be defined as any scalar or complex type indexed by any scalar type, with the index scalar specified inside brackets. Structures may be defined as any fixed list of scalar or complex types, each identified by a structure element name (for example, first_end_point or second_end_point in the *border* type).

The definition of complex data types in terms of scalars provides the information necessary to display them in a common frame of reference. For example, the *image* and *border* types are both defined in terms of *earth_location*, which makes it possible to overlay them geographically in the display.

VIS-AD array declarations do not specify the number and range of index values of the array. Instead, this information is part of an array data object, so that different array objects of the same type may cover different numbers and ranges of index values. Consider an *image* array. The *image* pixel locations are a finite sampling of *earth_location*'s, and the pixel values are a finite sampling of *temperatures*. The interpretation of the VIS-AD *image* type is that the finite samplings of *earth_location* and *temperature* are specified as part of each data object of type *image*, rather than being specified by the type definition. This is different than most programming languages, in which the range of values of variables and the sizes of arrays are specified in the declarations of data objects. For the *image* type, the finite sampling of *earth_location* determines the number of pixels in an *image* array, and the finite sampling of *temperature* determines the number of values each pixel may take (and thus the necessary number of bits per pixel), and these may vary between *image* data objects. Furthermore, the finite sampling of *earth_location* provides the *image* navigation (mapping between earth location and pixel

location in an image), and the finite sampling of *temperature* provides the *image* calibration (mapping between temperature and pixel value). Thus the *image* type is actually a high level data type, corresponding to a McIDAS image file "area" with its attached directory and codicil information for navigation and calibration.

VIS-AD type definitions provide a mechanism for specifying when the finite sampling of values varies over an array index and when it does not. For example, the sampling of *temperature* may not vary over the *earth_location*'s in an *image*, but the finite samplings of both *temperature* and *earth_location* may vary over *time* in an *image_sequence*. The mechanism for controlling the variation of finite samplings is used by VIS-AD primarily to increase storage and computational efficiency, and we will not describe it in detail here.

Any VIS-AD data object may take the special value MISSING, indicating the lack of information. This applies to scalar, structure and array types, and to any sub-object within a named data object. All operations are defined for MISSING input values, usually producing a MISSING output. Our experience with science data has shown the need to accommodate missing data values in all of our data analyses, so we have built it into the VIS-AD language. Missing data sometimes indicate an error condition, but are also often used intentionally to indicate partial data coverage. Because MISSING data handling is built into VIS-AD, including efficient internal data representations for sparse arrays, MISSING data can be used as a data structuring technique.

VIS-AD will include interfaces to the McIDAS data structures for image areas, grids and MD (Meteorological Data) files. These interfaces will consist of intrinsic functions in VIS-AD for converting between the McIDAS data structures and a set of corresponding VIS-AD data types. VIS-AD algorithms will also be able to invoke McIDAS analysis commands and system level commands.

4. DISPLAY WINDOWS

A VIS-AD display window is defined as a three-dimensional array of voxels, with values for colors at those voxels. The color values may be either true color or pseudo-color. Contour values may also be attached to voxels, represented as iso-level contour surfaces in three dimensions or as iso-level contour lines on two-dimensional slices. A display window also includes a temporal coordinate for animation, and widgets which can be used to select ranges of values.

Voxel coordinates are defined as the following implicit scalar types:

```
type x_coordinate = float;
type y_coordinate = float;
type z_coordinate = float;
type xy_plane = float2d;
type xz_plane = float2d;
type yz_plane = float2d;
type xyz_volume = float3d;
```

Voxel values are defined as the following implicit scalar types:

```
type color = float3d;
type red = float;
type green = float;
type blue = float;
type pseudo-color = float;
type contour = float;
```

The temporal coordinate for animation is defined as the following implicit scalar type:

```
type animation = int;
```

The implicit type *widget* refers to any widget for selecting ranges of values, and may assume any scalar type, determined by its mapping from an application scalar.

The user defines a frame of reference for a display window by a set of mappings from application scalars (those scalar types defined in the user's program) to display scalars. For example, given the application types defined in Section 2, we can define a frame of reference by:

```
map earth_location to xy_plane;
map temperature to pseudo_color;
map time to animation;
```

This creates a frame of reference in which the *image* type is displayed as a two-dimensional array of *pseudo-color*'s, the *border* type is displayed as a series of line segments over the *image*, and the *image_sequence* type is displayed as an animated sequence of *image*'s.

Adding:

```
map line_list_index to widget;
```

to the frame of reference will cause a widget to appear for selecting a range of values for *line_list_index*. Display of the *image* type will not be changed, but only subsets of *border* data objects will be displayed, confined to those line segments in the *border* array indexed by the selected values of *line_list_index*.

If the mapping of *temperature* is modified to map temperature to contour; then an *image* data object will be displayed as a set of iso-level contour lines of *temperature*'s in the *xy_plane*.

If the mapping of *temperature* is modified, and a mapping for *count* is added, as follows:

```
map temperature to z_coordinate;
map count to x_coordinate;
```

then an *image* data object will be displayed as a three-dimensional surface showing z as a function of x and y, and a *histogram* data object will be displayed as a graph in the *xz_plane*.

A frame of reference is a set of mappings from application scalars to display scalars, with each application scalar mapped at most once. *Float2d* and *float3d* scalars may only be mapped to display scalars of the same dimension (that is,

to *..plane*, *xyz_volume* or *color*), and *string* scalars may only be mapped to *widget*. As part of a scalar mapping, the user may specify a mapping function that controls the mapping of numerical application scalar values to display scalar values. The mapping examples above all invoke a default mapping function.

The user may define several display windows, each defining a different frame of reference. Different data types may best be understood in different frames of reference, and this is particularly true when the user needs to understand the relations between multiple data types. Thus it is important to allow the user to define multiple frames of reference.

The user has a variety of controls for a display window, including:

1) an attached text window for defining scalar mappings
2) widgets for selecting ranges of values for scalars mapped to *widget*
3) if *pseudo_color* is used, widgets for defining the mapping from the numerical values of the mapped application scalars to red, green and blue
4) if *contour* is used, widgets for defining the contour levels, intervals, and 2-D slice locations
5) widgets for controlling animation - animation is defined as time sequencing of the values of application scalars mapped to animation
6) mouse controls for panning, zooming, and rotating the projection from the three-dimensional array of voxels onto the two-dimensional display screen.

Quantitative information is important, so a display window contains numerical labels for application scalars, including

1) numerical scales along the x, y and z axes for scalars mapped to those axes
2) numerical and text labels for selected values of scalars mapped to *widget*
3) numerical scales for scalars mapped to *pseudo_color*, labelling the color selection widgets
4) numerical labels on contours for scalars mapped to *contour*, and labels for locations of 2-D slices
5) numerical labels on each frame of an animation sequence, for scalars mapped to *animation*.

5. HIGH LEVEL PROGRAMMING LANGUAGE

Users of VIS-AD express their algorithms in a language similar to C. The following is an example of a simple image processing algorithm using the data types defined in Section 2.

SAMPLE PROGRAM

```
/* main is the top level function of the algorithm */
main()
{
  /* these statements declare data objects with the types
     defined in section 2 */
  image goes_ir, goes_diff;
  border goes_border;

  /* call an intrinsic function to read a GOES infrared image
     from McIDAS area number 10 */
  goes_ir = read_mcidas_area(10);
  /* call an internal function to compute a difference operator */
  goes_diff = image_diff(goes_ir);
  /* call an external function which defines an edge finding operator */
  goes_border = edge_finder(goes_diff);
  /* call an internal function to remove null border segments */
  remove_null(goes_border);
}

/* this defines an internal function image_diff which takes one
   argument of type image and returns a value of type image */
image image_diff(image ir;)
{
  /* these statements declare data objects with the types
     defined in section */
  image diff;
  earth_location loc, lat_offset, lon_offset;
```

```
  /* set latitude and longitude offsets for a difference operator */
  lat_offset = (0.1, 0.0);
  lon_offset = (0.0, 0.1);
  /* initialize the diff image to the values of the ir image */
  diff = ir;
  /* for each earth_location sampled by the ir image */
  for each (loc in ir) {
    /* calculate a difference operator */
    diff[loc] = diff[loc] -
     (ir[loc - lat_offset] + ir[loc + lat_offset] +
      ir[loc - lon_offset] + ir[loc + lon_offset]) / 4;
  }
  /* return the diff image */
  return diff;
}

/* this defines an untyped internal function remove_null which takes
   one argument of type border - this function does not return a value
   argument, but alters its argument b */
remove_null(border b;)
{
  /* this statement declares a data object with a type
     defined in section */
  line_list_index ind;

  /* any border segment with zero length is replaced by MISSING */
  for each (ind in b) {
    if (b[ind].first_end_point == b[ind].second_end_point) {
      b[ind] = MISSING;
    }
  }
}
```

In this simple algorithm, *read_mcidas_area* is an intrinsic function that converts a McIDAS image file (called an area) into a VIS-AD *image* array. *Edge_finder* is a image processing operator defined by an external function. VIS-AD includes a library of compiled external functions implementing a variety of useful operators, and users can implement new external functions in either C or Fortran. *Image_diff* is an image processing operator defined by an internal function. Note that its formal parameter *ir* is listed with its type, in the style of ANSI C function declarations. *Remove_null* is a simple operator defined by an internal function. It illustrates that VIS-AD functions are called by reference, and may modify the values of their parameters.

In addition to the control structures *for each* and *if* illustrated in this simple example, the language includes statements for *if ... else*, *while*, and *break* (immediate exit from a *for each* or *while* loop). The language also allows any sub-object of an object to be accessed or assigned to, and complex nested expressions and logical conditions.

In VIS-AD all data objects must be declared as scalar or complex types defined in VIS-AD, rather than as *int* or *float*. This constrains every data object to be defined in terms of scalars that may be mapped to display scalars, and thus every data object may be displayed.

In the *image_diff* function, the *ir* array is accessed using expressions such as *ir[loc - lat_offset]*. In the likely case that the value *[loc - lat_offset]* is not one of the *earth_location* samples of the *ir* array, the value of the expression is the *ir temperature* value whose index is nearest to *[loc - lat_offset]*. The value of the expression is MISSING if *[loc - lat_offset]* is not near any sample of the *ir* array. As part of our VIS-AD development, we will experiment with array access that implicitly interpolates between array indices.

Because the range of values of arrays is specified as part of the data objects, VIS-AD will require a rich set of intrinsic functions to access the finite sample information of data objects for calculations that need to explicitly access array elements. On the other hand, many operations can be implemented more simply by exploiting the fact that data objects can be defined and accessed in physically meaningful terms, including implicit resampling and interpolation of array values.

Recently there has been great interest in visual programming languages, which express algorithms as data flow diagrams composed of basic modules. Examples include the Khoros system, AVS (Upson et. al., 1989), apE (Dyer, 1990), SGI's Explorer, IBM's Data Explorer, JPL Explorer, NCSA's DICE, and Iconicon's IDF. We have not adopted a visual programming language for VIS-AD for several reasons:

1) The goal of VIS-AD users is to produce algorithms, so we assume they know how to write programs.
2) A procedural programming model is more expressive than a data flow programming language, for example in partial updating of large data structures, and in controlling the order of execution of functions.
3) A key benefit of visual programming is the ability to interactively modify and execute an algorithm, but this is also possible with an interpreted procedural language.
4) For complex algorithms, a procedural implementation is probably easier to understand than a data flow implementation.

A key to the usability of any high level programming system is the availability of a rich set of high level operators. This has been the experience with the visual programming systems, and will also apply to VIS-AD. However, it is also important to allow the user to mix low level data types and operators with the high level operators. It is common for users of high level systems to be frustrated by their inability to store a little state information in an integer. Low level data types and operators also allow a user to quickly compose a new high level operator, without leaving the context of the high level programming system. Once the new operator has shown its value, it can be implemented more efficiently as a compiled external function. Because VIS-AD data types are user definable at both low and high levels, VIS-AD clearly supports a mixture of low and high level operators.

6. VISUALLY EXPLORING ALGORITHMS

VIS-AD will support high level functions for visualizing algorithm behavior, allowing the user to explore traces of data objects, and the way algorithm behavior varies with respect to changing input data sets and changing versions of the algorithm. These functions will be supported by defining array types which index values for the algorithm's data objects to be visualized.

Traces of algorithm data objects are based on the implicit scalar

```
type trace = list;
```

which indexes sequences of values of data objects.

The variations of data objects and traces with respect to changing input data sets are based on the implicit scalar

```
type data_set_name = string;
```

which indexes arrays of values of data objects or traces produced by applying the algorithm to multiple input data sets.

The variations of data objects and traces with respect to changing versions of an algorithm are based on the implicit scalar

```
type algorithm_name = string;
```

and also on selected algorithm scalar objects used as algorithm parameters. *Algorithm_name* indexes arrays of values of data objects and traces produced by multiple versions of an algorithm, where the text string values of *algorithm_name* are the names of the algorithm source files. Of course, any data object selected for indexing via *algorithm_name* must be declared in all versions of the algorithm. The selected scalar parameters index arrays of values of data objects and traces produced by multiple versions of an algorithm, parameterized by varying initial values of the selected scalar data objects.

The user controls these functions by
1) selecting data objects for trace, data set variation, and algorithm variation
2) selecting scalar data objects as algorithm parameters
3) defining a set of algorithm source files for multiple source versions
4) inserting trace statements into an algorithm to define execution points for saving trace values
5) selecting analysis data objects for display.

The scalars *trace*, *data_set_name* and *algorithm_name* may be mapped to display scalars, so that variations in algorithm behavior may be understood in multiple and flexible frames of reference. Visually exploring trace data objects, and data objects indexed by input data set name and by algorithm version, will be a powerful way to understand patterns in algorithm behavior.

7. CONCLUSION

VIS-AD extends the power of interactive debugging to high level data analysis algorithms through the use of visualization. VIS-AD can also be seen as a way of adding interactive data analysis functions to our VIS-5D system (Hibbard and Santek, 1990). Visualization of algorithms is a very difficult problem, and probably also the most important problem in visualization. It is likely that we will be improving VIS-AD over a long period of time, but we also anticipate that VIS-AD will significantly improve the productivity of scientists developing data analysis algorithms.

Just as VIS-5D was enabled by a new generation of high performance graphics workstations, VIS-AD will require even higher levels of performance in order to transform high level data structures into graphics at interactive speeds. We believe that appropriate hardware will be commonly available at about the time that VIS-AD matures as a useful software system.

8. ACKNOWLEDGEMENTS

We wish to thank Dr. James Dodge and Dr. Greg Wilson of the National Aeronautics and Space Administration (NASA) for their encouragement and support. Funding for this work was received from the NASA Marshall Space Flight Center (NAG8-828).

9. REFERENCES

Dyer, D., 1990; A dataflow toolkit for visualization; Computer Graphics and Applications, 10(4), 60-69.

Hasler, A., J, Dodge, and R. Woodward, 1991; A high performance interactive image spreadsheet. Preprints, Conf. Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology. New Orleans, American Meteorology Society, 187-194.

Hibbard, W., 1987; 4-D Display of satellite cloud images. Digital Image Proc. and Visual Comm. Technologies in Meteor., Cambridge, SPIE, 83-85.

Hibbard, W., and D. Santek, 1990; The VIS-5D system for easy interactive visualization. Visualization '90, San Francisco, IEEE. 28-35. Also in 1991 Preprints, Conf. Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology. New Orleans, American Meteorology Society. 129-134.

Hibbard, W., and C. Dyer, 1991; Automated display of geometric data types. Univ. of Wisc. Comp. Sci. Dept. Tech. Report #1015.

Rasure, J., D. Argiro, T. Sauer, and C. Williams, 1990; A visual language and software development environment for image processing; International J. of Imaging Systems and Technology, Vol. 2, 183-199.

Upson, C., T. Faulhaber, Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, A. van Dam, 1989; The application visualization system: a computational environment for scientific visualization; Computer Graphics and Applications, 9(4), 30-42.

omit

DESIGN FOR AND EXPERIENCE WITH THE McIDAS GOES INVENTORY

William Hibbard, William Lagerroos, Delores Wade and Nancy Troxel-Hoehn

Space Science and Engineering Center
University of Wisconsin - Madison

1. INTRODUCTION

The Space Science and Engineering Center (SSEC) has been archiving data from the Geostationary Operational Environmental Satellites (GOES) on videocassettes (Suomi, 1982) since 1978. In 1985 we began operating an on-line inventory of the GOES archive (Hibbard, 1986), to replace our previous paper inventory of archived images.

2. THE INVENTORY DESIGN

The inventory design was based on three basic ideas:

1. Generality of inventory information formats. New inventory record formats may be defined as needed, so the inventory can be adapted to changing data source operations and to new data sources. This also makes it possible for one inventory to be applied to multiple data sources, allowing archive searches based on coincidence between data sources.
2. Inventory by exception from a schedule. The operators define a schedule as a list of times of day, each associated with an "activity" record that describes an item (for example, an item is an image for the GOES satellite) scheduled to be produced by the source at that time. A set of schedules are defined for a data source (for example, these may be named NOMVAS92-6 and RISOP92-5 for GOES operations). The inventory documents the operations of a data source as a set of date/time intervals, each identified with a schedule name. "Comment" records in the inventory specify deviations of the actual archived items from the scheduled items. The inventory by exception idea was adopted from the paper inventory. It reduces the volume of inventory records.
3. Grouping of inventory information for secondary storage. Sets of inventory records are accessed from secondary storage as a unit. For example, for the GOES archive a unit consists of all records for one satellite for one year - this is possible because of the compression effect of the inventory by exception. These large units simplify the organization of disk storage, increasing inventory integrity. They also increase the efficiency of archive searches, particularly for searches that specify relations between items from different sources or at different times (for example, searching for a sequence of six items spaced at half hour intervals).

This design can be illustarted with a few simple examples. We start with two examples of activity record formats. We give the name of the activity, followed by a text description, then a list of field names and their formats:

MSI3B:
"a Multi-Spectral Imaging mode"
SCANS: 2 values of 2 bytes each
BANDS: 8 values of 1 byte each
DETECT: 6 values of 2 bytes each
PDLNUM: 1 value of 2 bytes

DS12B:
"a Dwell Sounding mode"
SCANS: 2 values of 2 bytes each
SPINS: 12 values of 1 byte each
DETECT: 12 values of 2 bytes each
PDLNUM: 1 value of 2 bytes

We also offer examples of comment record formats:

C-OTHER:
"a substituted image type"
SCANS: 2 values of 2 bytes each
PDLNUM: 1 value of 2 bytes

M-SAT:
"Missing data - SATellite problems"
BADSCANS: 2 values of 2 bytes each

M-GSEQ:
"Missing data - Ground Station EQuipment problems"
BADSCANS: 2 values of 2 bytes each

P-SAT:
"Poor data - SATellite problems"
BADSCANS: 2 values of 2 bytes each

P-GSEQ:
"Poor data - Ground Station EQuipment problems"
BADSCANS: 2 values of 2 bytes each

PRECEDING PAGE BLANK NOT FILMED

Schedules are defined as lists of times-of-day associated with activity records, and we offer examples of parts of two schedules, RISOP92-5 and NOMVAS92-6:

```
RISOP92-5:
    0:01 MSI3B (SCANS=101, 1701;
                BANDS = 8, 7, 8, 7, 8, 10, 8, 10;
                DETECT = 8, S, 7, L, 10, L;
                PDLNUM = 707)
    0:20 DS12B (SCANS=234, 404; etc)
    0:31 MSI3B (SCANS=101, 901; etc)
    0:41 MSI2B (SCANS=191, 531; etc)
    0:46 MSI3B (SCANS=191, 531; etc)
    0:51 MSI3B (SCANS=191, 531; etc)
    0:56 MSI3B (SCANS=191, 531; etc)
        .
        .
        .

NOMVAS92-6:
    0:01 MSI3B (SCANS=101,1701; etc)
        .
        .
        .
```

The inventory is organized by days, and associates a schedule with time intervals during the day. It also associates archive tape numbers with time intervals, as seen in the following excerpt:

```
    .
    .
    .

DAY = 92131
    0:00z - 2:00z  schedule: RISOP92-5
        0:20z M-SAT (BADSCANS = 234, 404)
        0:31z P-SAT (BADSCANS = 101, 200)

    2:00z - 23:30z  schedule: NOMVAS92-6
        4:01z C-OTHER (SCANS = 1, 1201;
                       PDLNUM = 112)

    TAPES:
        0:00z - 10:00z   17921302
        10:30 - 23:30z   17921311

DAY = 92132
    .
    .
    .
```

In the current inventory organization, all of the days for 1992 would be accessed as a unit from a disk file. However, if there is a need for more inventory information per day (for example, to include an explicit quality assessment for every image in the archive), the number of days in a storage unit could be decreased.

3. OPERATIONAL EXPERIENCE

We have had about seven years of operating experience with the on-line GOES inventory. The inventory has:

1. Been successful at documenting the contents of the GOES archive, including adapting to a wide variety of operational schedules and a wide variety of contingencies for missing data or poor data quality.
2. Provided a high degree of integrity for the inventory data. Over a seven year period the inventory system has faced numerous system crashes, and during the first year of its operations the inventory software contained numerous bugs. In spite of these problems, the integrity of inventory files was maintained,
3. Not been used for complex searches. This has been the primary failing of the original inventory design. We believe it is due to a poor user interface and a user preference for manually searching printed listings of the inventory.

4. ADAPTING THE INVENTORY

We have adapted the McIDAS inventory for the GOES AAA and the anticipated GVAR (GOES VARiable) data formats. The inventory has been fairly successful at adapting to these new formats. The adaptation process has highlighted a couple of places where more flexibility of data formats would be useful. We are planning to adapt our inventory to METEOSAT and other new sources. We are also using it to document major weather events (such as hurricanes) to be used in search conditions.

5. USER INTERFACE LESSONS

One very important lesson has been the need to structure the user interface in terms of the users' tasks rather than in terms of the inventory data structures. For example, we have added functions for merging the scheduled "activity" records with the "comment" records that specify deviations of the actual archive contents from the schedule. Thus the information is presented to the user in terms of actual archive contents.

We have also greatly increased the flexibility of the way that the inventory information is presented to the user, so that users can ask for whatever level of detail is appropriate for their task.

In the original inventory design, the user interface was constrained by the McIDAS user interface. All user input had to be in terms of McIDAS command lines, which consist of a command name followed by a list of parameter values. Commands could not accept free

format text input. In order to incorporate graphics into the user interface, the inventory would have been restricted to the "McIDAS tower" workstations. Thus we adopted a text-only user interface so that the inventory could be used at simple terminals.

However, the inventory could really benefit from a graphical user interface (GUI). With the availability of inexpensive graphics workstations and McIDAS-X designed to run on these workstations, it will be natural to give the inventory a GUI. Some examples of graphical interfaces for the inventory could include:

1. Show geographical coverage on a map outline display, and allow users to select data coverage by drawing over a map outline.
2. Show inventory items along a time line, coding information about items by color, shape or embedded text. Multiple data sources may be shown as several parallel time lines.
3. Show statistics of archive data quality as functions of time, data source, schedule type, etc.
4. Prompt for user specification of criteria for inventory search.

6. DATA FORMAT LESSONS

The original inventory design emphasized flexible data formats, and the primary lesson has been the success of this flexibility. In fact, even more flexibility would be useful. For example, the data in "activity" or "comment" record fields were originally allowed to be 1, 2 or 4-byte integers, or text strings. We have added floating point fields, and coded text fields (that is, fields that may contain one of a small set of different strings, rather than free user text).

The inventory is largely table driven, and the tables fall into two basic types:

1. Tables with fixed contents, implmented in data statements embedded in the inventory source programs.
2. Tables with variable contents, implemented in files. The inventory software includes user commands for updating these tables.

There has been a desire to change some of the fixed tables to variable tables. For example, the inventory is distributed over several physical files, and there has been a desire to segregate inventory information for new data sources into their own files. The names of files are contained in fixed tables, and it would be convenient if these were in variable tables. Segregating new sources into separate files makes it possible for the same "activity" and "comment" names to be associated with different record formats for different data sources.

The current structure divides inventory information into "activity" records, whose field values are constant over many items, and "comment" records, whose field values are different for each item. It would be useful to give the operators ways to link "comment" records to "activity" records, so that logically it would appear that an "activity" record had some constant fields and some variable fields. This could be addressed purely by user interface changes, but it may be better to actually link certain fields of "activity" records to fields of "comment" records.

The inventory implements an approximate navigation function for GOES images, where the mapping from image line/element to earth latitude/longitude depends only on the longitude of a satellite's sub-point. This approximate navigation is used for rough calculations of data coverage. Originally the information for generating this approximate navigation was contained in simple tables. However, this information is now contained in a format that is identical to the detailed GOES navigation, so that it can be used to drive McIDAS graphics programs.

7. INVENTORY SEARCH

Traditionally, scientists have used the GOES archive by extracting small data sets from it. This pattern has largely been dictated by technical and financial limits on the size of data sets that can be analyzed. Because of the limits on data set size, scientists have tried to carefully select their data sets. Thus, in our original inventory design, we stressed the importance of efficiently supporting complex criteria for searching the inventory.

A typical request for an archive data set may ask for:

1. Data on three consecutive days in April or May of 1983.
2. On each day, data from images spaced one hour apart, and from the same hours on each day.
3. Data in a rectangle over the U.S. East coast, covering the same region in each image.
4. No poor quality images.

This search specifies a relation between data at different times. Other typical searches may be based on relations between data from different sources (for example, between GOES data and surface observations). The inventory organization was designed to optimize the efficiency of such searches, using relatively few accesses to secondary storage.

8. CONCLUSION

The McIDAS GOES inventory documents the contents of SSEC's 120 Terabyte GOES archive, and has been operating successfully since 1985. The archive has adapted to changes in satellite operations, and to the new GOES AAA and GVAR data formats.

The least successful part of the inventory has been its user interface, but this is improving with a better understanding of the archive operators' tasks. We also believe that the archive could benefit from a graphical user interface under McIDAS-X.

The way scientists use the archive is changing with the development of the GOES Pathfinder system, and this will cause some evolution in the role of the archive inventory system.

The most important lesson of our experience with the inventory is the success of its flexible data formats, that have allowed the inventory to adapt to changing and unpredictable needs.

9. REFERENCES

Hibbard, W. and G. Dengel, 1986: The GOES catalog on McIDAS. Preprints, Conf. Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology. Miami, Amer. Meteor. Soc, 98-100.

Hibbard, W., 1992; A highly parallel approach to satellite archive processing. Preprints, Conf. Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology. Atlanta, Amer. Meteor. Soc, 82-83.

Suomi, E., 1982: The Videocassette GOES Archive System - 21 Billion Bits on a Videocassette. IEEE Trans. on Geoscience and Remote Sensing, GE-21(1), 119-121.