THE UNIVERSITY OF HULL

DATA INPUT FOR SCIENTIFIC VISUALIZATION

being a Thesis submitted for the Degree of

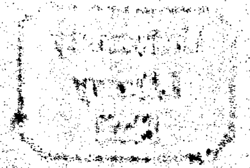
Doctor of Philosophy

in the University of Hull

by

Christian Mathers, BSc (hons)

August 2004



I could not have done this
without my parents support.
Thankyou Mum and Dad.

Abstract

Since the development of the modular visualization environment, the users of such general software have had to face the problems of file input. Simply put, the range and complexity of different file formats has prevented the developers of visualization systems from creating an individual solution for every format. This has left a gap, where users are left to fend for themselves by either extending the system to their needs, or using a format capable of being described by one of the input tools offered by such systems. Neither of these options is particularly easy, and the use of field dependent terminology can hamper such efforts.

This thesis proposes a model, architecture and methodology, for importing uncommon file formats and data into scientific visualization systems by way of interpretation. Using interpretation we are able to describe many file formats in a general manner, enabling further development of simple methods to aid users in solving their data input problems. The utility of these concepts is illustrated through the Interactive File Input Toolkit (IFIT), which allows users to solve their file input problems in a flexible manner. This tool is illustrated by a range of examples and test cases, and unlike other solutions it has the ability to discover as well as describe the content of a file. Finally, this thesis presents work towards an automatic method for determining a file's input parameters.

Acknowledgements

I would like to thank my supervisor Dr. Helen Wright for all her help and guidance throughout my time at the University of Hull. She is a person of endless patience, encouragement, humour and wit; qualities which were most certainly needed during the writing of this thesis.

This work has been supported by the EPSRC and the Numerical Algorithms Group Ltd under the CASE Studentship scheme. I would like to thank NAG for my funding and in particular, thank my industrial supervisor Jeremy Walton for his support, test cases and useful discussions during this work.

I would like to thank a number of other people who contributed collections of test data to this project including Dr John Whelan, Paul Chapman, James Ward, Dr Bill Hutchinson, Tim Dunstan, Jon Thorpe and David Burridge.

I would also like to thank those whom I have had useful discussion with including Prof. Roger Phillips, Derek Wills and James Ward at the Department of Computer Science at the University of Hull and Joanna Leng and James S. Perrin from the University of Manchester.

Many thanks are due to Peter Wilson for helping me to split not my infinitives. Equally many thanks are due to Helen Bristow for her literary help and support.

I would also like to thank Elizabeth for her endless patience, support and kind words of encouragement during the trying times. Finally, I would like to thank my parents for

supporting me through the hard (and mostly lean) times, and encouraging me throughout my life to expand my knowledge at every possible opportunity.

Publications

C. Mathers, H. Wright and J. Walton. "A Visual Programming Approach to Importing Data into Modular Visualization Environments", Submitted to IEEE Computer Graphics and Applications, 2004.

Table of Contents

Table of Contents v

1 Introduction 1

1.1 Visualization 3

1.2 Historical visualization 5

1.2.1 Graphs 5

1.2.2 Maps 6

1.2.3 3D Models 6

1.3 Modern computing period 7

1.4 Data input for scientific visualization 8

1.5 Inputting application data 9

1.6 Scope and goals for a file input system 10

1.6.1 Research problem 10

1.6.2 Scope of research 12

1.7 Thesis structure 13

2 Visualization models, file formats and input tools 15

2.1 Scientific data storage models 16

2.1.1 Lattices 16

2.1.2 Fibre bundles 17

2.1.3 Classification of scientific data with the E notation 18

2.2 Process models for visualization systems 20

2.2.1 Dataflow 20
2.2.2 A model centred approach 22
2.2.3 Osland's visualization reference model 24
2.2.4 The Visualization Input Pipeline 25
2.2.5 Visualization reference model 27
2.2.6 Discussion 28
2.3 Scientific file formats 30
2.3.1 Field-specific file formats 31
2.3.2 Language-based file formats 33
2.3.3 Self describing 'generic' file formats 35
2.3.4 User-defined and non-standard file formats 36
2.4 Visualization tools 38
2.4.1 AVS 39
2.4.2 IRIS Explorer 40
2.4.3 IBM Data eXplorer 41
2.4.4 Amira Viz 42
2.4.5 Khoros 42
2.4.6 PV-WAVE 43
2.4.7 MVE file input summary 43
2.5 Discussion 47

3 Data input considerations **48**
3.1 Understanding file storage 48
3.1.1 Interpretation is everything 50
3.1.2 Binary interpretations 51
3.1.3 Text interpretations 53
3.1.4 Structural interpretations 56
3.1.5 Semantic interpretations 66
3.2 The role of user knowledge in data input 70

3.3 Summary 72

4 A new approach to file input **74**

4.1 Approach principle 74

4.2 The file input dataflow model 76

4.3 A software architecture for file input 77

4.3.1 The value interpretation stage 77

4.3.2 The structural interpretation stage 79

4.3.3 The semantic interpretation stage 81

4.3.4 File input parameters 82

4.4 Summary 82

5 Using the Interactive File Input Toolkit (IFIT) **84**

5.1 Forensic file examination 84

5.2 Final requirements for a file input tool 86

5.2.1 User requirements 86

5.2.2 Output requirements 87

5.2.3 Functional requirements 88

5.2.4 Implementation requirements 89

5.3 An overview of IFIT 90

5.3.1 Transformation of user data 90

5.3.2 Specification modules 91

5.3.3 Visual feedback modules 92

5.4 Using the transformation and specification facilities of IFIT 94

5.5 Using visual feedback 106

5.5.1 TextView 106

5.5.2 ImageView 108

5.5.3 VolumeView 119

5.6 Summary 120

6 Evaluation of IFIT **122**

6.1 Test case selection 122

6.2 Test cases 123

6.2.1 Case 1: Medical imaging data 123

6.2.2 Case 2: Elipsometry data 124

6.2.3 Case 3: CT data 125

6.2.4 Case 4: Bathymetry data 127

6.2.5 Case 5: Multivatiate data 128

6.2.6 Case 6: Scattered data array 129

6.2.7 Case 7: Computational flow dynamics data 129

6.2.8 Case 8: Computational flow dynamics data 130

6.2.9 Case 9: Finite element data 132

6.2.10 Case 10: Gel electrophoresis data 133

6.3 Test case evaluation 134

6.3.1 Successful application 134

6.3.2 Limitations to application 137

6.3.3 Factors affecting the utility of the visual techniques 139

6.3.4 Factors affecting software performance 141

6.4 Usability evaluation 143

6.5 IFIT compared with existing solutions 147

6.6 Discussion 151

7 Towards autonomous data input **153**

7.1 The Fourier transform 154

7.1.1 Application to data input 156

7.1.2 Limitations to data input 158

7.2 Summary 162

8 Conclusions and further work **163**

8.1 Summary of achievements 163
8.2 Further work 167

References **172**

List of Figures

2.1 Two models that describe the visualization process in terms of dataflow . 21
(a) Upson (1989) Model 21
(b) The conceptual diagram of visualization stated by Haber et al (1990) 21
2.2 Brodlie's (1993) model centred approach 23
2.3 Osland's (1992) visualization reference model 24
2.4 Simplified visualization framework by Gallop (1994) 26
2.5 Felger et al. (1992) concepts for different architectures which enable semantic interaction 27
(a) Loose coupled architecture 27
(b) Independent architecture 27
(c) Tight coupled architecture 27
(d) Tight coupled – single connection architecture 27
2.6 Major components of the visualization reference model proposed by Bergeron and Grinstein (1989) 29
3.1 Byte alignment; a) a set of four floating point values, b) the byte values which make up the floating point values, c) the effect of incorrectly aligning the array by one byte 54
3.2 An example of a structure exhibiting cell variable connectivity 57
3.3 An example of a structure exhibiting cell regular connectivity, in this case having triangular cells 57
3.4 An example of a gridded connection structure 59

3.5 Uniform rectangular positions in two dimensions. The difference between coordinate values for each dimension is constant 60

3.6 Variable rectangular positions in two dimensions. Coordinate values are shared for all nodes at the same intersection along each axis 61

3.7 Body fitted positions in two dimensions. Coordinate values are different in each dimension and for all other coordinates in other dimensions 61

3.8 Array(i, j) stores nodal positions in its columns and individual cells in its rows. 62

3.9 The 3! possible ways of interpreting the shape of an array containing three variables over two dimensions 65

3.10 Arrays illustrating the absence (a) and presence (b) of identifier data . . . 70

4.1 The dataflow model for file input 76

4.2 The value stage architecture 78

4.3 The structural stage architecture 80

4.4 The semantic stage architecture 81

5.1 A simple IFIT example 94

5.2 A module network that can import a file containing an array and two explicit parameters which define its shape 95

5.3 SelectBytes' user interface;. 98

5.4 BytesToValues' user interface. 99

5.5 TextToValues' user interface 99

5.6 A module network that can import a single parameter from a file 100

5.7 A module network that can import three adjacent parameters from a file . 101

5.8 A module network that can import three non-adjacent or differently typed parameters from a file 101

5.9 A module network that can extract an array of binary values from a file . 102

5.10 A module network that can extract an array of text values from a file . . . 102

5.11 A module network that handles an array of differently typed text values extracting three for usage 102

5.12 ChangeDimLat's user interface; the rank of the array is specified with the 'Dimensions' parameter, this enables parameters 'Dim0' ...'Dim9' which are otherwise hidden. These parameters are then be used to specify the shape of the array 103

5.13 Two networks that can input a Windows bitmapped picture (BMP) file. These solutions illustrate how values can be taken from the file and wired in as parameters for the file input process. The height and width of the image are taken from the 54 byte header and then used with the knowledge that this file has three values (blue green red (BGR)) per node to dimension the array. The solutions show the different way in which colour channels can be swapped from BGR to RGB 105

(a) Slicing the array to swap and then combine the variables 105

(b) Converting the array variable index and swapping the channels by reverse selection 105

(c) An example of the output these networks produce 105

5.14 TextView in action; two views of the same file using different plain-text character interpretations, the first illustrating a direct view of the file's content as plain text, the second interpreting using control characters and white space to separate values onto different lines. 107

5.15 The visual effect of trialling different widths for an array whose actual width is 640 109

5.16 Vertically interleaved lines casued by the width being less than the actual width of the array. 110

5.17 Horizontal repeats caused by the trial image width being a multiple of the correct dimension 110

5.18 DEM data comparison illustrating 'psychedelic' banding 111

(a) Same DEM exhibiting 'psychedelic' banding 111

(b) Correctly interpreted DEM 111

5.19 Two examples of artefacts caused by the actual rank of the array exceeding ImageView's *rank* 2 output 112

(a) Regular vertical breaks of continuity 112

(b) Apparent vertical repeats 112

5.20 Text values in a file as viewed through ImageView 113

(a) A 'wood grain' text texture 113

(b) A low contrast text texture . . . 113

5.21 Aspect ratio as an indicator of incorrect input parameters . . . 114

(a) A correct view of a CT dataset . . . 114

(b) A similar view of the same dataset, however its aspect ratio distorts the image which may lead the user to check the parameters they have used . . . 114

5.22 Two different binary interpretations of 32-bit values that result in line artefacts . . . 115

(a) 8-bit unsigned . . . 115

(b) 16-bit signed . . . 115

5.23 Textures illustrating the effect of multiple variables in an array . . . 115

(a) Binary data with multiple variables . . . 115

(b) Text with fixed width records . . . 115

5.24 Single distinct vertical break of continuity . . . 116

5.25 Contrast problems found when using ImageView with an automatic range generation for mapping data values to greyscale values . . . 117

(a) High contrast texture caused by error values . . . 117

(b) Low contrast caused by an incorrect selection including non-data values . . . 117

5.26 'White-noise' like texture taken from an ImageView of a JPEG image . . 118

5.27 ImageView textures that show different blocks of data, each block occuring where the texture changes dramatically. . . . 118

(a) Different blocks of text values . . . 118

(b) Diffferent blocks of binary values . . . 118

5.28 VolumeViewer displaying an incorrectly dimensioned array, and the same array when correctly dimensioned . . . 120

6.1 A file containing two X-Ray images held in a single array. . . . 123

6.2 A proprietary chemistry file format containing several variables. . . . 125

6.3 A DICOM CT dataset containing slices of a pig femur. . . . 126

6.4 A file containing bathymetry (sea depth measurements) in the form of a gridded DEM. 127

6.5 A file holding a scattered array of sample points with many variables of different binery types. 128

6.6 A file containing a scattered set of 3D points and one additional vairable . 129

6.7 A volume containing flow and pressure. The metadata is held in an external file (top pipelines). The loading network combines the metadata providing a reusable file reader for different sized volumes with this number of variables 130

6.8 A file containing simulation flow data 131

6.9 A 2D finite element CFD dataset comprising quadrilateral cells over a surface. 132

6.10 This data was previewed on printed output. The file format was proprietary and from old hardware. This prevented the user from accessing it for further analysis and visualization purposes. What was found was an array containing an 40 variables each of a genetics plot 134

(a) Genetics data input map with discovery modules still attached . . . 134

(b) ImageView output of the detected genetics data array 134

(c) IRIS Explorer graph with one profile from the data 134

7.1 A sine wave represented in both domains 155

(a) The time amplitude domain 155

(b) The frequency amplitude domain 155

7.2 The relationship between three different measurements in the time domain and the frequency domain 155

7.3 The pulse trains in power spectra for a $rank$ 2 and $rank$ 3 array 157

(a) $x(t)$ with periodicity d_1 in a $rank$ 2 array 157

(b) The power spectrum pulse train $P(f)$ of $x(t)$ with a peak separation of $\frac{1}{d_1}$ 157

(c) $x(t)$ with periodicity caused by the dimensions d_1 and d_2 of a $rank$ 3 array 157

(d) The power spectrum pulse train $P(f)$ of $x(t)$ with two sets of peaks with separations corresponding to $\frac{1}{d_1}$ and $\frac{1}{d_1 \cdot d_2}$ 157

7.4 The pulse trains in the power spectrum of a real world dataset which was a *rank* 4 array. The spacings which can be used to acquire dimensions $d_1 \ldots d_3$ are marked on each view of the spectrum 159

List of Tables

2.1 A visualization framework comparison, each author's framework is illustrated. Terms describing processes that provide the same functionality are horizontally adjacent in the table. The first process is that of general data input, which has not been handled by all authors. 30

2.2 Visualization software file input provisions other than programmed extension. †the number of modules which are used to produce this input 44

3.1 Fourteen n-node cells which are commonly used for deriving connections in cell regular and cell variable data 58

3.2 Illustrating the storage required for coordinate X when using different rectangular and body fitted grids 62

3.3 Table 3.3(a) A 4 by 4 Array containing variable X which, given any location in i is constant for all locations at j. Table 3.3(b) illustrates a compact representation of the same data. 64

(a) Normal representation 64

(b) Compact representation 64

3.4 User knowledge of a file input problem 71

5.1 Transformation modules in IFIT and IRIS Explorer; those marked † are provided with IRIS Explorer. The table shows the input and output of each module by its location. 91

5.2 Specification modules in IFIT and IRIS Explorer; those marked † are provided with IRIS Explorer. This table shows which modules are needed to specify the two types of output data given the set of available inputs. . . . 92

6.1 Comparison between AVS's File access objects and IFIT's input facilities 148

6.2 The modular network approach in relation to other types of file input approaches 150

7.1 The expected spacings between peaks in the frequency domain for a *rank* 3 array 156

Chapter 1

Introduction

The dramatic fall in the cost of computer hardware and the development of software supporting Visualization in Scientific computing (ViSC) has lead to an increase in the number of users who can benefit from visualization software. However, the uptake of such software is limited by its ability to read different data sources.

Hamming (1962) famously stated, "The purpose of computing is insight not numbers", visualization enables this to be achieved. It concerns the transformation of the abstract to the visual, and enables us to use our advanced abilities for processing visual stimuli to understand information. Visualization in scientific computing (ViSC) is a term first coined by McCormick et al. (1987) and concerns the improvement of software and skills supporting scientific visualization by combining advances in computer hardware with improved methods for data processing and generalized models of manipulating and storing data. Developments in ViSC have lead to a range of powerful commercial software packages which enable a user to gain insight from their data. The number of applications for such systems has grown, encompassing research and development in earth sciences, medicine and industrial design. A trend in visualization software has been toward ever more versatile environments that offer support to many application domains.

McCormick et al. (1987) first stated the phrase 'fire hoses of information' to convey the vast amount of information being produced by many of the computer-based scientific

systems of the day. He also described the steps and resources needed to make sense of such data sources. One section of the report describes the problem of "The information-without-interpretation dilemma", this notes the diversity and specifies some of the fire hoses of information to which he referred earlier in the report. While many advances have produced a visual interpretation for these types of data, the problem of information without interpretation still remains in file access. Visualization packages cannot use data in any files for which they have no reader, there may be information in the files, however without a reader there is no interpretation for it, rendering it useless. Modern visualization systems like modular visualization environments (MVEs) are highly adaptable and cater for a wide range of data sources. The only major hurdle for many users is inputting their file format. For users who have software which is capable of producing a standard file format output, this is usually a simple issue which involves finding the right reader from a library or simply selecting their file to open it. Tools are provided in most MVEs for when this is not the case. However, these tools are seldom simple and are ill equipped for solving problems where users lack knowledge about their format or data. Finally, users can resort to creating their own extension to the MVE's capabilities to input their particular data, this can be both complex and time consuming.

Before specifying the problem that this thesis intends to solve in detail, we shall examine the rise of visualization as a tool for science and its transition into a computational science. After this the area of visualization and the demands which have produced the myriad of files that now require an alternative method of input will be discussed.

Visualization is a diverse subject area which spans many fields and disciplines, the next section will describe some of the main elements of visualization and briefly chart how it has risen to a computational field.

1.1 Visualization

There are many definitions for the term 'visualization'; each provides a clue to the nature of the word's use, 'visualization' is defined in the OED (Pearsall 1998a) as a derivative of 'Visualize' which is defined as:

> "1. form a mental image of; imagine: it is not easy to visualize the future",

and

> "2. make (something) visible to the eye; DNA was visualized by staining with ethidium bromide.",

Webster's Masters English Dictionary (*Webster's Master English Dictionary* 2002) agrees with the first definition with:

> "to form a mental picture of; to make visible to the mind or imagination; to construct a visual image in the mind".

There are two themes from these dictionary definitions which are useful when re-applied to the definitions used by practitioners in the field of ViSC. First, that visualization involves mental models which are of a visual nature. Second, that visualization involves the presentation of visual imagery to fit what maybe invisible or non-visual phenomena into a visual mental model.

McCormick et al. (1987) stated in relation to scientific computing that:

> "Visualisation is a method of computing. It transforms the symbolic into the geometric, enabling researchers to observe their simulations and computations. Visualization offers a method for seeing the unseen".

This takes several notions, that visualization in this field is related to a computational problem, it is transformational and it is view oriented. Haber and McNabb (1990) take a similar transformational notion of visualization in relation to scientific computing with:

> "A series of transformations that convert raw simulation data into a displayable image. The goal of the transformations is to convert the information into a format amenable to understanding by the human perceptual system".

'Visualization' is a much debated word. All these computational definitions when combined with the dictionary definitions give us a picture of what visualization means. Visualization for a person involves the construction of a mental model which maps some phenomenon to visual imagery in the mind. Therefore, much like committing a complex idea or mathematical problem to paper, externalizing the mental model to some other medium enables it to be examined without the burden of internally generating the imagery. A person can have a mental model of a city, but the load for utilising this model for any particular problem can be harder than using a map, which can be thought of as an external mental model of the city on paper. This externalization enables the thinker to gain a new perspective, a clearer focus or an overview which can in turn prove insightful.

Another important aspect of visualization is that this mental model once externalized becomes easy to communicate through visual presentation. Again using the map problem, communicating the location of a place from your own internal mental model is difficult, but pointing to the location on a map is relatively easy. Verbal communication of such a visual model can be complicated; language often relates to qualitative as opposed to quantitative descriptions of visual phenomena, whereas visual presentations use the mind's powerful visual interpretation abilities to find patterns, trends and interesting features that can lead to insight.

Taken into the realm of computation, visualization becomes a way of processing data into visual models which can be interacted with. This process, just like mapping data to a

mental model, is a series of transformations from the abstract to the visual. The mapping of the data to a view can be naturalistic or symbolic, depending upon the dataset and its attributes.

Describing visualization by example leads us to a list of techniques including: graphs, charts, maps, scale models, atomic models, architectural models and engineering blueprints, some of these are described in the following sections.

1.2 Historical visualization

From the earliest maps and charts like Minard's map of Napoleons march to Moscow (which can be found in Tufte (1983a)) through to newer uses of visual tools like atomic wire models Kendrew et al. (1958) used in chemistry, the use of wind tunnels with smoke for understanding a shape's aerodynamic properties and the use of scale models in ship stability testing.

Visualization existed for centuries prior to the advent of the computer. It cannot be characterised by any specific algorithm, process, or application. However, it can be typified as a way of thinking and communicating involving the visual senses. To that end the great thinkers in history have used visualizations to communicate their ideas to others and to gain insight into their problems. The following three sections describe some interesting "real-world" visualizations that have been created.

1.2.1 Graphs

Graphs are some of the earliest forms of visualization, and have been used to show relationships between n variables where one of the variables is independent, and all the others vary in relation to the independent one and so are dependent upon it. A reputed 10th-11th century graphical representation of the planets' orbits Tufte (1983b) is one of the earliest known graph visualizations. The graph shows multiple lines, each one representing a

planet's inclination over time, it is the earliest known example of an illustration attempting to use visual form to describe natural phenomena and was taken from a section of text for monastic schools.

1.2.2 Maps

Maps such as topological maps and terrain contours, have long been used to show two-dimensional spatial relationships, often in a geographical context. Charles Joseph Minard created some elegant maps that illustrated Napoleon's march to Moscow Tufte (1983c). The map illustrated many factors over the course of the campaign, including troop numbers, environmental factors, and the paths of the army's advance and subsequent retreat. It was not the first map ever drawn, but it highlights how complex information can be rendered in an elegant manner which makes it more understandable.

1.2.3 3D Models

Constructing a small-scale mock up, or a physical reconstruction, of a phenomenon is a form of visualization. The use of 3D models to gain an understanding of a problem is common. The aerospace industry create visualizations for airflow over the surface of their aircraft by creating a scale model and placing it in a wind tunnel with jets of coloured smoke. In a similar vein, the ship building industry also uses scale models to test the stability of hull designs in large water tanks. There are many other examples in science, engineering and mathematics where the problem has been represented by a physical creation. Two other famous uses of physical model visualizations are, James Clerk Maxwell's clay surfaces, and Richard's Box. The details of these visualizations are outlined below:

James Clerk Maxwell was a renowned scientist of the nineteenth century. Among his accomplishments were the first 3D visualizations using clay models West (1999); the inspiration for these came from a mental model described by J. Willard Gibbs. The mental

model Gibbs was using would be known as a surface plot in current terminology. Maxwell saw the value of these methods for thinking about scientific data from Gibbs paper and created a clay sculpture using Gibbs' data and mental model. Maxwell's approach allowed Gibbs' model and mathematical techniques to gain widespread acclaim, and they have been used for almost a century. The visual aspect of this work has been overlooked until very recently and has come to the fore with the use of 3D computer graphics. From Maxwell's models both an understanding of the data and a communication of how Gibbs was thinking about the data could be gathered; this really shows the value of a good visualization and how it can allow ideas to be communicated more effectively.

Richard's Box Richards (1968), is a method of visually combining an electron model with electron density contours. It involves the use of thin acetate upon which contours are drawn. These are mounted on 36" x 36" Perspex sheeting and illuminated. This construction is hung by piano wire next to the molecular model that is made to the same scale and also illuminated. Finally a half-silvered mirror is used to superimpose the two models, allowing the user to judge if the map is a good fit to the model. This visualization shows how visualization techniques can be used to judge accuracy and pick out abnormalities in the way the data has been interpreted.

1.3 Modern computing period

As computing developed as a tool for science, many programs produced outputs that could be printed or displayed. Initially software would be specifically coded to produce an output for a particular platform using very customised routines which accessed the graphics hardware directly. Over the years, multiple layers have interceded releasing software from being tied to a particular type of graphics hardware, libraries like OpenGL, for which a good guide can be found by Woo et al. (1997) alongside GKS and DirectX, have all generalized the problem of producing graphical outputs at a low-level. High-level libraries like PHIGS and OpenInventor (*Open Inventor* 1993) have created additional

layers which make user interactions and view control easily obtainable for users who need to program visualizations into their software. These libraries lead to a range of off the shelf or turnkey products which each offer a particular field a range of appropriate data manipulation and visualization techniques. The next major development was the Modular Visualization Environment (MVE), apE by Dyer (1990) and AVS by Upson et al. (1989) were both early examples of such environments. MVEs are toolkits for building visualization applications. The user can choose from a range of data processing and visualization techniques and connect them together to create a custom application for their problem.

1.4 Data input for scientific visualization

McCormick et al. (1987) noted the range and quantitative nature of scientific data, his 'firehoses of information' quote still describes the data sources for scientific visualization. Scientific data comes from a diverse range of interdisciplinary sources. Scientific data can be described in general terms as being numeric, quantitative and structured. This data comes from physical measurements or simulated results. Visualization system developers are faced with a need to support these sources and the myriad of existing 'standard' file formats which are used by different disciplines.

Commonly-used scientific data formats tend to be widely supported. As a result, these do not normally pose a problem for the users of scientific systems. However, the proprietary file formats often found in obscure measuring hardware and file formats produced by users, or by other software developers can provide a problem for the user. This is because it is unlikely that the visualization system will be able to directly input data from these formats.

1.5 Inputting application data

Modern scientific data by its nature does not always come from a user-designed item of equipment or piece of software, and this trend is increasing as stock items of scientific hardware, measurement equipment and software become more prolific and are developed by external companies. This is moving the user's role further away from software engineering and towards the scientific aspects of their work. While this can be considered a more efficient use of their time it can impede their ability to use visualization software because they are no longer aware of the technical skills needed to translate their data onto other systems.

Most programs provide file access as part of their functionality. The ability to load or create data for viewing or modification and the ability to save this data is an essential and often underestimated aspect of most software. Only when users find they cannot load a particular form of data or save it in a form that can be used elsewhere does this functionality become questioned. Most software developers offers a range of options for loading and saving data. These options usually relate to formats which are simple and therefore more commonly implemented or industrial standards in their particular field. Many examples can be found of applications from non-scientific fields which have data input problems caused by a plethora of file formats, these include word processors, raster graphics packages and desktop publishing packages. All have many different standards and sources for their application data, and problems can often arise, especially between different hardware platforms, when a file format is incompatible between packages. The same can be said for many other types of general purpose application, MVEs with their high adaptability and applicability to many kinds of scientific data are perhaps a worst case of this phenomenon because the number of different fields and applications that they could conceivably support is so great.

Most file access routines open a specific file format. They fail completely if an attempt is made to input a format which has not been defined by the routine, usually opening an

error message dialogue box to warn the user of the error. If MVEs were to solely adopt this route, then every file format would need a specific routine to be developed and as there are so many formats this is an unreasonable notion. Equally, because so many file formats store such similar data, it would involve a huge replication of code at the expense of the developer. Currently file formats are solved on a case-by-case basis. Such solutions can be created by the user adding routines to the system, or by the developer aiding a user by adding a routine to access the user's data. Finally, if conversion software exists that supports both the user's file format and an appropriate format used by the MVE, this can be used to input the user's data.

The hardware that produces a file can affect its content rendering it inaccessible using identical routines on a different hardware platform. Some formats compensate for this effect with rigorous standards for interpreting their content, others do not. These differences along with different types of data, data storage philosophy, demands and uses of file formats has lead to a multitude of different data storage formats. These formats impede the usage of visualization software due to the difficulties faced by users in accessing their data.

1.6 Scope and goals for a file input system

The key problems that face file input systems for scientific data will be described in this section in addition to a specification for both the goals and scope of this research.

1.6.1 Research problem

The aim of this research has been to improve the usability of scientific visualization software. Within this remit, it has identified a key problem faced by users of this software, namely file input. This process describes the entry of a user's file-stored data into one of these programs. The difficulty of this task ranges from entering the location of the file, to

a programming project incurring costs in user's time and the expertise of others.

Currently, there are six techniques that users of scientific visualization software can use to input their data. These methods are described in chapter 2 under the names of hard-coded, header files, scripted readers, external tools, modular networks and programmed extensions. Most MVE packages implement one or more of these techniques to provide users with access to their data. Each method can be described in terms of the complexity of its usage and its overall flexibility.

This projects aim is to find a better method of inputting data into ViSC systems. This project targets non-standard, user-defined and field specific file formats (those usually not covered by hard-coded solutions). Overall, this research aims to:

- simplify the problem of creating solutions to file input problems for ViSC systems;
- find a solution which can be applied to a broad range of scientific file input problems;
- work towards an automatic solution for file input.

A large part of this problem is tied to a lack of standardised methodology for dealing with file input and output. The result of this has been an ad-hoc approach to dealing with files of different formats. Therefore, non-standard file formats present a wide variety of ways in which users have stored their data, each different enough to need an individual solution, and yet each similar enough to make a common description seem plausible. As part of this research a general methodology for solving these problems should be found; this will allow the creation of more advanced tools for file input which can offer the needed flexibility without the complexity that comes with existing approaches.

The final solution should enable a wide range of file formats to be input. This should include file formats from different fields and different types of datasets.

The project should attempt where plausible to investigate automatic methods for determining the content of a file, with a goal of reducing the need for user interactions in the file input process.

Given the overall goal of loading non-standard file formats, the requirement for a simplification over existing approaches and a high level of flexibility, the scope of this research is now described.

1.6.2 Scope of research

There are numerous file formats in circulation, over 100 in the field of Chemistry alone. In order to find a solution, we need a representative selection of file formats that are not currently supported by the hard-coded loading systems of the MVE we choose to implement any system on. Additionally, the range of files we are attempting to find solutions for would ideally be user-defined, or output from proprietary software and hardware, and therefore not commonly available.

The file formats which will be the target area for this research are described in chapter 2 under the sections termed 'field-specific' and 'user-defined and non-standard'. They include scientific file formats that are:

- the output of proprietary application software;
- the output of user developed software that does not use an existing standard;

The scope of this project is to exclude file formats using the following techniques:

- compression;
- encryption;
- sub-byte values.

Compression whilst widely used in medical and satellite imaging data, is less prevalent in field-specific file formats and user-defined file formats. Most file formats that can use compression are also likely to have an option for the output of raw data. Finally, the complexities of implementation and added proprietary rights issues, have led to the removal of compressed files from the scope of this project.

Encryption is a similar case to compression. It is only usually encountered among the formats put forward by the providers of copyrighted materials like films, music and electronic literature and is therefore outside the scope of this research.

Finally, the notion of values being stored in less than one byte has also been dropped from the scope of this project for reasons of low usage. In the author's opinion and experience, they do not represent a wide enough cross-section of the file formats within the other specifications of this project to require examination at this time.

1.7 Thesis structure

This thesis has the following structure:

Chapter 2 – Visualization models, file formats and input tools, provides a background for this work. Current models for storing and processing scientific data are described followed by a review of the current state-of-the-art in data input and file formats found in the field of scientific visualization

Chapter 3 – Data input considerations, takes an in-depth look at the theory and mechanics of acquiring data from a file. It describes the different facets of the file input problem in terms of both the data involved in and how it is represented in file formats.

Chapter 4 – A new approach to file input, presents an approach and model for solving file input problems followed by a supporting architecture for the file input process.

Chapter 5 – Using the interactive file input toolkit (IFIT), presents a new approach to file input. It describes IFIT's structure and how its different components are used to solve file input problems

Chapter 6 – Evaluation of IFIT, presents a range of test cases which are used to produce a qualitative evaluation of IFIT's abilities. It also evaluates IFIT's usability and compares IFIT with the different input techniques described in chapter 2.

Chapter 7 – Towards autonomous file input, illustrates how the user in the loop interactions presented in chapter 5 have lead to algorithms which could automate some aspects of the file input problem

Chapter 8 – Conclusions and further work summarises the work covered by this thesis before describing the conclusions that have been reached and some avenues of future research

Chapter 2

Visualization models, file formats and input tools

Software and intellectual developments supporting ViSC have lead to flexible visualization software like MVEs. These tools enable the user to construct applications that meet their visualization requirements through an easy-to-use interface. These highly flexible environments offer their users many different visualization techniques for their data.

Many MVE users face a challenging problem when loading their data. This stems from the myriad of different file formats which abound in science and engineering. Their number prohibits MVE developers from directly supporting every file format, instead targeting just those which are common. This is a limiting factor for ViSC as it moves into fields where the user is no longer the architect nor designer of their software and its output.

Overall this chapter will describe a context for file input in scientific visualization, which will be the basis for comparisons that will be drawn in chapter 6. The leading models and frameworks for MVE design will be reviewed. In addition, this chapter will describe some of the more common file formats and standards that are used in scientific circles for data representation. Finally, different visualization systems will be reviewed in terms of their input tools and provisions.

2.1 Scientific data storage models

The necessity for ViSC systems to have powerful data models has grown up alongside the development of interdisciplinary visualization packages. Prior to the development of these systems, ad-hoc visualization routines and application specific visualization systems were the major source of visual output; such systems would be integrated with the specific data structures that the application field required. While this was effective for software supporting a single field or application, an interdisciplinary visualization tool requires an abstract data model.

Scientific data comes from a myriad of different sources. Gallop suggested in Gallop (1994) that by taking a step back from the format of data arriving at the system, and instead looking for shared characteristics in different datasets, an abstract data model could be produced that would store many types of data. The next three sections outline three powerful abstract data models that have been used in the design of commercial visualization software.

2.1.1 Lattices

Bergeron and Grinstein (1989) propose the need for a standard form to store a user's arbitrary database while it passes through the visualization system. This process is likened to the conversion of coordinates to the Cartesian axial system to enable them to be rendered in PHIGS. A primitive type termed the lattice was defined as their standard form. Lattices preserve any ordering present in user's databases in addition to describing their data's dimensionality. The lattice notation L_n^k is used to describe the dimension of the domain k and the data n. The following are examples of how different types of data would be classified using the lattice notation:

- L_3^0 Scattered 3D points;

- L_3^1 line in 3D;
- L_2^1 line in 2D;
- L_3^2 Surface in 3D;
- L_1^3 Scalar volume;
- L_3^3 3D Vectors in a volume.

While the notion of the lattice does describe some aspects of a data set, it only classifies scientific data. Bergeron and Grinstein do not explain what is stored in a lattice structure, nor do they elaborate on the limitations of the lattice, which can be assumed to include an inability to describe some types of irregular cell-based data. The lattice model can describe many types of gridded scientific data and was used in the design of IRIS Explorer's primitive data structure.

2.1.2 Fibre bundles

Butler and Pendley (1989) single out fibre bundles as the "natural geometrical objects for visualization". A fibre bundle is a structure from differential geometry; it is a space derived from a pair of arbitrary spaces. A bundle comprises the Cartesian product of a base and a fibre space. For example, flow data over an aircraft wing where the base space is the surface geometry of the wing and the fibre space is a vector space of airflow over this surface. The fibre bundle structure defines a copy of the vector space for every point on the wing's surface.

This abstract model was extended in Haber et al. (1991) to form the field data model, which is a unified abstract model for scientific data. The field model has been used in the design of IBM's Data Explorer. Their paper illustrates a data model for describing continuum fields, discontinuous geometries and wire line structures. Much like a fibre bundle, a field is comprised of independent and dependent variables. The independent

variables in the field model are described by a base space and the dependent variables are described by a fibre space. Field space is their terminology for a fibre bundle and, as such, is the Cartesian product of the base space and the dependent variable space.

Their model extends the notion put forward in the earlier paper, it describes how aspects of a dataset's regularity can be used to produce a uniform compact data representation for both regular and irregular grids. One example of this compact form is to define how regularly sampled positions in a gridded dataset can be reduced to a few shared parameters. They go on to show how the Cartesian product of their field elements can be used to produce a compact representation of cells that dramatically reduces the amount of information that needs to be stored in order to describe such data.

2.1.3 Classification of scientific data with the E notation

Brodlie et al. (1992a) presents a classification scheme for scientific visualization which attempts to model the underlying field of the data. Much like Bergeron and Grinstein's lattice classification it categorises different types of scientific data by their type and dimensionality. Unlike their work it aims to classify the underlying field of the data, not the data itself. Brodlie's classification is based upon the notion that there is an entity which is the desired output to visualize. This entity can be described in terms of a range of values over a number of independent variables, expressible in mathematical terms as a function of many variables.

The notation is based upon an entity E and classifies it by the type of function and the dimension of its domain. This type of function is a superscript which can be classified as one of the following:

- no function just points to visualize(P) ;
- a single value or scalar (S) function;

- an array of values or vector (V) function followed by a subscript defining the number of components;
- a matrix of values or tensor (T) function followed by a subscript which defines the shape of the matrix.

The dimensionality of the entity is defined using a subscript. An entity's dimensionality can be over every point in a continuous domain, e.g. E_3, or over regions of a continuous domain, e.g. $E_{[2]}$, or finally as an enumerated set, e.g. $E_{\{1\}}$. Time can add a dimension to the domain or be classified separately using t in the domain subscript. Finally composite representations can be represented by combining multiple entities in a nested fashion. Below are some examples of different visualization classifications using the E notation.

- Bar chart is categorised as $E^S_{\{1\}}$
- Line contours are categorised as E^S_2
- 3D Vectors in a volume are categorised as $E^{V_3}_3$
- Second order tensor volume is categorised as $E^{T_{3:3}}_3$

This classification scheme, while effective at grouping techniques, does lead to some ambiguities in distinguishing between the data and the visualization technique.

The E notation illustrates how many different types of visualization are rooted to the same underlying field. This abstraction equally applies to the formats used for storing scientific data. A scientific file format has to represent the underlying field of a particular dataset. Therefore, a classification scheme of this field provides insights for the storage of scientific data.

2.2 Process models for visualization systems

ViSC attempts to facilitate scientific insight. As a result, several different analytical models of scientific investigation have been used as a basis for many of the frameworks and reference models developed for ViSC. These reference models illustrate the processes needed to turn data into images. They describe the visual nature of most scientific analysis and evaluation. Several important conceptual models for the process of transforming raw data into visuals and for engaging in scientific problem solving and analysis are presented in this section.

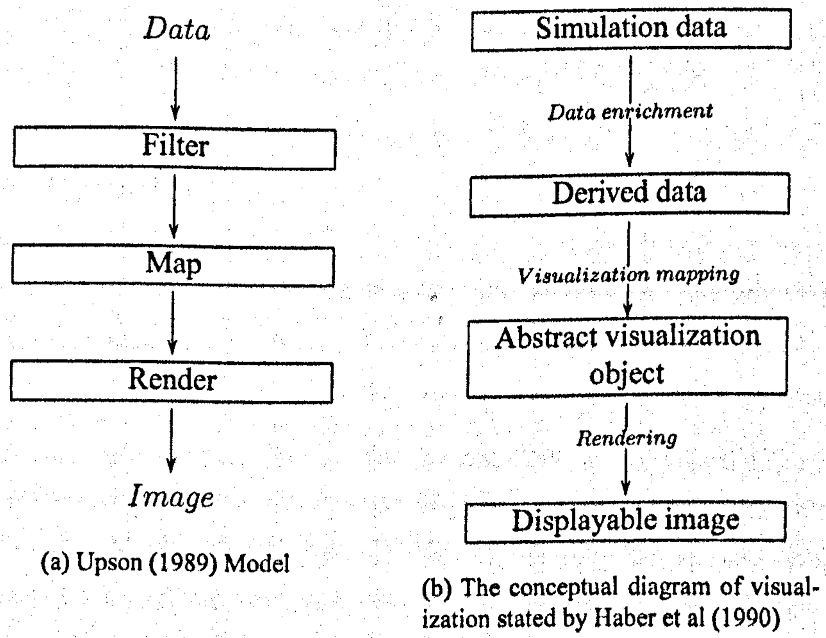
2.2.1 Dataflow

Upson et al. (1989) described the visualization model of AVS as a breakdown of the steps involved in the analysis of a numerical simulation. Haber and McNabb (1990) later described a conceptual model for visualization which similarly saw the visualization as facilitating the evaluation of simulation results. These two models have been influential in the design of visualization systems, since the concept of dataflow is both a simple and a powerful breakdown of the steps involved in turning scientific data into images. Each model views the visualization process in a subtly different light, although as Wood (1998) states:

> "they are in many ways the same model expressed as either a 'process driven' model (Upson et al) or a 'data driven' model (Haber and McNabb)."

The main differences between them lie in the descriptions behind their utility. Haber and McNabb use the notion of the Visualization Idiom as a device for illustrating to users what has been done to their data and, therefore, explain the meaning of the output visualization. They also describe a layered software design and architecture for producing scientific visualizations, which takes much of the developmental burden from the user and

encapsulates it into high-level reusable modules freeing the user to focus on their science. Upson et al's description, however, was used to facilitate a practical implementation of what is now a commercially successful MVE.



(a) Upson (1989) Model

(b) The conceptual diagram of visualization stated by Haber et al (1990)

Figure 2.1: Two models that describe the visualization process in terms of dataflow

For the purposes of this work, the common terminology of filter, map and render will be used to describe the three transformations which both these models describe.

Filter is the first stage in figure 2.1(a) and the first transformation in figure 2.1(b). It serves the purpose of preparing data for transformations later on in the visualization pipeline by reducing the data into a more relevant and meaningful form. Examples of filter operations include interpolation, extrapolation, smoothing, selection, sub sampling and the calculation of gradient and flow lines from a vector field.

Map defines the transformation of the user's data into what Haber & McNabb termed the Abstract Visualization Object (AVO). These are geometric objects that can be rendered. The conversion of filtered data into AVOs involves mapping variables

into attribute fields of graphical objects. An AVO may have one or more attribute fields including geometry, time, colour and surface texture.

Render encompasses all the operations required to turn the AVO into an image. As a result it deals with the geometric and viewing transformations of the AVO followed by operations including illumination, colouring, texturing and hidden surface removal. This stage marks where the process of visualization leads into the graphics pipeline Foley et al. (1996).

Dataflow has come to define the de-facto standard model for data manipulated in current MVEs and as such it is an important concept. However, it is interesting to note what dataflow lacks. Firstly, there is no concept for preventing the user creating visualizations which are incorrect because they have used an inappropriate numerical operation upon the data. Secondly, because dataflow in the context of these models takes data directly from simulations, they do not specify nor recognise the need to exchange data with external sources and systems.

2.2.2 A model centred approach

Brodlie's approach described in Brodlie (1993) takes the notion of dataflow and stresses the additional need to model the user's data to prevent inappropriate transformations from being performed upon it. Without this modelling to define the ranges and appropriate numeric manipulations for a given variable, there is scope for the production of a visualization containing semantic errors. This would either provide a plainly incorrect view of the user's data or subtly alter its meaning, and hence, interpretation. This notion of scientific scrutiny is also raised by Haber and McNabb (1990); the notion of misuse, however, is not discussed, only the need to offer transparent methods that enable a visualization's meaning to be understood. In Brodlie (1993) the need to model the user's data is exemplified using a measurement of the percentage of oxygen in a sample during

a chemical reaction. If the raw data values are interpolated using a cubic algorithm, the output can have negative values which are physically impossible and, if visualized, will result in an incorrect visualization. Essentially, anything users know about the limits and domain of their data should be used to model an underlying field; this can then be used to produce an accurate and meaningful visualization. Figure 2.2 illustrates Brodlie's model of the visualization process which uses a two stage approach. First, an empirical model is constructed from the data samples during the modelling stage which recreates the data's underlying field. Second, data is extracted from the underlying field in the viewing stage to produce a meaningful view of the data.

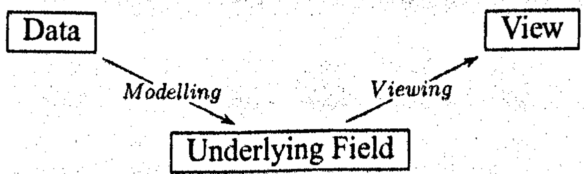


Figure 2.2: Brodlie's (1993) model centred approach

Brodlie also continues the development of the E notation described on page 18. This new scheme also regards the underlying field as a function of one or more variables based on the field model. These functions are classified by their output and input or dependent and independent variables. The classification notation uses the form of a function $F(x)$, where F can be ordinal (O) or nominal (N) and represents the dependent variable, and x represents the independent variable.

Ordinal variables have values that can be put in order, whereas nominal values are like enumerated types, i.e. with names and values but no inherent order. The type of the dependent variable, be it a scalar (S), vector (V) or tensor (T), is also classified and can be aggregated using the '+' operator. Vector dimensions are denoted using a subscript value, as are the dimensions and rank of Tensors. The independent variable is classified using a subscript showing its dimensionality. It can be subject to a range '[...]' or a restriction '{...}' operator.

- $N^S(O_2)$ Place names on a map
- $O^S(O_2)$ Height over a 2D region
- $O^{2S+V_3}(O_3)$ Pressure, temperature and flow in a volume
- $O^{T_{4:4}}(O_3)$ A four dimensional second order tensor defined over a volume

2.2.3 Osland's visualization reference model

The Osland (1992) visualization reference model, illustrated in figure 2.3, breaks up the visualization process into nine stages. Each stage passes data both up and down the pipeline, while receiving control parameters from the command interpreter.

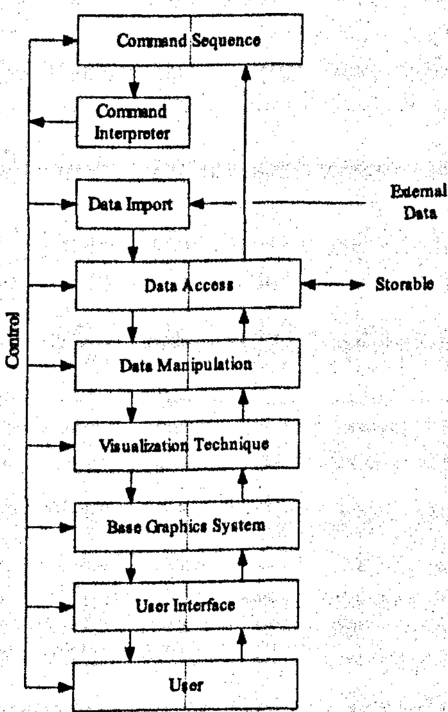


Figure 2.3: Osland's (1992) visualization reference model

The pipeline's upstream flow (towards the user) converts either internal or external data into a visual output for the user. Its downstream flow (toward the command sequencer)

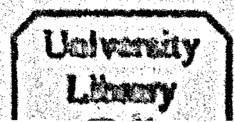
passes user input in the form of metadata. The Base Graphics System, Visualization Technique and Data Manipulation modules are all described as transforming the metadata coming back from the user toward the command interpreter, which implies an inverse mapping occurring at each of these stages. An example of this could be the transformation of mouse coordinates into screen and then world coordinates, which could then be transformed into the domain of the visualization technique and used to access the underlying data values at the location of the mouse.

The data import stage is particularly interesting, more for its inclusion and description than its contribution to scientific visualization or data input. Its inclusion and description emphasise that visualization systems will primarily deal with external data sources. This highlights the need for facilitating simple data input in any visualization system, as more often than not the user will be accessing data created prior to using the visualization system.

Gallop (1994) proposed a framework for visualization software which takes Osland's visualization reference model and simplifies it to become a four stage bi-directional pipeline. While it does show a clear breakdown of visualization from base graphics to user data, much like the seven layer model. It looses the notion of data input, instead opting for an application as the ultimate source of data.

2.2.4 The Visualization Input Pipeline

Felger and Schroder (1992) propose an approach for enabling visualization systems to provide interaction with the application data which has been changed during the visualization process. This type of control termed 'semantic interaction' allows the user to work directly with their data values after they have passed through the visualization pipeline and been turned into an image. Using a conceptual pipeline with processes for the data source, data preparation, graphical mapping, rendering and display, they propose a 'visualization input pipeline' or VIP to complement the existing visualization pipeline which is termed


University Library

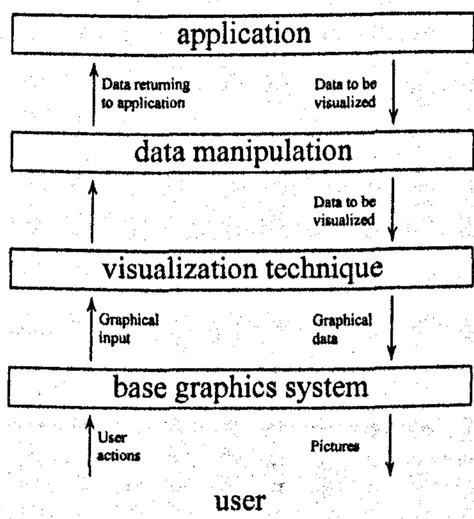


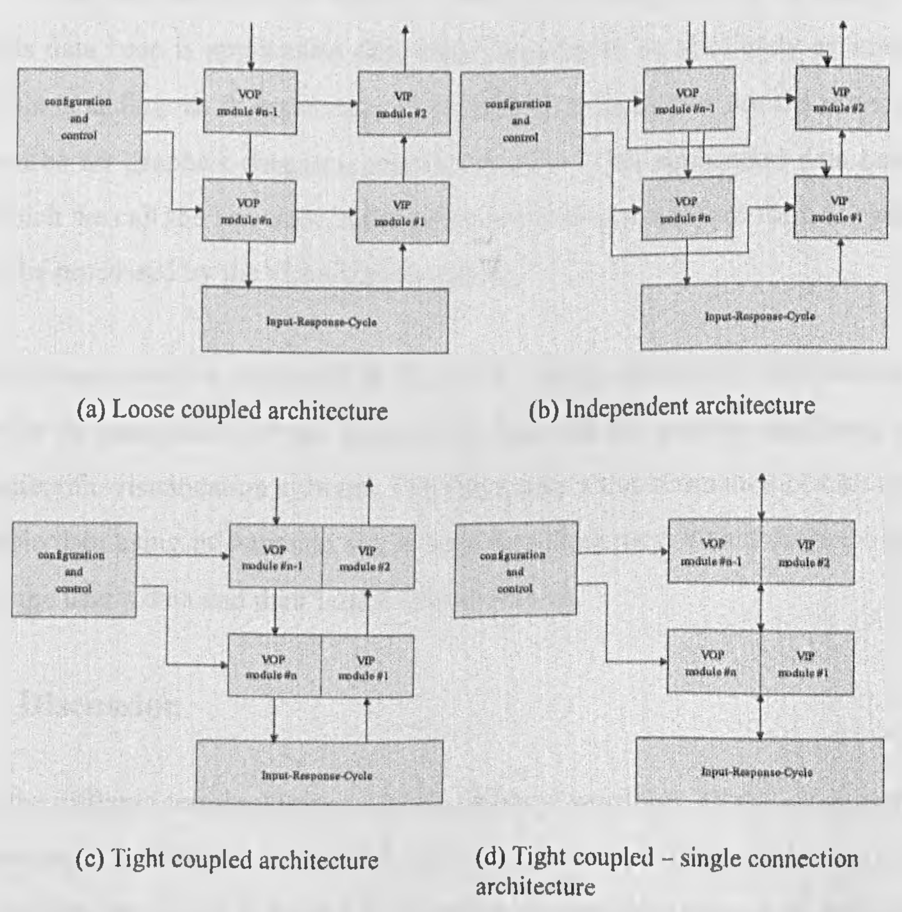
Figure 2.4: Simplified visualization framework by Gallop (1994)

the visualization output pipeline (VOP) to avoid confusion. This VIP is the inverse of the VOP, turning user interactions with the visual output of a visualization system back into data values and control parameters. In MVEs this translates to a requirement for existing modules to have corresponding ones which produce an inverse mapping for the image, AVO and filtered data back into the user's raw data values.

They present three different techniques for inverting data from the VOP, to account for functions that have no one-to-one correspondence between their output and input data. They also present four different architectures for coupling the VIP and VOP within a visualization system, each of these is illustrated in figure 2.5. The architectures range from independent to tight coupled single–connection. They differ in the level of shared resources and communications they use to enable semantic interaction.

The independent architecture requires every VIP and VOP module to have a separate process, which means that every VIP module requires three connections; these provide parameters, interaction data and source data from the corresponding module in the VOP. Conversely, the tightly coupled single–connection architecture has VIP and VOP modules

sharing a common process, parameters and data in addition to using two-way connections to send and receive data between processes. The other two architectures lie between these in terms of connections and shared data. All the architectures attempt to minimize the change required to enable semantic interaction for any given visualization system.



(a) Loose coupled architecture

(b) Independent architecture

(c) Tight coupled architecture

(d) Tight coupled – single connection architecture

Figure 2.5: Felger et al. (1992) concepts for different architectures which enable semantic interaction

Overall this paper shows how the notion of dataflow can be extended to user interactions with the visualization output and promotes the merits of enabling users to acquire and change the actual data values through this form of interaction.

2.2.5 Visualization reference model

Bergeron and Grinstein (1989) proposed a data model, described in 2.1.1, that recognises the need to model the user's data source. The reference model which they describe is for the visualization of multidimensional data. With respect to user data sources it states that:

> "The user has a specific data base which needs interpretation. In principle, this data base is application dependent and should be absolutely arbitrary. This is analogous to application-dependent data bases that are the ultimate source for graphics data in a graphics system. This application data base, which we call the *raw data*, must be represented in a standard form in order to be processed by the visualization system,"

Their reference model is illustrated in figure 2.6, and is noteworthy with respect to this project for its recognition of raw application data and the general data input problem facing scientific visualization systems. They prescribe a transformation of the user's data into usable data using information stored in a data dictionary, which defines a mapping between the user's data and their lattice data structures.

2.2.6 Discussion

Despite the different terminology and subtly different wordings, all the reference models for processing scientific data refer to the same three processes. These processes, as termed by each author, are shown in table 2.1. To reflect this project's interest in data input, the table also includes any data input processes which have been included in the models.

There are major differences in these models as they address different issues relevant to scientific visualization. Visualizing simulation data is the primary concern of both Upson's and Haber's models, whereas Bergeron, Brodlie and Osland's models address the need to visualize both simulated and observed data. Finally, both Felger and Osland's models have an emphasis is on the user's interactions with the visualization process both addressing separate issues.

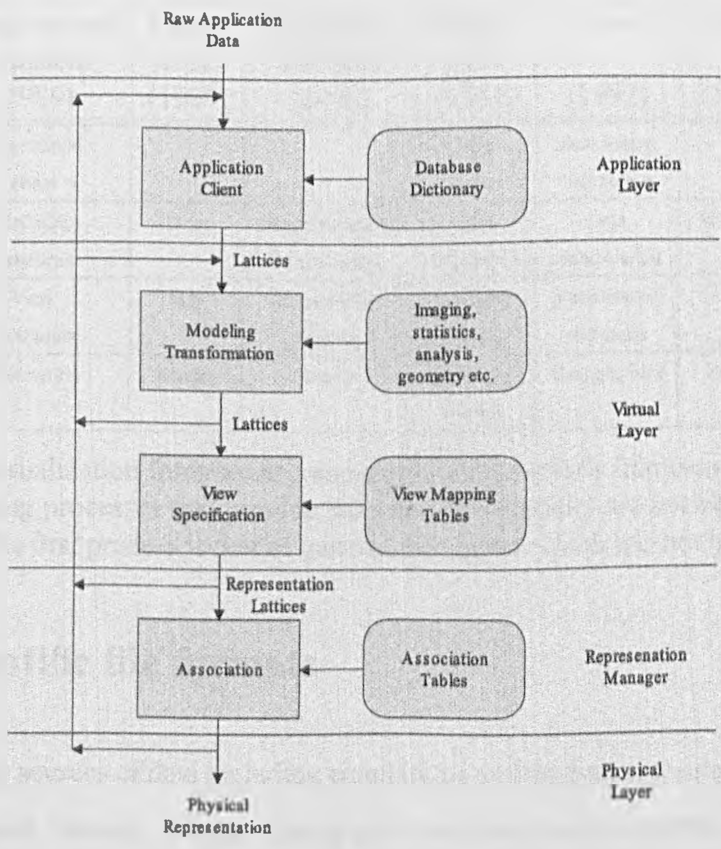


Figure 2.6: Major components of the visualization reference model proposed by Bergeron and Grinstein (1989)

The data models presented in section 2.1 directly relate to the problem of data input as they describe both the type of information which is to be transferred and the way it is generalised in MVE systems. The classification schemes by Bergeron and Grinstein and Brodlie et al. also describe information in the visualization system, and are relevant as they enable different types of visualization to be categorised by the type of underlying field which it represents. Finally, the process models show current thinking on processing visualization data. While all these models provide powerful and general breakdowns of the process of turning data into images, whilst also addressing issues of control, interaction and the need for correct processing for the type of data, it is notable that even models which show how the data initially arrived in the system do so in only a cursory manner.

Bergeron and Grinstein (1989)	Upson et al. (1989)	Haber and McNabb (1990)	Felger et al. (1992)	Osland (1992)	Brodlie (1993)
Application client			Data source	Data Import and access	
Modeling transforms	Filter	Data enrichment/ enhancement	Data preparation	Data manipulation	Modelling
View specification	Map	Visualization mapping	Graphical mapping	Visualization technique	
Association	Render	Render	Rendering system	Base graphics	Viewing

Table 2.1: A visualization framework comparison, each author's framework is illustrated. Terms describing processes that provide the same functionality are horizontally adjacent in the table. The first process is that of general data input, which has not been handled by all authors.

2.3 Scientific file formats

ViSC has many sources of data including simulations and analytical results, remote, medical and industrial imaging systems, data loggers and devices for scientific measurement, and user inputs from digitising devices. The fire hoses of information which McCormick et al. (1987) described have since become more abundant and more diverse. The needs of scientific users dictate flexibility and interoperability between these different sources and the software for processing, analysing and visualizing their output. This has resulted in countless standard interchange formats with ad-hoc designs that can be specific to every different scientific discipline.

A file format is a standard for data exchange but the reason behind different file formats comes from many sources including practical requirements, group interests and the need to support legacy software. As a result, some standards are less of a useful tool for the transmission of scientific data and more of a burden.

The next four sections review a small but important range of file formats for scientific data storage. Field specific, language-based, self-describing and user defined (non-standard)

file formats will be reviewed.

2.3.1 Field-specific file formats

There is a plethora of file formats which have become the de-facto standards in different scientific disciplines. Some have been designed to aid the transfer of a particular type of data, others offer encompassing support for all the different types of data in a particular field. This section will review several formats representing standards that are in common usage.

DEM The USGS Geo Survey Digital Elevation Model (DEM) stores terrain elevations for positions on the ground at regularly spaced intervals. DEMs are organized into three types of record. The first record contains all the metadata for the DEM, the second record, which will comprise the majority of the DEM data, contains individual profiles with header information and the third record contains all the accuracy information relating to the DEM. The USGS is currently undergoing conversion of all its DEM information into its new Spatial Data Transfer Standard (SDTS) format which provides a standard for transferring other GIS data types like vector lines and raster image data.

GRIB & BUFR The World Meteorological Organization (WMO) has developed two standard formats Berges (2002) for the transfer and exchange of meteorological data between different systems: Gridded Binary (GRIB) and Binary Universal Form or Representation (BUFR). GRIB stores regular gridded arrays of binary values and is used for the transmission of observational data such as air pressure and temperature. BUFR is a flexible format for archiving meteorological data and can be applied equally well to other scientific data. Another evolving standard, it defines a protocol for transmission of quantitative data. BUFR uses a unique Data Description Language (DDL) which is highly extensible and uses self-descriptive records.

mmCIF (Fitzgerald et al. 1993) is an evolution of the Crystallographic Information File (CIF) (Hall et al. 1991), which is a subset of the STAR (Self-defining Text Archive and Retrieval) format. CIF can store all forms of text and numeric data and was developed by a working party on crystallographic information in an effort sponsored by both the International Union of Crystallography (IUCr) Commission on Crystallographic Data and the IUCr Commission on Journals. This produced a data dictionary for archiving experiments and results, which was adopted by the field. In 1990 a working group expanded the dictionary to include macromolecular crystallographic data items, resulting in mmCIF.

FITS The Flexible Image Transport System (FITS) (Wells et al. 1981), originally developed near the end of the 1970s, was designed to enable the exchange of astronomical image data between different hardware platforms and so solve the problems caused by differences in the way primitive binary values were represented. In addition to this FITS was able to solve the problem of describing what sort of instruments were used to acquire the data and where they were directed to obtain the data. FITS has evolved to include other metadata new storage functionality like spanning, and a range of structure and syntax for defining astronomical information.

DICOM Digital Imaging Communications in Medicine (DICOM) is a format for the transfer of generalized medical images. Developed by a joint committee of the American College of Radiology (ACR) and the National Electrical Manufacturers Association (NEMA), it is a multi part standard to facilitate the interchange of imaging and associated medical information between different computer systems in a medical environment. It defines how values are encoded, structured and used alongside a host of different possible data elements which can be stored, their relationships and specifications for different types of medical imaging which can be described by this format.

These examples illustrate a few of the field-specific file formats which are widely used. As a result, these specific examples are usually supported by scientific visualization software, and while important they are not the primary target for this research. However, other less common field-specific files formats are a target for this research. The type of data such formats store and the different approaches to storing are illustrated by these examples.

2.3.2 Language-based file formats

The use of a language as a method of data description and storage has been applied in a range of file formats. VRML (Virtual Reality Mark-up Language), XML (eXtensible Mark-up Language), XDR (eXternal Data Representation by Sun Microsystems) and PostScript by Adobe Systems Inc are all notable examples of language-based formats and are described below.

VRML (Carey et al. 1997) was developed to support the access of interactive 3D virtual worlds and objects over the World Wide Web (WWW). VRML uses plain text description utilizing tags with a hierarchical scene graph to describe scenes and objects. It is a common import and export format for many 3D graphics packages because of its portability and platform independence.

XML (*Extensible Markup Language (XML) 1.0* 1998) has been designed to describe data and is an extensible platform independent mark-up language. XML files are plain text which when used in combination with either DTD or XML Schema can define self-describing data structures. XML comprises named elements enclosed in tags. Elements are placed around text values to give them a meaning and hierarchical relationship to the elements. There is a parent-child relationship between enclosing tags and their content, and as this can include other elements this allows hierarchical structures and many forms of data to be described. XML has two main problems as a storage medium, these are its verbose nature and openness to bad design practices.

XDR (Network Working Group 1987) is both a standard and API that defines what is transferred at the ISO presentation layer. It uses implicit typing and value representations at a binary level. This solves problems caused by different platforms having different byte ordering and byte alignments. XDR transfers as a language similar to C or Pascal, however it is a data description language (DDL) not a programming language. This form of data description is very flexible and enables the unambiguous definition of data structures and their content. XDR is supported through library routines, which encode and decode XDR data streams. The XDR standard defines portable binary interpretations and a DDL for transmission of structured data formats.

PostScript (Adobe Systems Incorporated et al. 1990) from Adobe is another language based file format. It is known as a 'page description' language because of its common use in specifying printed layout. PostScript files are scripts containing a sequence of commands that provide a rich command base in relation to defining graphical data and page layouts. PostScript is device independent, the language uses postfix notation and is stack-based. It defines graphical data with the target applications of printing and graphical document transfer. A complex language, PostScript is an industrial standard language that is now mostly run by the printers and software dealing which them, though human readable, most PostScript files are machine created from the source's graphical data.

Language-based file formats use programmatically defined structures to describe their data. As a result they have the potential to be unambiguous, human readable, self-describing and extensible although to provide access, software can need quite complex compilers or interpreters. The relative merits of these languages can be measured in terms of how well they are supported by their proponents and how easy it is to gain access to comprehensive library routines. They need to be recognised because many of these types of file format are field independent and used in a wide range of applications.

2.3.3 Self describing 'generic' file formats

The production of a single file format that can be used to exchange any type of scientific data has been a goal of the simulation community for some time. NASA's Common Data Format (CDF) is an example that has been in use since 1985. Self-describing file formats use abstracted data storage, access and manipulation routines in the form of libraries that can be linked to several different programming languages and are compiled under a wide range of platforms and operating systems. Software programmed with a CDF reader can then access any file saved in CDF regardless of the file's subject matter. CDF, Network Common Data Form (netCDF) and the Hierarchical Data Format (HDF) are notable examples of these formats and described below.

CDF was developed by NASA to unify the storage and manipulation of scientific data from a range of different disciplines. It achieves this through describing different datasets with a data dictionary that is stored alongside the data values. In this way, each file contains all the necessary semantics to be self-describing. CDF is implemented as an abstract interface with associated libraries; the format of CDF files is hidden from the programmer, with the interface providing access to all file content and the libraries carrying out all the file input and output. The CDF data model supports multidimensional gridded data (described on page 59) and either multivariate data on a shared grid or individual variables each with their own grid. A comprehensive description of CDF can be found by Goucher and Mathews (1994).

netCDF was originally developed to provide a common platform-independent interface between Unidata applications and real-time data sources. It employs several powerful concepts from CDF, including the abstract interface and library, in conjunction with 'XDR-like' platform-independent binary types. netCDF files store data in self-describing objects that can be accessed transparently through the library. netCDF offers a similar abstraction to that offered by graphical libraries. Simply put, it

defines both datatypes and valid access functions for these types. The underlying workings of netCDF and the actual storage of netCDF data are hidden from the user. netCDF also uses a standard file format which is implemented in a similar manner to XDR, implicitly specifying byte order and how data should be interpreted in netCDF files. Information on NetCDF can be found at (*Network Common Data Form* 2000).

HDF development was started by the National Center for Supercomputer Applications (NCSA) in 1988. HDF was to facilitate scientific data management by offering a standard and extensible method for scientific data transfer which was efficient whilst also supporting many platforms. HDF is implemented and supported by a data access library and API as well as a range of software tools. The file format is tag-based and supports multidimensional gridded data, multivariate datasets, raster image data, mesh data, spreadsheets, finite element data and sparse matrices. HDF information can be found at (*Hierarchical Data Format* 2000).

All these formats are widely used in scientific circles; they are often supported by MVEs and are not a primary target for this work. They are interesting because they each illustrate different ways of describing data. They offer a prescriptive solution to file input with their self-descriptive design, which is lacked by other file formats. Users who own or create software which outputs data in one of these formats will be able to load it into an MVE with ease. Any tool for data input will need to describe data using a similar range of attributes and structure definitions that these formats use to describe their data.

2.3.4 User-defined and non-standard file formats

Given the range of formats we have already seen and the countless others which exist, it can be seen that there are many good reasons for the use of an existing standard file format including:

- de-facto standards provide an easy route to portability and data exchange
- time and resources are released by not developing an 'in house' format
- developmental support, APIs and libraries may be available which reduce the cost of using a standard
- industrial bodies or field encourages and promotes the use of a standard.

However, there are also reasons which prevent the use of a standard and instead lead to a non-standard, native or user-defined file format:

- a standard is not widely enough advertised or adopted to encourage its use
- protection of commercial interests through the use of a proprietary standard
- the lack of a suitable standard for the user's data requirements (this is unlikely)
- a suitable standard exists but licensing fees may limit its uptake
- a standards use may be discouraged by an unwieldy API, an overly complex description or a storage intensive nature
- a lack of access to or control over the source software's output, or lack of access to those who could modify it.

As a result, many programs output a native, proprietary or closed format which is unpublished or rarely used. These formats, often designed by programmers, have a tendency to fall into one of three categories:

Text records are commonly used to store variables for scientific data. They usually comprise a plain text file, with a line which describes the variables using names like 'height', 'pressure' and 'speed'. This header line is then usually followed by lines

of values with the corresponding number of variables. While storage-intensive, this informal storage format is easy to produce, platform independent and relatively easy to support because it can be analysed so easily. Values in these files are either separated by spaces or control symbols such as commas. For large datasets, this format is generally too storage-intensive, nonetheless examples do exist of large datasets stored using text records.

Row and column files can offer storage for 2D data; a header usually defines how many rows r and columns c are in the block of data. What follows are r lines of separated values with c values per line. A simple but storage-intensive format for data storage. Many examples of 2D scalar datasets like height fields are stored in this format.

Raw binary files usually store an array as a contiguous block of data, sometimes preceded by a header. Essentially this format is usually just a copy of an array from memory, and as such is simple and much less storage-intensive than plain-text (although platform dependent). They can be very difficult to input unless all their parameters are known, such as the rank and shape of the array, and what sort of binary values it contains.

Some of these categories are supported by visualization tools which will be described in the next section. However, the ease with which they are input depends entirely on how much the user knows about their file and how easy it is to use the visualization system's input tool.

2.4 Visualization tools

This section will review several current systems supporting ViSC and what they offer the user in terms of file input support. The review has been compiled from direct usage experience where possible, the Advisory Group on Computer Graphics (AGOCG) Review of visualization systems Brodlie et al. (1995) and Scientific Visualization: Techniques and

Applications Brodlie et al. (1992b) in addition to the reference materials each package provides.

- AVS Express
- IRIS Explorer 5
- IBM Open Data Explorer OpenDX
- PV-Wave 8
- AMIRA
- VisiQuest (formerly Khoros)

2.4.1 AVS

Advanced Visual Systems (AVS), originally developed by Upson et al. (1989) is an MVE based on the dataflow model. It uses the unified field data model described in section 2.1.2 and has developed from a dataflow paradigm to an object-oriented model in the latest edition, AVS Express. AVS is an application builder, it enables the user to create visualization applications in either a visual workspace by connecting modules or with its internal V scripting system. AVS supports file input through the following techniques:

- a range of data import modules for different file formats including AVS's own range of native file formats;
- the AVS file input wizard, which detects filename extensions and proposes one of the above modules for inputting the data. It also enables the user to fill in any parameters they may need;
- the field file format, this format is implemented as user edited text files. Each file contains reserved words and parameters recognisable by AVS. These describe a

referenced file's content in the terms of the AVS field data structure. If the file contains data or datatypes which cannot be described by AVS field data structures then this solution will be ineffective for that particular problem. A more advanced version of the field file format has been developed by Manchester's International AVS Centre (IAC) to handle data which has a cell regular structure;

- ADIA is a tool for making field headers using a GUI from AVS 5, which is not in AVS Express. It produced field file format headers taking the user through step-by-step choices to specify their field data;
- file access objects are AVS Express functions which can be combined to create fields, no GUI, functions with archaic parameters to enter data;
- the file import tool produces an AVS file input solution, given a range of parameters;
- V scripts using file access objects to import data;
- making a new extension to AVS using C++.

2.4.2 IRIS Explorer

IRIS Explorer was originally developed by Silicon Graphics Indigo (SGI) and is currently owned by the Numerical Algorithms Group (NAG). The present version IRIS Explorer 5 is an MVE based on the dataflow model. It provides the user with the facilities to build applications and compile them into stand alone data visualizations, as well as offering effective collaborative visualization through a range of collaborative modules (Wood 1998). It offers the user several routes (dependent on platform) to input their file data.

- a range of data input modules for different file formats, including several modules which offer multiple input facilities for images;
- a parameterised text reader module for simple text file formats;

- DataScribe, a UNIX tool, can produce script files that map the users data into IRIS Explorer's data structures;
- the IRIS Explorer APIs offer easy access and manipulation of data for programming extensions to the system;
- QuickLat offers a simpler way of programming an extension for a reader module, making the internal data structures of IRIS explorer easier to access when programming using this external tool;
- Module builder also simplifies the process of programmed extension, enabling the user to build an interface and describe the parameters which are needed by a module, then link them with the code the user has written.

2.4.3 IBM Data eXplorer

IBM's Data eXplorer and the open source version OpenDX use both dataflow and the unified field data model. OpenDX offers a range of file input tools though its data prompter:

- the general array header file format. This much like AVS's field file format enables the user to describe their data using keywords and values which can specify file input parameters and metadata.
- Data prompter, a tool for inputting data. It creates a general array header through step-by-step interactions with the user, offering a simpler way of describing their file if is in a form describable by the general array header format
- HDF, NetCDF and CDF are all supported through the Data prompter file input tool
- a range of image file formats are also supported through this tool
- spreadsheets are also supported through a parameterised reader tool
- programmed extension is also supported in IBM OpenDX.

2.4.4 Amira Viz

The Department of Scientific Visualization of the Konrad-Zuse-Zentrum fur Informationstechnik Berlin (ZIB) has provided the newest MVE, Amira Viz, distributed by Template Graphics Inc (TGS). It is a package which specialises in image and volume datasets. It has support for non-standard external data using three methods. The first is a header file for defining stacked slices to form a volume dataset. The slice description file enables the user to specify each slice of a volume through a different file name, its position coordinate in the stacked axis and, overall, the pixel aspect ratio of all the images. It has a header file format called stacked-slices, which describes the files that compose a volume dataset. The parameters for a volume, including the x and y pixel size of the images and the spacing for each image is stored in addition to the names and locations of the slice files.

The second method Amira supports, is the input of raw binary data using a parameterized reader, which enables the file to be loaded by specifying a header size as well as and the bounding box dimensions and coordinates of up to a 3 dimensional array. Other parameters include the 'index order', which specifies whether the first dimension or the last dimension is varying fastest, the byte ordering, type of binary primitives in the array and number of variables.

Finally, it advocates the construction of own export or convert filter to produce AmiraMesh or HxSurfaces, which seems to indicate that the user can either modify their own software to output Amira compatible types or produce an extension to Amira.

2.4.5 Khoros

Khoros, originally developed by the University of New Mexico and now under the name of VisiQuest and owned by AccuSoft was originally an interactive image display package. VisiQuest has a range of reader modules and some parameterised tools for accessing simple ASCII and binary data files. Finally, data can be input through programming

an extension to VisiQuest.

2.4.6 PV-WAVE

PV-WAVE 8 by Visual Numerics is an array-oriented language for creating applications for visualization and data analysis. Users of PV-WAVE can enter commands from the keyboard which are immediately executed or write scripts that can be compiled and executed. It supports data input through a range of function calls. For unsupported file formats there are read and write routines in both binary and plain text. Finally, for particularly complex file formats there are low-level file access functions.

2.4.7 MVE file input summary

All the visualization systems reviewed in section 2.4 share common techniques for solving the problem of file input. They can be classified in one of six ways. These are:

- hard-coded readers
- header files
- script readers
- monolithic tools
- modular networks
- programmed extensions.

Each classification can lead to many solutions for file input problems. Hard-coded solutions are currently the most prolific and exist in every program which has to access any file-based data. The three latter solutions provide compatibility between scientific software and existing visualization systems. Table 2.2 highlights what each system supports in addition to programmed extension which they all support.

Visualization package	Approx. No. hard-coded readers	Header files	Script Readers	Monolithic tools	Modular networks
VisiQuest (Khoros)	29			Raw data tool	
PV-WAVE 8	13				
AMIRA	41	Slice description file		Raw data tool	
IRIS Explorer 5	25(18)†		yes	DataScribe & QuickLat	
AVS Express	41	Field format header	yes	Import Wizard & File import tool	File access ojbects
IBM Data eXplorer (OpenDx)	12	General array header		DataPrompter	

Table 2.2: Visualization software file input provisions other than programmed extension. †the number of modules which are used to produce this input

Hard-coded readers All visualization software provides functionality which allows users to input their files. The majority of these provisions can be described as hard-coded; they are extremely simple to use and work in an efficient yet inflexible manner. Only the location of the file is normally needed to input a dataset using a hard-coded solution. This will result in a successful input if the file format matches the format built into the solution. Any deviation in the file from that format and the solution will return an error message and fail to load any data.

Each solution is specific to the format chosen, with the result that hard-coded solutions are only provided for file formats that are industrial standards or widely used. There is little incentive for developers of visualization systems to produce hard-coded readers for less common file formats. Additionally, different versions of a file format can make its support an ongoing process, requiring updates to prevent the system's specific implementation from becoming incompatible with newer versions of the format. A cost is incurred for the developer in keeping input modules

up-to-date.

Most MVEs have hard-coded solutions packaged up into modules which can be placed into the working environment, thus usually placing the emphasis on the user to find the correct reader for the correct file format.

Header files enable the user to create a separate description of a file's content by providing metadata in a format that the MVE can read. A header file can either ignore, replace or supplement any metadata held in the file depending on its own structure and flexibility. In this respect, header description files are similar to DDLs. They are limited though by their ability to describe another file (or files) in terms of the MVE's own internal data structures and, if the file data cannot be described in these terms, then this type of solution will be inapplicable.

Scripted readers Script files can perform a similar, although more complex, task to a header file. Just as a header file can describe the file in terms of the application's data structure, a scripted reader can convert the data into the application's data structures. One example of such a reader can be found in IRIS Explorer running in Unix: the scripts are generated with an external tool and combine parameters with a mapping between file content and IRIS Explorer's data structures.

Monolithic tools These are provided with the MVE and are usually organised into one or more dialogues, which enable the user to specify a range of input options, providing the user with choices for the type of grid or connectivity, the variables and the dimensions of arrays. While they can be simple, this is not always the case. Additionally, they are inflexible and cannot easily be extended by the user to input files which are out of the scope of their parameters. Most MVEs have one of these, be it a simple raw data reader like that found in Amira Viz or a more complex tool like DataPrompter in IBM DX.

These tools are provided as part of the MVE package. They usually have a simple step by step interface which allows the user to choose the type of data in the file and

then fill in the parameters the tool needs to produce a solution. The solutions that these tools produce can range from directly inputting the file to the production of a reusable script or header file that enables the chosen file to be input again without using the tool.

Modular networks A modular solution utilises the MVE's workspace to build an input solution by wiring modules together. AVS is the only package to offer this form of solution and uses a range of different modules called 'file access objects' (*Core AVS/Express and the Object Manager* 2004) to import data from files. There are other modules in AVS called 'mappers', 'combiners' and 'extractors' which can manipulate this data into the AVS field data types. This method is much simpler than creating a file reader by programming since it only uses the same skills which are needed to produce a visualization in the environment. However, it is still more complex than using a hard-coded reader.

Programmed extensions All the MVEs reviewed provide the user with APIs and tools to extend them. Usually, a programming language like C or FORTRAN is used to implement new modules in an MVE that can then be connected into module networks providing user-defined functionality.

Extending a visualization system by programming a new module is the most complex route for users to input their data. To produce a new module successfully the user needs detailed knowledge about the data and the file format. Moreover, users need the skills to implement a module in the MVE, including familiarity with the data structures and APIs and a level of competency with the supported programming language.

Some MVEs simplify this process, providing external tools that encapsulate the user's code and provide simple access to the sections of the MVE data structures they need to use. One example is IRIS Explorer's QuickLat tool (*IRIS Explorer User's Guide (Windows NT/2000)* 2000). However, even with such tools, pro-

grammed extension remains a complex undertaking.

2.5 Discussion

This chapter has reviewed the relevant data models, process models, frameworks and classification systems for ViSC. The problem of data input deals with many data sources, different examples of file formats from such sources have been described and classified into four groups, 'field-specific', 'language-based', 'self-describing' and 'user-defined'.

The different visualization systems and particularly their data input techniques have also been reviewed. From this review six different techniques for data input have been identified in the present range of MVEs, these are, 'hard-coded', 'header files', 'scripted readers', 'monolithic tools', 'modular networks' and 'programmed extension'.

The user will often have data files of a field-specific or user-defined format. As a result, they will rarely be able to use a hard-coded technique to input their data, leaving the latter five MVE supported file input techniques. Each of these needs technical skills and knowledge to produce a solution. As a result, the task of entering data can be the hardest problem users face when attempting to visualize their data.

The next chapter will define the problem this research will aim to solve and its scope in terms of desired outcomes for the ViSC field and software components.

Chapter 3

Data input considerations

The previous chapters have described what is meant by data input in the context of scientific visualization; they have also defined the scope of this research. Now we are going to look further into the central difficulties of finding out how scientific data is stored and retrieved from storage.

3.1 Understanding file storage

Our interest with file storage lies not in the physical or low-level mechanisms of how or where files are stored, nor does our interest lie with how the operating system retrieves, transfers or acquires the contents of a file when one is opened for reading. The main interest lies in what is retrieved after these various mechanisms have been applied.

At a high level a file can be described as a block of memory. This block of memory contains whatever a program places into it, which is defined by the file creation routines used in the program. These routines are created by programmers and with this notion a file could conceivably contain anything; however, common sense and the nature of scientific data have led many programmers to similar solutions when storing different forms of data.

Continuing with the notion of a file as a block of memory, it can be seen that most systems import data in a two-step sequential process, firstly reading a specified value and then

placing it into a data structure. The sequential nature of most loading processes comes about as each value is read; when this happens, the pointer to the file content moves towards the end of what was read.

An example would be loading five floating-point values. Each value could be read in by individually incrementing the pointer in the file to the end of each value. Alternatively they could all be read in at once taking the pointer in the file to the end of all five values. Other data structures can be read in a similar way, by accessing their fields individually, or, if the file mirrors the memory layout of the data structure, they can be accessed as a single block of data.

The process of loading is usually dynamic; in the sense that some values at specific positions in a file can alter the way the rest of the file is read into a program's data structure. The utility of having such control, or descriptive, values can be seen as giving a file or file format greater flexibility, by parameterising some aspect of the input process.

These descriptive values, termed metadata, can be classified into one of two groups. The first group concerns content-oriented metadata, which describe the data content of the file. The second group concerns file-oriented metadata, which describe the file format or organisation of the file. Differentiation between content-oriented and file-oriented metadata is important, because the former describes the data within our target program, whereas the latter could be an important tool in deciphering its format.

Some examples of content-oriented metadata include the dimensions of a data set, such as the height and width of an image, the number of variables or fields in a record, a bounding box for the data, the physical measurements for separation between adjacent samples, the type of brick connectivity for finite element cell data and text headers for fields which describe their content.

Examples of file-oriented metadata include the positions of various blocks of data relative to the start of the file, the type of compression used, the version of the file, its creation

date, and the delimiters or run-length encoding tags used.

Since metadata is descriptive it is common for programmers to create systems that place metadata (of both forms) at the start of the file. The benefits of this approach when later reading the file include the ability to allocate memory dynamically for data structures without having to read the entire file first, and simplicity in the sense that there is only a single structure that predefines the rest of the file.

Metadata at the start of the file is referred to as a 'header'. In some file formats this header is in an entirely different file, leaving a data-only file to fulfil the majority of the storage requirements. The header-only system works for many kinds of data, but often due to their complexity, or just through different flexibility requirements, some formats place metadata in between data in the file.

One example of storing metadata in between blocks of data values could be for an aggregate dataset, where several different sets of data are held in one file, and so require a description for each individual block.

Metadata can sometimes be stored at the end of a file, in a similar manner to the header by using a 'footer'. This also allows a grouping of metadata to be separated out from the core of the file. However, unlike a header, a footer is only useful for controlling the loading process if the whole file is buffered first, although a footer could be a sensible place to store attribute values and other individual parameters that may affect the interpretation of the data.

3.1.1 Interpretation is everything

"Interpretation, the action of explaining the meaning of something" (Pearsall 1998b, OED) is exactly what data input is all about. All computer programs work through the use of interpretations, linking values with meanings; without knowing what numerical data means, how can it be used correctly?

The need for interpretation exists at many levels, from individual bytes to large arrays of complex records. Files are generic storage, and in loading a file we need the correct interpretation in order to retrieve the values as they were originally placed in the file. From our understanding of files so far we can break the problem of interpreting a file into three sections.

At the lowest level interpretation needs to take place for the bytes read from the file. This is because a single byte can have several meanings, which include it being part of a larger value representation, e.g. a 32-bit or 64-bit value, or on its own, being a signed or unsigned numeric value representation. Without this interpretation we can gather no values from the file, and hence are unable to read anything of use. Existing ways for loading data apply an interpretation to bytes by having the byte interpretations built-in to the software.

At a higher level interpretation is required to show the organisation of values within the file. This is in terms of what the connectivity, domain and range of the data is. This can be expressed by defining the possible meanings of an arrangement of values within a file.

Finally, at the highest level interpretation conveys the meaning of values, in terms of the signified such as colour, pressure, distance, time and density. This final interpretation may be necessary to describe data within the visualization system or specify a variable's name or a variable's type, e.g. that the following variable is a date or time and hence comprises several values.

3.1.2 Binary interpretations

When a file is read, blocks of bytes are interpreted as various binary values. Interpreting a block of bytes is usually done by copying the specified range of bytes from the file into a block of memory. This memory has been defined as containing a specific binary representation and labelled as a variable. In most programming languages there are sev-

eral standard types of value representation according to which one or more bytes can be interpreted.

All the languages examined in the course of this project have stored binary numeric values in one or more bytes. These include Fortran, Cobol, Pascal, C, C++ and Java. It can be conceded that there exist value representations of less than one byte, however the great majority of programs and file formats will deal with values at a byte level. There are also subtle differences between the programming languages, in terms of the precision and storage of certain binary types. However, for the purposes of this research they can be generalised as part of one problem, and that is the correct identification of the type representation for a block of bytes. The question we seek to answer is not *what* are the choices available for storing a binary value; the answer to this question is both definite and finite. The question is, rather, *which* of the choices to use as a representation

Further complications of binary interpretation include the 'endian' of the hardware or software that writes a file. The term 'endian' *The Jargon Dictionary : Terms : The M Terms : middle-endian* (2003) describes the sequence in which bytes are stored and interpreted. Big endian is used to describe storage of bytes in a 'most significant byte first' fashion, and little endian is with the least significant byte first. There is also the little used variant of middle endian, where the significance of the bytes is not in a linear order and is neither ascending nor descending. The implication is that if we take a numeric value from a file stored in an endian different from that of our current system, the interpretation of all the multi-byte numeric values held in the file will be incorrect.

Related to the problems of type and endian interpretation is byte alignment. In interpreting an array of bytes into an array of multi-byte values, we need to consider where to start. In hard-coded loading solutions the start of an array of values is either known because of other previously encountered data stating where it is sequentially, or is at a point which, when other data leading up to it are read in, will leave the file open at the position where the values are stored. For this work there is no such specified start point and so it is

both possible and likely that there will be other file content preceding the block of values which needs to be interpreted. When this occurs the start point for the values sought will neither be clear, nor likely to be aligned by chance with the start of the first value. Any misalignment from the start will either miss values from the desired set, or cause incorrect interpretations for multi-byte values and all those following, as shown in figure 3.1. In figure 3.1 the bytes we are trying to interpret are labelled *B*, the effect of using the correct start point for the interpretation and hence the correct alignment is labelled *A*, whilst the effect of an incorrect alignment is shown in *C*. The dark grey byte value is out of the range of the dataset, and could be any value that followed the block we want to interpret. A similar effect occurs when the interpretation begins before the correct start instead of after it.

Finally values can directly correspond to some form of symbolic system like text characters. This can indicate a text-based representation of numeric data values or just nominal data values; this is discussed in the next section.

To summarise, all the data we wish to extract from a file into values will require interpretation. Furthermore for multi-byte values a correct alignment and endian needs to be specified. Finally, whether the values are text in nature or not needs determining.

3.1.3 Text interpretations

The alternative to storing data in binary form is to convert the values into text and store them in a text-based file format. The benefits of storing values as text include removing problems caused by the type of endian, byte alignment or binary primitive type interpretation. Text values can also be interpreted in a manner which is hardware independent.

We still need to know that the file contains text though, and many such files are not labelled as text, so the first question is: does this file contain text? The answer to this can be found either by direct examination by the user, i.e. looking at it represented as text,

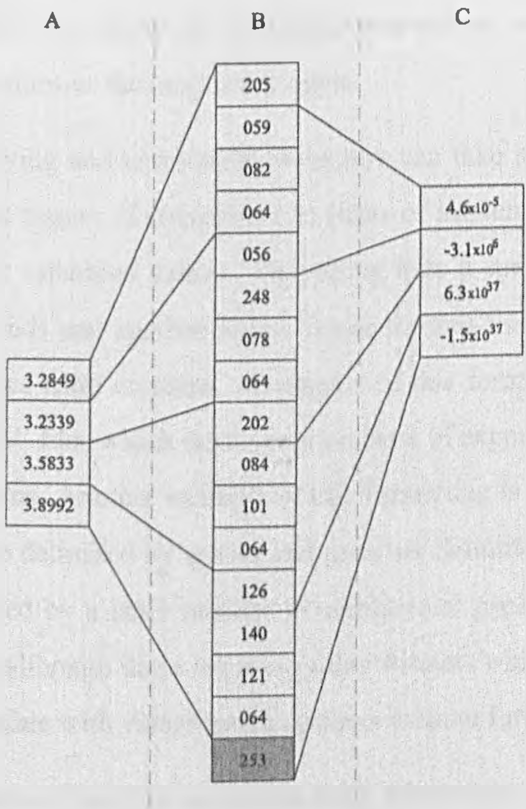


Figure 3.1: Byte alignment; a) a set of four floating point values, b) the byte values which make up the floating point values, c) the effect of incorrectly aligning the array by one byte

or by a test to see if the majority of its byte values contain numbers within the range of characters and symbols used for text storage.

In a text-based file each byte encodes a single character; these can be combined to produce larger strings of characters which may pertain to the values for which we are looking and by parsing these strings we can find entire values. If these values are numeric they will need to be converted into a binary primitive type in order to be used within the target visualization system.

It is important to note that data encoded as text is far simpler to comprehend because it is stored in a human-readable form and has, as a result, values clearly separated by spaces,

line returns, or symbols etc. However, ambiguity remains as to the level of precision required to represent values or the range of integers.

The process of identifying and combining characters can take several different routes, each having a different degree of complexity in terms of implementation. The simplest form uses delimited or separated values. Delimiting uses a specific character to identify where one value ends and another starts. Some formats use multiple delimiters to separate records or rows from columns. Examples of this formatting include comma-separated value or 'CSV' files which are a common form of export format in spreadsheet and database applications. Another example of this formatting is a row and column output, where columns are delimited by spaces and rows are delimited by new lines. These specific formats are used by a large number of commercial packages, as well as many scientific applications, although there are many other formats which, while similarly delimited, are not compatible with visualization systems without further description.

The more complex forms of text file format are those which use contextual tags, such as SGML, HTML, XML, TEX and MSI. The characteristic of these formats is that sequences of characters can define reserved words which can alter the meaning of the values held near or inside pairs of tags. Also in this type of file there are variable assignments, where a named variable is set equal to a value literally, written out as e.g. "WIDTH=200" or "DENSITY=0.005e-03".

Finally, an attempt can be made to extract values from text files without knowing how they are formatted by using rule-based parsing. Using this technique we can define some basic rules which can identify primitive numeric and text data types. These rules are then applied to separate out sequences of characters into their respective types. These strings of characters are then converted into binary data types if they represent numeric data. Once again, the levels of complexity and functionality in such systems can vary a great deal.

Delimiting and rule-based parsing of text files will form the main thrust of this research.

3.1.4 Structural interpretations

Structural interpretation is concerned with how data values are related. All datasets comprise one or more nodes which in turn contain one or more variables. Most datasets have connections between these nodes. The structure of a dataset describes the connections present between the nodes and so defines their respective neighbours. In describing structure we need to look at how these connections can be specified and encoded into sequences of values.

Many solutions to the problem of structural interpretation take into account the notion of physical positions and, as visualization deals with physical phenomena, this is a reasonable assumption. However, this section assumes that positions are just another variable of the dataset, with the emphasis being on how nodes are connected to other nodes.

Connectivity

First datasets will be classified as having four possible types of connectivity: these are *scattered*, *gridded*, *cell regular*, and *cell variable*. There are two concepts by which each of these classifications is defined, that is the way the connections are defined between nodes and the way the arrangement of these connections forms cells.

Scattered data has no connections present between nodes in the dataset. Scattered data nodes have no neighbours. Any techniques for rendering or processing which require some connectivity information will require it to be generated using a process such as triangulation.

Cell variable data has no consistent pattern of connections between its nodes, thus requiring explicit definition of each connection and then how these connections are built up into cells. The key notion here is that cells may be of different types; this means that the number of nodes and nodal connections per cell needs to be speci-

fied. An example could be a series of polygons, where there is a different number of sides for each polygon.

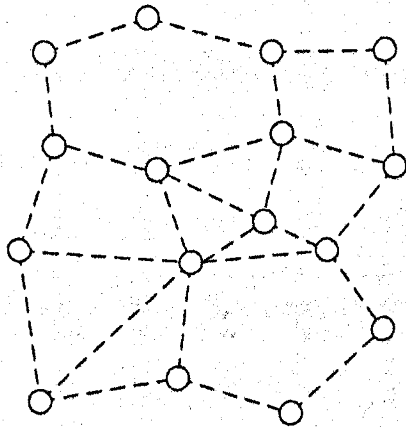


Figure 3.2: An example of a structure exhibiting cell variable connectivity

Cell regular data implies each cells connections. All nodes are connected into lines, triangles, quadrilaterals, hexahedrons or some other n-node cell. The key notion compared with cell variable connectivity is that all the cells are of the same type.

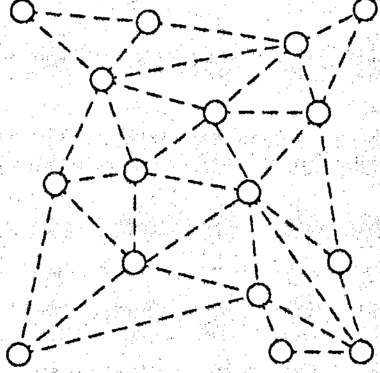


Figure 3.3: An example of a structure exhibiting cell regular connectivity, in this case having triangular cells

For each group of n nodes there needs to be a definition of the connections which form the cell. This is usually achieved by stating nodal values in the order which they need to be placed into each cell. Some examples of commonly used cells are illustrated in table 3.1.

No. Nodes	Example connections
2	
3	
4	
5	
6	
7	
8	
9	
27	

Table 3.1: Fourteen n-node cells which are commonly used for deriving connections in cell regular and cell variable data

Gridded data implies the connections for each node of the dataset. It is the most compact form for storing connection information. It also has the most stringent requirements for regularity in the type of information stored. Linear connections between nodes are assumed along each of m linearly independent axes over which the data is defined. This results in $2m$ connections per interior node to neighbouring nodes in each axis.

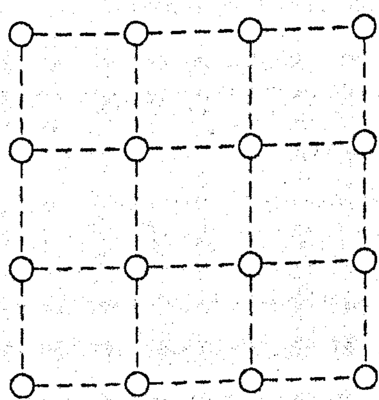


Figure 3.4: An example of a gridded connection structure

Compact array storage

An important consideration when attempting to determine the structure of a dataset is the use of compact data representations. Many compact representations use array indices to imply more information about data held within an array. This section will look at examples of such representations and then illustrate their use in relation to this work.

Arrays hold a number of elements which can be referenced using indices. These indices can be used to create a compact representation for a dataset. Uniform rectangular grids are prime cases where arrays are used to provide compact data storage. A uniform rectangular grid has both gridded connectivity and regularly spaced coordinates for the positions of each node in the dataset as illustrated in figure 3.5. In a uniform rectangular grid the connections between the gridded nodes of the dataset can be implied using the indices of the array. Connections in a uniform rectangular grid are formed between array elements

with adjacent indices. As well as a compact description of the connections in a uniform rectangular grid, the positional regularity is used to remove the need to explicitly state coordinates for every node. Instead, functions are used to generate the coordinates for each node. An example of such a function is shown in equation 3.1; in this function the coordinate x is generated using a minimum and maximum value for x which are min_x and max_x as well as the number of data points along the x axis described by dim_x and the location of the node along the x-axis as defined by its index i in the array. A common example of data using uniform rectangular grids can be found in digital elevation models, where an array of height values are stored as a 2D array, with the physical coordinates for each node produced from a single fixed point and two spacing measurements.

Figure 3.5: Uniform rectangular positions in two dimensions. The difference between coordinate values for each dimension is constant

$$x = f(min_x, max_x, i, dim_x) = min_x + i \cdot \frac{max_x - min_x}{dim_x - 1} \tag{3.1}$$

Uniform rectangular grids are a special case of variable rectangular grids and so are not the only example of a gridded data structure which uses a compact form to store coordinates. A variable rectangular grid has the same gridded structure as a uniform rectangular grid but has a variable cell size. The coordinate data held in a variable rectangular grid has each node at a given position along each axis sharing a common coordinate value as illustrated in figure 3.6. In a variable rectangular grid each dimension requires a vector of

Figure 3.6: Variable rectangular positions in two dimensions. Coordinate values are shared for all nodes at the same intersection along each axis

explicit coordinate values; these coordinate values are constant over all other dimensions of the dataset. The common way to implement this type of scheme is to use the data array indices to refer to the vectors of values which store coordinate positions for each axis. Variable rectangular grids are used in many forms of simulation, including MHD (magneto-hydrodynamics) and CFD (computational flow dynamics).

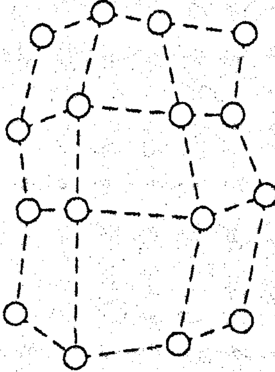


Figure 3.7: Body fitted positions in two dimensions. Coordinate values are different in each dimension and for all other coordinates in other dimensions

Finally body fitted grids store gridded data whose coordinates can have a different value for every node in the grid. An example of such a structure is illustrated in figure 3.7. Because each node can have different coordinates they require storage per node in the same manner as dependent variables. These types of grid are often used in the aerospace

$Array(i,j,k)$	$X_{uniform\ rectangular}$	$X_{variable\ rectangular}$	$X_{body\ fitted}$
5, 5, 5	2	5	125
10, 10, 10	2	10	1000
20, 20, 20	2	20	8000

Table 3.2: Illustrating the storage required for coordinate X when using different rectangular and body fitted grids

industry for work involving aerodynamic surface simulations.

The benefit of using these different data structures for storing coordinates is outlined in table 3.2. This table shows the number of values required to hold a single coordinate value X for each of uniform rectangular, variable rectangular and body fitted grids of different sizes.

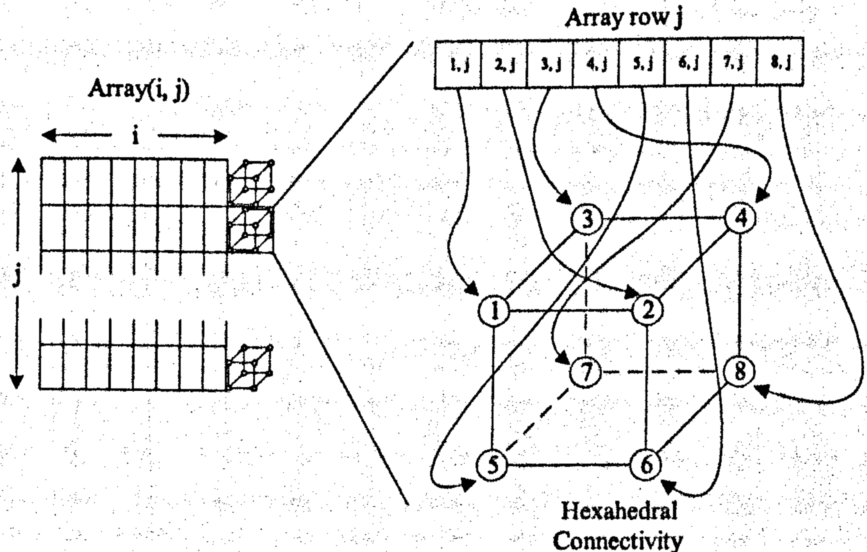


Figure 3.8: Array(i, j) stores nodal positions in its columns and individual cells in its rows.

Further compact representations based on array structure arise with cell regular data. In these datasets the connections are unlike those in gridded data because the number of connections per node is variable. However, there is regularity in these structures because each cell has the same number and configuration of connections which allows the data

array indices to imply connections. If we take a cell definition and number the nodal positions in the cell from $1 \ldots n$ then in the data array we can use one of the indices to imply these nodal positions. Individual cells are distinguished by using another index. An example for hexahedral cells is illustrated in figure 3.8.

The last two structures, cell variable and scattered, do not benefit from compact representations as gridded and cell regular structures do: cell variable structures have less regularity to use for implying nodal connectivity and scattered data has no structure to encode. Cell variable data can be described in many ways including the use of cell regular primitives like lines, triangles and quadrilaterals to construct more complex cells like polygons, prisms and bricks or it can be described by using a dictionary of pre-defined cells which are then repeatedly referred to. All cell regular data requires additional information to describe individual cells, and this can be implied or explicit in a dataset.

Each of these compact representations can be achieved for variables and structure which exhibit certain regularities. From these examples we can determine that structural information does not need to be explicitly stated for every connection in the dataset; nodal connections and the make-up of a cell can be determined when assumptions about how data is connected are written into the loading system. Equally, variables do not require to be stated for every node in the dataset if they also exhibit regularity which allows for them to be reduced in a similar manner and then implied for every node in the dataset.

Recognising the existence of these space-saving methods and their use not only in visualization data structures but also in file formats, may allow the development of algorithms and mechanisms by which they can be handled. So to summarise, an array index can imply a nodal position in an n-node cell or a nodal position in a gridded dataset and it can, with additional values and interpretation, imply one or more variables that are in some way dependent on the index value. This is illustrated in table 3.3.

	i_0	i_1	i_2	i_3
j_0	1.2	2.3	4.6	8.5
j_1	1.2	2.3	4.6	8.5
j_2	1.2	2.3	4.6	8.5
j_3	1.2	2.3	4.6	8.5

(a) Normal representation

	i_0	i_1	i_2	i_3
j_n	1.2	2.3	4.6	8.5

(b) Compact representation

Table 3.3: Table 3.3(a) A 4 by 4 Array containing variable X which, given any location in i is constant for all locations at j. Table 3.3(b) illustrates a compact representation of the same data.

Array metadata

Now that we know how arrays can be used to describe different structures, we move on to describe the metadata which defines an array. There are two items of metadata which are required to correctly interpret an array: its rank and shape. The rank is a single value which corresponds with the number of linearly independent indices over which the data ranges. The shape of an array is a list of n values which specify the maximum value of each index. A rank n array has $n!$ interpretations of the order in which its dimensions are laid out in the file.

To identify the rank and shape of an array of values we need to know if multiple variables are stored in a single array. If so, the rank of the overall dataset is incremented by one to provide a variable index which has an extent corresponding to the number of variables within the array. Any variable at any point in the dataset should then be accessible by the use of $n + 1$ indices.

The different effects of interpreting the shape of an array can be seen in figure 3.9; the diagram illustrates one dataset with five variables. Three of the variables are schematically illustrated in the figures by white, grey and dark grey colouring; these three are dependent upon the remaining two independent variables, the 'height' and 'width' dimensions of the array. Overall, we find a total of three dimensions for the array (the height, width and

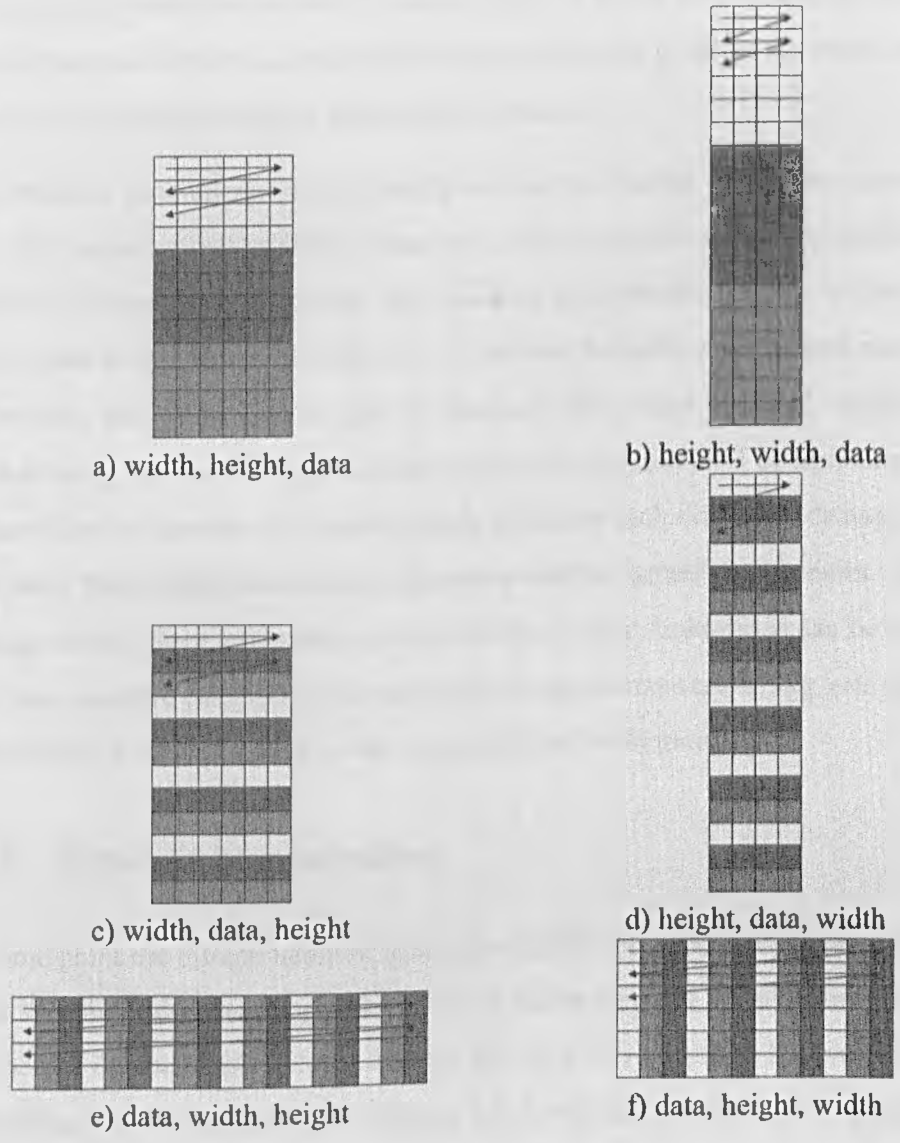


Figure 3.9: The 3! possible ways of interpreting the shape of an array containing three variables over two dimensions

number of dependent variables), which gives us 3! possibilities for the order in which the dimensions can be interpreted.

In interpreting structure we have two aims: firstly, to define the metadata for arrays of values and hence describe the data's actual storage; secondly, to define the nodal connectivity involved and thus describe the data's actual structure.

The results of such interpretations should be a series of arrays which have scattered, gridded, cell regular or cell variable connectivity. For scattered connectivity nothing more is required; for gridded connectivity only the array metadata needs to be defined correctly. Cell regular arrays need to have the type of cell and the index to node relationship defined before they can be used. This type of structure often refers to nodal variables held in another array (described in next section). However for structural interpretation we need to know how to generate the connections required for each cell in the dataset, and hence only need think about how these connections can be formed at this point. As for cell variable arrays, there are numerous ways in which their connections can be determined. This data can have cells formed by references to other arrays containing cell regular data, references to arrays of nodes, or dictionary style cell definitions.

3.1.5 Semantic interpretations

At some point the interpretation of data from the file ceases to be a problem for the data input system, and becomes a problem for the filter stage of the visualization pipeline. The border for this changeover is blurred, but must be recognised lest we attempt to re-implement or redesign existing techniques for manipulating data prior to visualization.

Semantic interpretation for data input occurs at this border and describes the processing of data which has been correctly interpreted from the file. It is required because of two main problems. First, the structure of the data in the file is unlikely to mirror directly that of the visualization system. Second, the values in the file may not have a valid interpretation

within the visualization system.

Certain locations within a visualization system's data structures are designed with the purpose of holding certain types of data, a common example being coordinate data for nodal positions. Equally some of the processes in visualization systems employ common interpretations for certain types of data, for example, image data in IRIS Explorer is recognised as RGB triplets. These standards may differ from those used in the source program which produced the file and so we find two basic requirements for semantic interpretation, firstly to place variables which have a particular meaning in the correct part of each data structure, and secondly to offer conversions and transformations which allow data not directly supported by the visualization system to be used.

The next three subsections will outline examples of semantic interpretations and conversions necessary to enable a visualization system to use a particular kind of data. First is the compatibility of primitive types, the second is concerning visualization variables and the third is concerning indirection variables.

Compatibility of primitive types

Previous sections have predominantly dealt with finding and describing the content within files. With this knowledge we still have a crucial problem; it is not whether we can decode the file into an equivalent data structure, but whether we can store and use this decoded data. Interpretations may result in data which, though correctly extracted, cannot be used directly by the rest of the system; such data will require conversion.

If we cannot support a file's primitive binary value representation, then a safe conversion must be provided to force the values into something that the visualization system can use. For example, if a particular visualization system cannot manipulate 16-bit unsigned integers then an appropriate conversion, into a type which it can manipulate, will be required.

Concerning visualization variables

Not all types of data have a direct visual mapping. Variables without such a mapping are abstract and can be mapped to any compatible abstract visualization object. However those variables which do have a visual meaning can require additional interpretation for their use in a visualization system. Three examples of visualization variables follow which describe some of the problems which can occur:

Coordinates represent a location in physical space, in one to three dimensions. These could be in the Cartesian coordinate system or another axial system such as polar, cylindrical or toroidal. In both IRIS Explorer and IBM Data Explorer there are specially allocated sections of the data structures that contain coordinate information. Axial systems that are not supported will require a safe conversion to one that is, for example, converting polar coordinates to Cartesian coordinates.

Glyphs are used to represent data comprising multiple values per node, which alter some visual property of a graphical object. One example is two dimensional directional vector arrows, which use two variables, or one variable consisting of two components, to set the direction of the arrows.

An issue arises when using glyphs due to the order in which the variables are interpreted. If the visualization system's interpretation differs from the storage in the file then the variables will need to be reordered. Linked with this issue is a need to separate the variables which are to be rendered using glyphs from any others in the dataset. Equally the processes which generate glyphs often require only a single array containing the necessary variables. Overall, visualizing data using glyphs requires the ability to group, separate and reorder variables in arrays.

Colour data represents an image or texture using one or more colour channels. Using colour presents two difficulties. First there are many different colour models, and hence many different meanings can be assigned to the variables in a colour tuple.

Several examples include: greyscale; red, green, blue (RGB); hue, saturation, value (HSV); cyan, yellow, magenta, black (CYMK); red, green, blue, alpha (RGBA). If the colour model is not supported by the visualization system, then the colour values will require conversion. For example, a HSV triplet to an RGB triplet. The second problem relates to the way that the colour channels are stored in an array. An example is the triplets of a Windows bitmap picture (BMP), which uses the RGB colour model but stores the values in reverse order from blue to red, resulting in a requirement for visualization software to reorder the values.

Three major requirements can be extracted from this discussion of visualization variables. These are: the ability to extract, group and reorder the arrangement of variables in an array; conversions for colour and coordinate data into compatible forms; the ability to allocate variables to a particular section of the system's data structures.

Visualization systems also contain variables which are used to link nodal values in other arrays together. These 'indirection variables' are described in the next section.

Concerning indirection variables

Not all that is extracted from a file contains measured values relevant to the data – some provides a useful way of compressing or structuring the data. Reference data and identifier values are two examples. Values in reference data represent nodes whose actual values are stored in another array or location. This location is indexed by the reference value and so can be found if the source data is known. Cell variable and cell regular data structures are often stored using this method, with nodal data held separately from connection information. Colour table images are another example where reference data is used to reduce the amount of storage needed by storing references to colours in an array which has gridded connectivity.

Identifier data is another example of indirection data. Instead of referring to tuples or

x	y	z
x_0	y_0	z_0
x_1	y_1	z_1
x_2	y_2	z_2
$\vdots$	$\vdots$	$\vdots$
x_n	y_n	z_n

a)

id	x	y	z
23	x_{23}	y_{23}	z_{23}
42	x_{42}	y_{42}	z_{42}
10	x_{10}	y_{10}	z_{10}
$\vdots$	$\vdots$	$\vdots$	$\vdots$
id_n	x_{id}	y_{id}	z_{id}

b)

Figure 3.10: Arrays illustrating the absence (a) and presence (b) of identifier data

records by an offset index from the start of the array, an identifier variable can explicitly state the tuple's order or give a unique point of reference in an array. Identifier data is not used for the purposes of compression, and has utility often based upon the notion of a unique identifier within the source software. This type of variable is illustrated in table 3.10 where coordinate values x, y and z are listed a) without and b) with identifier data.

A common use of identifier data can be found in file formats storing cell-regular and cell-variable data. Finite element data is just one example of an application field that uses identifier data. References to specify individual cells are held in another array or data structure. These can refer to either an index value in table a) or an identifier in table b).

In a similar manner connectivity data can also contain identifier data. This can allow cells constructed from nodes to, in turn, be used to construct more complex cell structures. For example, hexahedral cells can be constructed using six references to quadrilateral cells.

3.2 The role of user knowledge in data input

The most complex and necessary series of interactions which most visualization system users face is importing their data. There needs to be a way of enabling the user to find a solution, because no generic, automatic solution exists. Users have an important role to play in the problem of data input. For current solutions they have to choose the correct parameters and program the correct function in order to load their data. In the system we

propose they will not *require* such detailed knowledge, but any they do have can be of value and utilised to speed up the process of data input. In order to see the value of user knowledge we need to look at what they know, in terms of technical and application-based experience.

A user's knowledge can be categorised by looking at how they have worked with the data, source software and target visualization system. At this high level we cannot directly describe a user's technical skills or knowledge but their time working with the data does give us insight into what sources of information they may possess from their experiences.

Visualization system experience
None User Expert

Source software experience
None User Expert

Experience with data
Viewed Worked with

Table 3.4: User knowledge of a file input problem

Table 3.4 illustrates the different categories and levels of experience a user can have when attempting to solve a file input problem. The first category is experience with the visualization system into which they will be importing the file. If they have no experience or are just a user then they may not be able to customise the system to load their data. The result is that they will be unable to use the system for their data without help from an expert or visioneer. At this level we need to provide non-technical tools for the description of data, as the user may not share terminology commonly used for describing the data they want to load. Alternatively, expert experience in this area means that the user is capable of altering the system to meet their own requirements. In this case we should try to simplify the task of input so that it takes less time and fewer resources.

The second category of user knowledge covers their experience with the source software that produced the file. With no experience of the software, the user will have no knowledge of the file format used except for clues such as the file extension or what others might have told them. Users of the source software may have an idea what sort of data is used

and may also have knowledge about the way in which the data needs to be processed. The author of the software should be able to specify in detail the data structures used and the format of the file.

The last category of user knowledge covers their experience with the data itself. This is the most effective source of knowledge for our input problem as it is not usually specific to a particular program, and does not require the user to have worked with any particular software. Thus it is a general source of information and may be more commonly available than the others so far considered. If the user has ever viewed the data in a numeric form, they may be able to describe maxima or minima of the dataset, or the number of variables, and types of values involved (floating point or integer). Moreover if they have ever seen it visualized they may know its dimensionality or be able to correct its interpretation if this is in error. If the user has analysed the data they may have knowledge about the number of variables, what they measure and their structure or type of structure in some field specific terminology.

If the user has all the details about their data, the file and the target system then the problem of file input is how to specify these details in a swift and simple manner. If they lack knowledge about aspects of their data the file and the system, then for each of these knowledge deficits they will have to discover the values and interpretations they require.

3.3 Summary

In this chapter we have described in detail the problems faced when we attempt to interpret a file. The data models illustrated in chapter 2 present the structural and semantic attributes of scientific datasets. They focus on the data requirements for presenting visual information and, therefore, the data structures relevant to producing a display. They do not focus on the low-level representation of these structures or file storage needs. This chapter has taken such descriptions and illustrated how they relate to file storage; it has

also shown the different interpretations needed to represent values in file storage. From this we have defined several key notions of how files are organised:

- A file's content can only be used after a series of interpretations and transformations;
- Files contain many values which need to be interpreted. These values can be data or they can be file- or content-oriented metadata. Data comprises the variables that need to be stored for a scientific dataset. Content-oriented metadata describes or controls the description of the data. File-oriented metadata describes or controls the description of the file;
- File content can be interpreted at three levels
- The first set of interpretations describe the binary and plain text interpretations needed to represent individual values;
- The second set of interpretations are the structural interpretations. They show how the different types of connectivity, defined by the data models from section 2.1, can be represented using arrays of values;
- The third set of interpretations bridge the gap between file content and visualization data structures. They address the need to place variables, as defined by the data models in section 2.1, into areas of the MVE data structures that reflect their meaning.

In addition to these key notions, this chapter has described the importance of user knowledge in the file input process. Section 3.2 outlines the different levels of knowledge that a user may have about their file input problem and how this affects the complexity of forming a solution.

Chapter 4

A new approach to file input

This chapter will present a new interactive approach for solving file input problems. This approach will be supported by a model and architecture for file input, which will define the necessary interpretations required to solve such problems.

4.1 Approach principle

The approach we propose should be as flexible as existing scripting and programming methods. It should allow interpretation to be parameterised where required and be simple and consistent in usage with the rest of the target visualization system. A core part of this approach will be to unpick the process of importing data into a more detailed model of how data is retrieved and then stored within visualization systems.

The common requirement among loading systems has been identified in section 3.1.1 as specifying the right interpretation. In the previous chapter, the user's knowledge was highlighted as a valuable asset in solving file input problems. The user's participation will always be needed as either part of forming a solution or verifying one. By focusing on the high-level data aspects of a problem, the user's knowledge may be more easily applied to checking the interpretation used for a file and specifying its content.

A key problem in the creation of file input solutions, is that some information about the

way data is stored may be missing. In these cases, a methodical examination of different file input parameters is needed. If we use the notion of 'forensically' examining of a file, by taking what is known about its content and then trialling different parameters and interpretations to gain insight into any missing information, this could lead to a better specification of the file's content.

In such an examination there will need to be tools to discover (or rediscover) these values. Feedback is an essential part of the way this approach will tackle the problem. The ability to provide a user with interactive or near interactive visual results from parameter changes should enable them to trial solutions in a timely manner. Also, in providing the right kind of feedback we may be able to speed up the process of importing a file and help the user to find errors in their file input solutions.

Our approach is to abstract away from describing different types of file format, instead pursuing solutions for different types of data. The aim is to produce a widely applicable solution for file input problems and encompass a wide range of problems. Taking a data-oriented point of view, many file formats can be seen as containing the same data constructs; it is this which will allow the design and creation of a general solution to the file input problem. The use of appropriate feedback and parameter trialling is another important part of this new approach. By enabling the user to apply their knowledge to the problem, and discover missing specifications for their file content, some file input problems may be greatly simplified.

The next section will describe a dataflow model which provides a high level description of the relationship between interpretations in the file input process. Following the model a software architecture will be described which defines the low level processes by which the model may be implemented.

4.2 The file input dataflow model

Current solutions to the problem of data input for scientific visualization have no common thread, no standard way to break down the problem, and no specific way of importing a file's content into the application's data structures. Because scientific visualization has never had a specific or well described model for the storage or retrieval of data, many *ad hoc* approaches have been used instead. This in turn has resulted in users having to program and script their input solutions. Each instance of a solution is very similar in functionality to the next, implemented in a similar manner, but with just enough inflexibility to be useless for anything else.

Our contribution to this field is a dataflow model for the file input process that typifies a systematic decomposition of the problem into three distinct stages that allow us to classify both the general process for each stage and the data communicated to the next stage. With this model a more generic approach to solving file input problems can be adopted. The model is illustrated in figure 4.1, with the value, structural and semantic interpretation stages each having the type of data they require flowing between them. Value interpretation interprets bytes from the file into usable values. Structural interpretation takes these values and both describes their connectivity and forms them into arrays. Semantic interpretation provides support for modifying arrays into meaningful structures and for converting the values into a form which can be used by the target visualization system.

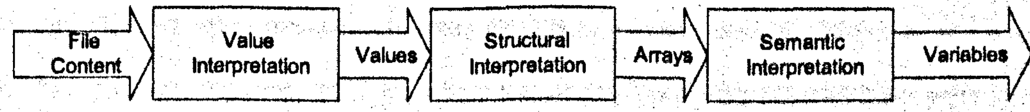


Figure 4.1: The dataflow model for file input

The first of the three stages, the value interpretation stage, takes raw file content and converts it into values. It does this by interpreting the data into a machine-usable binary form, e.g. converting four bytes into a floating point value, or converting the ASCII

sequence '3402' into a 16-bit integer binary value.

The next stage provides a structural description for the values passed to it. This structure is defined in terms of nodal connectivity and the arrangement of values into arrays. Nodal connectivity is necessary for defining where values lie in the domain of the dataset, and the definition of arrays determines where values lie within data structures.

Finally the semantic stage allows the grouping, extraction, and referencing of data as well as conversion of variables to allow them to enter the filter stage of the dataflow pipeline. An example might be identifying three variables as being components of a three dimensional vector, or identifying a block of values as depth data instead of height data for an array of values describing bathymetry.

4.3 A software architecture for file input

Looking in more detail at how each of these stages breaks down provides a set of interconnecting processes which enable the description of different file formats.

4.3.1 The value interpretation stage

As we discussed in section 3.1.1 interpretations are the key to this model, and the value stage provides the interpretations outlined in section 3.1.2, namely the conversion of raw file content from bytes into useful binary values.

The input to this stage is a sequence of bytes of a specified length (the length of the file), which is essentially a buffer for the complete contents of the file. The stage outputs values, either as individual values or as sequences of values. These can then either be used in the visualization software or further interpreted in a subsequent stage. For the most basic input problems the output from this stage may require no further interpretation. For example, any data which is a single variable in one dimension could be loaded at this

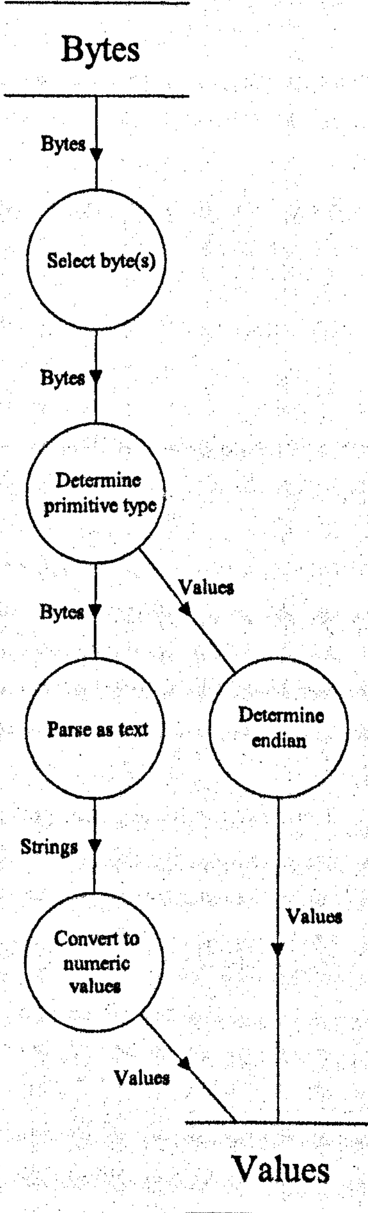


Figure 4.2: The value stage architecture

stage, and if the variable was in a form usable to the visualization system, i.e. did not require conversion, then no further interpretation is necessary. The interpretation of this type of data could be visually verified when rendered as a graph or viewed as text.

The first process in this stage is a selection process. Making several selections allows multiple *interpretation pipelines* to be made. The utility of such multiple pipelines will become apparent later.

The question still remains as to which primitive type is held in the selected portion of data; the next process in figure 4.2 interprets a block of bytes as a specific binary primitive type. The result of this process is a sequence of numeric values, including 8-bit values which are capable of having a text as well as a numeric interpretation. From this process there are two possibilities for interpreting the resulting values; the first is to continue using them as binary numeric values, the second is to interpret them as text.

In order to correctly interpret multiple-byte binary primitive types their endian must be determined. There are two common choices: little- and big-endian and several other rarely used byte orderings; all could be implemented but the first two are required.

Alternatively, interpreting the bytes as text requires two processes. First the character values need to be parsed into separate strings and then the next process down converts these strings into binary numeric value representations.

The output from this stage comprises both individual values and sequences of values. These can be used or visualized in a manner which allows the parameter choices for value interpretations to be verified. This verification can be done either by comparing the data values produced at the end of the value interpretation stage with known values in the file or by graphical evaluation using a histogram or graph.

4.3.2 The structural interpretation stage

This stage allows us to re-describe sequences of values as arrays of values. Moreover, it enables us to recreate nodal connectivity for the data in these arrays. It enables multi-variate and multi-dimensional data to be represented within the visualization system by specifying a description of its structure. Once again, a selection process can be used to

separate contiguous blocks of values for different interpretations.

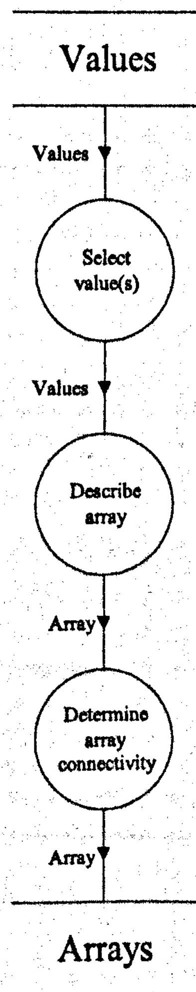


Figure 4.3: The structural stage architecture

After values have been selected by the 'Select value(s)' process in figure 4.3 they have two types of interpretation imposed upon them. The first is applied by the 'Describe Array' process which places the values in a rank n array by associating array metadata with the selected values. Next, the 'Determine Array Connectivity' process allows an array to be described in terms of scattered, cell variable, cell regular or gridded connectivity. The output from this process consists of arrays of values with some form of connectivity relationship.

4.3.3 The semantic interpretation stage

This stage is an interface between the arrays of values produced by the structural interpretation stage and the data structures of the target visualization system. It deals with the restructuring and conversion of arrays coming from the previous stage and determines potential relationships between them. The results of this stage are output as variables that can be visualized.

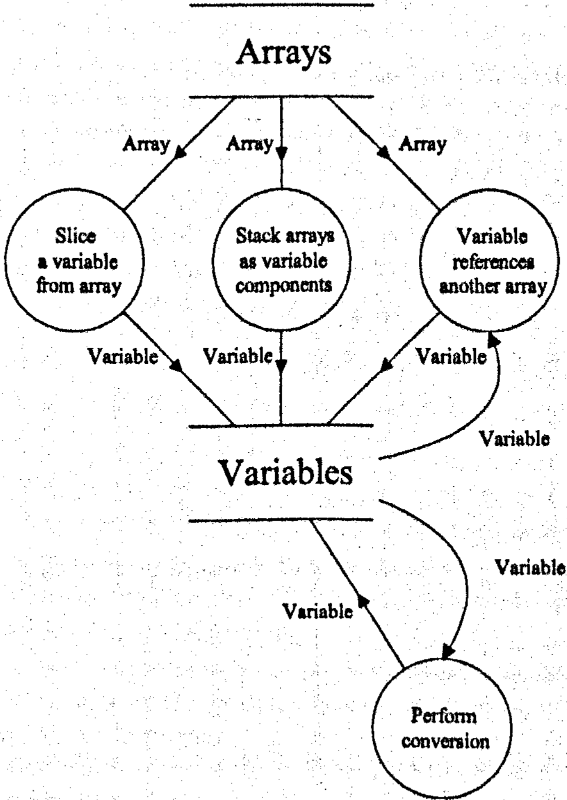


Figure 4.4: The semantic stage architecture

In this stage multiple arrays of values can be sliced and stacked in a different order to produce a form that is useful to the system. These processes are illustrated in figure 4.4. Reference variables are interpreted here, with variables in one array referring to either indices or variables in another array. Finally, conversions are applied to variables which are not directly usable by the target visualization system. Examples include variables

identified as requiring normalisation or processing into a different axial system such as transforming polar coordinates to Cartesian coordinates.

4.3.4 File input parameters

Each stage of the architecture has processes that transform data and the user's control over these processes is dictated by the parameters each one requires to describe their respective transformations. These file input parameters include metadata and file control values needed to make the data usable; they can be described as either explicit or implicit.

Explicit file input parameters are found in the file and comprise metadata and control parameters which have been stored alongside the data to enhance the flexibility of the file format. They describe aspects of this data which may differ between files of the same format. Examples of explicit file input parameters often include array dimensions, like the height and width of an image file, physical bounding boxes and slice separation in CT datasets.

Implicit file input parameters for the purposes of this research can be regarded as file input parameters which cannot be found in the file, instead they are usually codified into input software. Implicit file input parameters can include the number of colour channels in an image file format, the binary primitive type used for a variable and the locations of different data items within a file format.

4.4 Summary

This chapter has described a new approach to producing file input solutions. To support this approach a model of file input has been put forward that illustrates the interpretations required to input a file (outlined in chapter 3).

The requirements for producing these interpretations have been described in terms of the

processes and metadata necessary to parameterise the process of file input. The notion that metadata can be built into the reader and found in the file or user has been posed, and the notion that the user can prove an effective source for metadata has been further explored.

The file input dataflow model has been presented as an effective way of breaking down the problem along with a flexible software architecture for visually programming file input solutions. The benefits of such a system include a flexibility which is similar to that of programmed or scripted solutions, and some of the simplicity and definitive user choices for interpretation which can be found with monolithic input solutions.

The next chapter describes an implementation of the architecture, named the "Interactive File Input Toolkit" (IFIT), which allows the solution for many of these file input problems to be described.

Chapter 5

Using the Interactive File Input Toolkit (IFIT)

The previous chapter discussed the problems that are involved in describing data and accessing file-stored data; it also described an interpretation-based model for data input that described the necessary interpretations to convert file content into useful data structures within an MVE. The file input architecture following the model describes the processes that are required to interpret a wide range of file formats using a data-oriented view of file input. This chapter will first outline a new approach to producing solutions to file input problems before presenting a software toolkit for solving file input problems named IFIT.

5.1 Forensic file examination

In chapter 3, user knowledge was highlighted as a valuable asset in solving file input problems. User participation will always be needed as either part of creating a solution or verifying one. The amount users know about their file formats and data sets affects the complexity of creating and verifying a solution. The problem is that some information about the way their data is stored may be missing; the notion of 'forensic' examination attempts to tackle the problem by developing tools that can be used to discover (or rediscover) this information. Once captured, the information can then be applied in the creation

or validation of a file input solution. Examining a file forensically involves taking what is known about the file and data it contains and applying it in an investigation where the file is tested to discover what input parameters are needed and what they are. This is done by using a combination of traditional file analysis techniques and iteratively trialling parameter values using visual tools. This approach for discovering and then extracting stored data is technically possible with some existing tools and through programmed extension. However, current solutions can hamper the user, firstly with the time taken to set up each trial and secondly by their lack of appropriate feedback to analyse the outcome.

Facilitating forensic examination requires that the user is provided, where possible, with adequate feedback and interactions which will allow the rapid trialling of different input parameters. Performance improvements in comprehending data and parameter change by using interactive feedback can be seen in the field of computational steering (Johnson and Parker 1994). In this field, simulations are bi-directionally linked to a visualization allowing the user to change parameters by interacting with the visualization. This real-time feedback allows the user to see the effects of different parameter changes and test different ideas much more quickly and intuitively than with a batch mode simulation. Linking similar visual feedback mechanisms to insightful visualizations of file data at different stages in the data input model would facilitate forensic analysis by allowing users to trial what they know and discover what they lack.

A tool to solve file input problems forensically requires new and informative views of file storage in addition to those from existing diagnostic and examination tools used by visualization experts. Binary files currently present the greatest problem, as the standard means of examining their content involves the user visually inspecting them with either a text or hexadecimal display tool. While useful to those who know what to expect and are seeking verification, these views are largely meaningless to most users and offer little in most cases.

5.2 Final requirements for a file input tool

The final requirements for a file input tool are described in the three sections below; they are broken down into requirements for the user, output, functionality and implementation.

5.2.1 User requirements

The first task any MVE user faces before they can visualize their data is to load it. As a result this aspect of an MVE's usability has important ramifications for all users. The following requirements relate to the needs of MVE file input tool users.

Consistency of interface The GUI and usage of this tool must be consistent Nielsen (1993) with those found elsewhere in the MVE.

Unambiguous terminology Terminology used in the interface must avoid using words which are overloaded with many field-specific meanings that could cause problems for users from a particular field.

Clear feedback and Outputs The effect of parameter choices should be illustrated to the user by an appropriate feedback mechanism. Feedback should always be available to the user regardless of the stage they are at in the problem solving process. Outputs should allow validation testing to be performed upon any solution and errors to be traced to a particular stage of the file input process.

Transparency and accuracy The user must be made aware of any changes that have been made to the raw data values during the file input process. Equally, operations which modify data values should do so in a manner which preserves their accuracy. The importance of providing the user with an accurate picture of what has been done to their data up to the point of rendering is crucial for retaining scientific accuracy and integrity in the way any rendered output is interpreted.

5.2.2 Output requirements

IRIS Explorer is to be used as the primary MVE platform for demonstrating this file input tool. Outputs from this tool must use IRIS Explorer's core data types. They must also be compatible with IRIS Explorer's conventions for interpreting different types of data. The following output requirements mirror the data constraints of IRIS Explorer.

Data structures IRIS Explorer has three core data structures of interest to this work: lattices, pyramids and parameters. Any tool must output these to be of use in the environment. Lattices are the main storage medium for data values; they offer the ability to hold multidimensional, multivariate gridded and scattered data, provided the variables are all of the same binary primitive type. Storing multiple variables of different binary primitive types requires multiple lattices to be used. The pyramid data type (The 2000) uses lattices to store nodal data which is then referred to by connection data to store cell-regular and cell-variable data. Pyramid cell-based data only allows index offset references as opposed to identifier reference data for connections. Finally IRIS Explorer supports generalised parameter values.

Common data interpretations The fixed interpretations which IRIS Explorer expects for graphical data are in the following forms:

- colour image pixels are described using RGB triplets,
- coordinates are described using the Cartesian axial system with one to three components in an XYZ ordering,
- vectors can be described using one to three components in a XYZ ordering.

Binary value restrictions IRIS Explorer supports 8, 16 and 32-bit signed binary integer values in addition to 32 and 64-bit floating point values in both the lattice and pyramid data types. This means that unsigned binary integer data and other binary

value interpretations will require conversion or casting into an appropriate binary primitive type.

5.2.3 Functional requirements

The scope of the project described in chapter 1.6 and analysis of existing systems from chapter 2, along with experience of producing file input software, produces the following required functionality for an MVE's file input system.

Broad applicability The software needs to offer a way of producing file input solutions for a wide range of MVE users who have non-standard file formats which cannot be input in a trivial manner by using the existing tools.

Support for generic data operations A dataset may be spread over multiple files; conversely, many datasets may be condensed together into a single file. The file input architecture illustrates all the processes necessary to input file data; any implementation must offer all the minimum functionality outlined in the architecture.

Extensible The software needs to offer the ability to extend the system to allow maintenance and development. This enables files which present a unique challenge to the system to be handled using a combination of IFIT and programmed extension.

Reusable The ability to produce solutions that enable the user to load files of the same format which have subtly different input parameter values is required. These solutions should automatically take values from the file (if they exist) to set parameters needed in the input process. A complete solution which can be reused should be a possible output of the system. Other partial solutions are then possible for problems that, due to a lack of user knowledge or limitations of the tool, may still prove useful to the user and perhaps provide a staging point for extension.

Interactive feedback Appropriate visual feedback needs to be generated for the user.

This feedback should be linked to the underlying file input parameters using direct image manipulation (Chatzinikos and Wright 2003).

5.2.4 Implementation requirements

A key aim of the implementation is to prove the utility of the architecture and dataflow model for file input. It should also illustrate how effective the forensic approach is to producing file input solutions and the worth of visual feedback in file input problems.

Several simplifications can be made with respect to implementing a solution based upon these general aims:

- The tool will only support the binary primitive types that are available in standard C. These types are fairly comprehensive but it must be acknowledged that others exist, examples include binary coded decimal numbers and fixed point binary numbers. For a complete solution, other binary primitive types would need to be identified and be developed as part of the toolkit's interpretations;
- Arrays that contain multiple binary primitive types can use only their first or last dimension to index the different variables in an array. This is another reasonable assumption for the majority of input files, and likewise, could be developed further after this project.

As IRIS Explorer is the test environment for this project, its data types and their interpretations have been identified as a desirable part of IFIT's output. Also IRIS Explorer's extension mechanisms must be taken into consideration. These include Schema scripting, a module and data API and the ability to extend the system using FORTRAN, C or C++ programming.

5.3 An overview of IFIT

IFIT is a collection of modules that extend the file input facilities of IRIS Explorer. It is implemented using C and C++ with both OpenGL and the IRIS Explorer data access API, as an extension to IRIS Explorer's module library. IFIT allows the construction of module networks which are capable of interpreting file content into usable data. Unlike many existing systems IFIT leaves the monolithic and ad-hoc approaches to file input with a modular design. The inherent flexibility of modular networks and several new visualizations enable the user to forensically examine files, allowing them to investigate and verify different interpretation parameters in real-time.

IFIT modules can be placed into one of the three following groups: transformation, specification and visual interaction. This section will briefly describe each of the groups in more detail as well as illustrating each module's usage and location in the file input architecture.

5.3.1 Transformation of user data

Transformation modules implement the core of the file input architecture, converting file content to values, arrays and then variables. Table 5.1 shows both IFIT extensions and existing IRIS Explorer modules which implement the transformation sections of the file input architecture. Modules are located in the table with respect to their inputs and outputs, these correspond to the data transferred between the stages of the dataflow model for file input.

The location of IRIS Explorer's modules in this table is a result of its existing data manipulation functionality which focuses upon arrays and variables. These would be the normal output from its original file input software and modules. The ability to handle arrays and variables is needed in the filter stage of the visualization pipeline. At this level the difference between the variable stage of the file input process and the filter stage of

the visualization input pipeline becomes blurred.

IFIT's transformation modules fill the positions outlined in the dataflow input model for which IRIS Explorer lacks an implementation. IFIT also provides modules that allow the parametric description of variables and address the user's need to produce more complex structures to describe their data. These modules are described in the next section.

Inputs	Outputs			
	Bytes	Values	Arrays	Variables
File location	ReadRawBinary			
Bytes	SelectBytes SearchSelectBytes	BytesToValues TextToValues TextRecordToValues		
Values		SelectValues	ChangeDimLat SelectValues	
Arrays			StackDimLat SliceDimLat CropLat†	DimToVar ChannelMerge†
Variables			ChannelSelect†	Mixer† SphereToCartes† MultiChannelSelect†

Table 5.1: Transformation modules in IFIT and IRIS Explorer; those marked † are provided with IRIS Explorer. The table shows the input and output of each module by its location.

5.3.2 Specification modules

IFIT has several modules which combine or add information to arrays and hence do not transform the data but specify additional attributes for its interpretation. Table 5.2 shows specification modules and their main inputs and outputs and linked usage. ComposePyr takes an array of connections and an array of nodal data to create cell-regular or cell-variable structures. IFIT extends this module's functionality with VarIdentifierMap, which takes both arrays and converts the connection data from name-based to index-based references. The ComposePyr module can then use these references to define connectivity

in conjunction with nodal data. Alternatively IRIS Explorer's Trangulate2D or Triangulate3D modules can be used with just nodal data to produce a connected cell-regular structure.

SetUniformCoords and SetCurvCoords both provide the user with a way of specifying the physical coordinates for a gridded array. SetUniformCoords provides three different ways of specifying coordinates for a bounding box: minimum per dimension with either the maximum, range or sample spacing defining the bounds in each dimension. This enables the user to use metadata directly for defining the bounds of their data in any of these three forms. SetCurvCoords enables the user to combine an array of data values and an array of coordinate values to produce data nodes with individual coordinate values for body fitted, cell-regular or scattered data.

Inputs	Outputs	
	Gridded data	Cell regular data
Scattered nodes		Triangulate2D† Triangulate3D†
Gridded array	SetUniformCoords	
Gridded array and Coordinate array	SetCurvCoords	
Scattered nodes array and array of index references		ComposePyr†
Scattered nodes array and array of named references		ComposePyr† VarIdentifierMap

Table 5.2: Specification modules in IFIT and IRIS Explorer; those marked † are provided with IRIS Explorer. This table shows which modules are needed to specify the two types of output data given the set of available inputs.

5.3.3 Visual feedback modules

IFIT contains three modules which allow the forensic discovery and verification of file input parameters by using interactive feedback. The first of these modules is called TextView and provides a window that shows byte values interpreted as plain text. It has

several options for interpreting standard end-of-line characters and can show the effects of using different delimiters to separate values in the file. Its main benefit compared with an external text editor is that, as part of the environment, it can be more closely integrated with IFIT modules. TextView is a useful tool for dissecting a file format: however, if the file contains binary data then all it can do is verify that fact because the output will appear largely meaningless. In this case there would normally be little recourse for a user, unless they had either produced the software which had output the file or had a detailed description of its content. IFIT solves this problem by providing ImageView and VolumeView. Both are modules that can produce a view of a file's content regardless of how it is stored.

ImageView generates a greyscale image of the values with which it is provided. The user interacts with the image using mouse drag actions to change its width; this activity is in real-time and forces the given values to be interpreted as a new image with the new width. The visual effect is both simple and powerful allowing 1D arrays held in the file to be located and, for 2D arrays, their dimensions to be found. Moreover, the artefacts which are generated by incorrect interpretations can be highly effective in identifying which interpretation parameters are wrong or what else is in the viewed area of the file.

VolumeView extends the functionality of ImageView into 3D, presenting the user with three views and allowing them to find the shape of a $rank$ 3 array. Each of VolumeView's three views present a slice through the array using a different axis. Three views alone cannot convey the content of the entire array, so animation has been provided to allow the user to have a better view animation. The animation pushes each slice through its respective axis in the array, preventing arrays with large areas containing zero values from causing the user problems because they have no information and so present no useful view. The animation also provides additional feedback that can guide the user toward the correct dimensions of a $rank$ 3 array.

5.4 Using the transformation and specification facilities of IFIT

This section describes how IFIT can be used to solve file input problems. First, IFIT's general usage will be discussed, followed by examples of simple module networks for extracting different types of data from files.

Loading any file using IFIT requires the construction of an appropriate module network; this involves placing modules into the MVE work area and then setting their parameters and 'wiring' them together. Wiring modules is a simple process of clicking on the output button of a module and then on the input button of another module to create a pipeline between those modules which transfers data. An IFIT module network can extract single values and arrays of values from a file; once the required values and arrays have been obtained they can be used to describe other aspects of the dataset, combined or visualized. The resulting topology and choice of modules in any network is dependent upon the number, structure and content of files that are to be input, in addition to the intended usage of the solution. The way in which a network is constructed depends upon how much of the file's content the user requires in order to visualize their dataset, and the user's knowledge about their data and file storage.

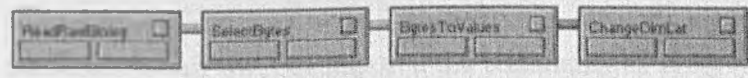


Figure 5.1: A simple IFIT example

For example, figure 5.1 shows a network of four IFIT modules that input a file containing a header followed by a single array of binary values. The first module in this network is ReadRawBinary which accesses the file and produces a cache of byte values. The next module linked to ReadRawBinary's output is a SelectBytes module which selects the data portion from the file. Following this a BytesToValues module interprets the selection into a *rank* 1 array of values. The last module, connected to BytesToValues' output, is

ChangeDimLat which specifies the rank and shape of the array; it also assigns a bounding box with the same shape as the array which permits the array to be rendered. The output from ChangeDimLat can be rendered or manipulated with the many tools at the user's disposal in IRIS Explorer.

The network in figure 5.1 can be used to input one array with a maximum dimensionality of *rank* 9, which is the limit in the user interface in ChangeDimLat. To describe this array six parameters are required. These describe the location and content of the array with an additional two to ten parameters needed to describe the rank and shape of the array. It illustrates the type of module network needed to input a file containing only implicit file input parameters (See section 4.3.4 on page 82 for a description of both implicit and explicit file input parameters). Explicit input parameters can dramatically affect the complexity of solutions made using IFIT, because each one requires a set of interpretations, which for the majority of cases are not shared with other items of data held in the file.

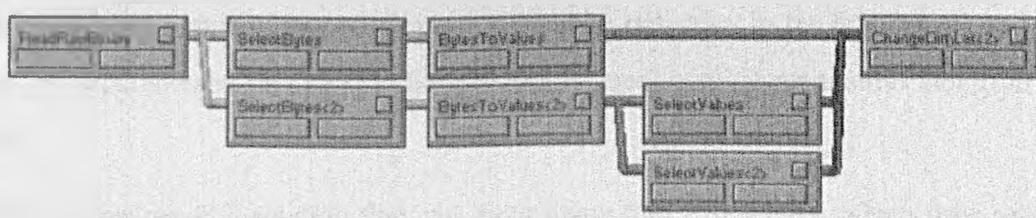


Figure 5.2: A module network that can import a file containing an array and two explicit parameters which define its shape

The module network in figure 5.2 illustrates how explicit parameters can be used to specify an array using IFIT and how they affect the complexity of a solution. In the example a binary file stores a *rank* 2 array and two parameters, that describe the array's dimensions. The solution interprets both the parameter values as dimensions for the array, and then interprets the array and sets its shape using the two parameters, which are wired into ChangeDimLat. The first module in this network is ReadRawBinary which accesses the file and produces a cache of byte values. The next two modules, linked to ReadRawBi-

nary's output, are both SelectBytes modules which select the data portions from the file. These are followed by two BytesToValues modules which interpret each selection into a differently typed *rank* 1 array of values. At this point the way the values are interpreted differs; the lower route has two SelectValues modules which take the array of two header values and select individual values to become parameters. The upper route connects to a ChangeDimLat which re-dimensions the array using the two parameters from the lower route, finally outputting a *rank* 2 array of values.

Complete file interpretations may not be necessary for any given problem; a file may contain data and parameters that are not needed by the user. A file may contain information which, though needed, can be specified by the user instead of being described in IFIT. This choice is dependent upon what the user needs from the file in order to visualize their data, and whether the user intends to make a solution that allows the repeated input of the file and others of a similar format. Where the need for access is one-time only, it is easier for the user to identify their data, extract it and then leave the rest of the file uninterpreted. This results in a much less complex solution and a reduction in the number of modules and input parameters that need to be specified to input explicit parameters stored in the file.

When a user needs a solution that can input many files of a format which uses several explicit parameters to store necessary metadata, each explicit parameter will have to be interpreted and wired into the network, otherwise the user will have to specify each one on a per-file basis. If the time taken to accommodate a parameter is less than the time taken to specify it for the range of files which will be input, then the choice is obvious. The choice is harder when less is known about the file format, as a search through the portions of the file which have been left uninterpreted will be required to locate and describe any explicit parameters which have been discovered.

Array parameters can easily be discovered using ImageViewer and VolumeViewer. However if the user is attempting to extract values from portions of the file containing mixed

binary values or other binary structures, they face a much harder problem. The user needs to know either where and how they are stored in the file or the actual target value that they are seeking. Without either of these the problem of extracting values from mixed binary structures in a file becomes intractable.

If the implicit and explicit file input parameters are known they can be manually entered by the user into the appropriate stage of a solution. If not, they will then need to be discovered using IFIT's visual tools. Once the explicit parameter values which are needed by the user have been discovered, their location in the file will need to be found and the values described using IFIT. These parameters can then be wired into the appropriate module. The effect of this is to acquire the parameter from each file which conforms to the user's IFIT network description.

Each value or array we intend to acquire from a file requires an interpretation pipeline to be created. All pipelines flow from at least one ReadRawBinary module, which caches the whole file content into memory. Each pipeline allows a different interpretation to be made of a selection of the file's content. The next section will discuss how IFIT modules can be combined to create networks that can input a user's data.

Simple value interpretation

Extracting either an array or a single value from a file requires both a location and value representation to be specified. The location can be specified with one of two modules, SelectBytes or SearchSelectBytes. These provide the ability to select portions of the file and allow the creation of multiple independent interpretation pipelines for different sections of a file. For example, the file header may require five different single-value interpretations and the data portion of the file just one, either way a total of six different interpretations will be needed to get the respective values out and hence six different selections to interpret.

SelectBytes offers several different parameters, illustrated in figure 5.3, for selecting the

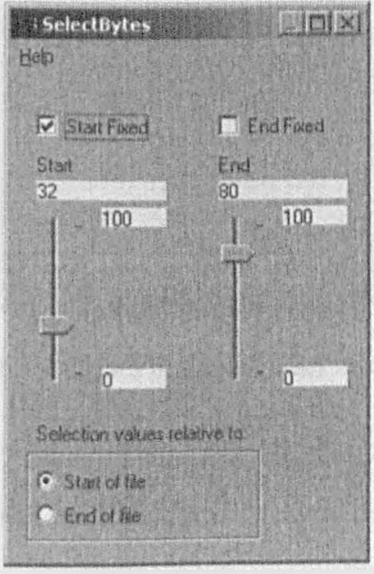


Figure 5.3: SelectBytes' user interface;.

file's content. The selected area is inclusive of bytes from the 'selection start' to the 'selection end'. Both ends of the selection can be set to 'fixed' or left free to extend to the size of the file on changes in the module's input. Finally, there is a parameter which selects from which end of the file the selection's start and end are measured.

These parameters have been chosen to give the user the greatest scope for producing a selection which does not require parameters to be taken from the file. They increase the likelihood that the user can produce a selection which does not become invalid with different files of the same format. SearchSelectBytes does a similar job, although it is more flexible to change in both the size and position of a selection than SelectBytes. This is because it uses user-defined search strings to determine the location of an item. Thus, an item's position within the file becomes less of an issue. SearchSelectBytes can output a single value after matching a single tag, which caters for assignment statements e.g. "$width = 108$" or it can output a series of values between a pair of tags e.g. "$vectors \ldots end$".

When a portion of the raw file has been selected, the next step is to choose an appropriate module for the selection's value representation and connect it to the selection module's

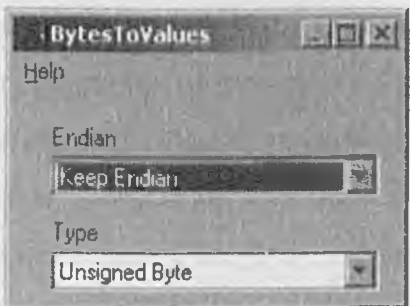


Figure 5.4: BytesToValues' user interface.

output. There are three IFIT modules that convert file content into values: BytesToValues, TextToValues and TextRecordToValues. BytesToValues, illustrated in figure 5.4, interprets a sequence of bytes as binary values. TextToValues delimits and then parses text value representations as does TextRecordToValues. TextToValues' parameters are illustrated in figure 5.5. It can input values separated by delimiting characters and values which are stored using a fixed number of characters per value.

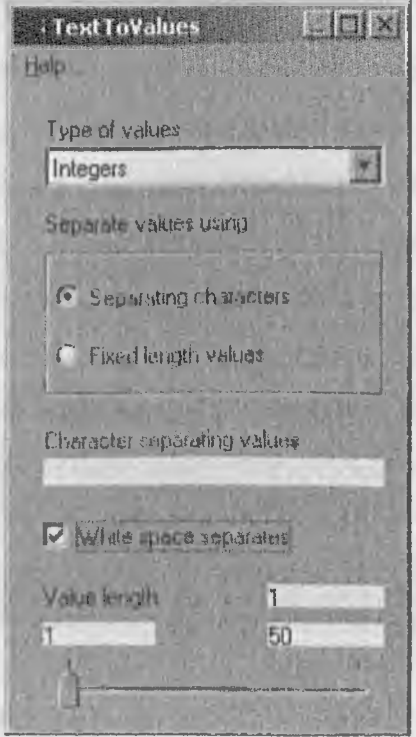


Figure 5.5: TextToValues' user interface

Finally, TextRecordToValues caters for plain text arrays which have different types of values. Its user interface is not illustrated as it is very similar to TextToValues with three additional parameters to specify the number of values per record, which value is to be extracted and whether the record stores values adjacently in the array using the first dimension as the variable index or separated using the last dimension of the array. At this point the dimensions of the array are not known, as it has not been interpreted. However, if the record length is known then these two arrangements of values can be found irrespective of the array's dimensions.

The output from all these modules can be used in IRIS Explorer. Once interpreted, another selection at the value level is offered by SelectValues, which has a similar interface to SelectBytes with the addition of a single value or array output from which to choose. Figures 5.6 to 5.8 illustrate three simple module networks that extract individual values from a file for use as parameters. Figures 5.9 to 5.11 illustrate three other similarly simple module networks that can extract arrays from a file.

Figure 5.6 shows the network needed to extract a single value, held as a binary primitive type, from a file. It uses the SelectValues module, to take a single value from BytesToValues' output and turn it into a parameter. SelectValues makes a selection with either user entered or wired in start and end parameters. It can output either a single value or a range of values, and, like SelectBytes, its selection bounds can be fixed or free to accommodate different sized array bounds. Figure 5.7 shows how three adjacent values of the same binary type can be extracted and used as parameters, and finally, figure 5.8 shows how three non-adjacent values or three values with different binary types can be extracted from a file.

Figure 5.6: A module network that can import a single parameter from a file

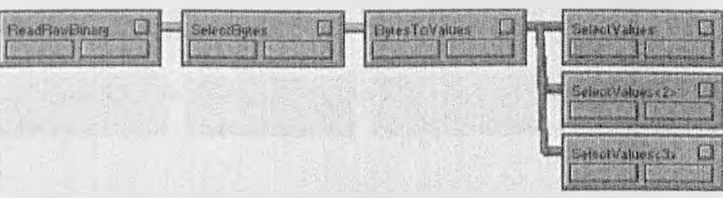


Figure 5.7: A module network that can import three adjacent parameters from a file

Figure 5.8: A module network that can import three non-adjacent or differently typed parameters from a file

Array manipulation and structural interpretation

The architecture for the structural stage defines how values extracted from a file can be interpreted as arrays that have a defined connectivity. In addition, the semantic stage architecture defines how these arrays can be combined or sliced into variables, as well as offering further aids to define their connectivity. This section will highlight modules that deal with describing and manipulating array structures and producing different types of connectivity.

Figure 5.9 shows the network needed to extract a single array of binary values from a file. The output from BytesToValues is wired to ChangeDimLat, which reinterprets the output from BytesToValues from a $rank$ 1 array to a $rank$ n array (where $1 \leq n \leq 9$) with user specified parameters. ChangeDimLat uses a static interface with 9 shape parameters although its algorithm holds for any shape of array.

Figure 5.10 shows the same functionality illustrated in figure 5.9 but for a file which uses plain text to store numeric values. Finally figure 5.11 shows how TextRecordToValues is wired into a module network to extract three variables from an array containing several

different types of numeric value.

Figure 5.9: A module network that can extract an array of binary values from a file

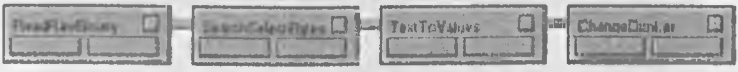


Figure 5.10: A module network that can extract an array of text values from a file

Figure 5.11: A module network that handles an array of differently typed text values extracting three for usage

Any sequence of values extracted from a file using the modules outlined above can be used as a one dimensional univariate dataset with either gridded or scattered connectivity.

IFIT's ChangeDimLat module was illustrated in the previous section; it allows the user to set the rank and shape of an array as illustrated in figure 5.12. In addition it produces a coordinate bounding box which allows the array to be visualized as a univariate gridded dataset. ChangeDimLat prevents impossible settings from being applied to an array, i.e. those settings which would extend the array's size beyond the number of values it actually has. It also warns the user when the settings produce an array that does not include the complete sequence of input values.

Essentially arrays which are output from ChangeDimLat can be used as gridded multi-dimensional univariate data. Multivariate data held in a single array can be described using DimToVar. DimToVar takes a single parameter, this specifies which dimension of the array which is to be interpreted as a variable index. This changes the meaning of the

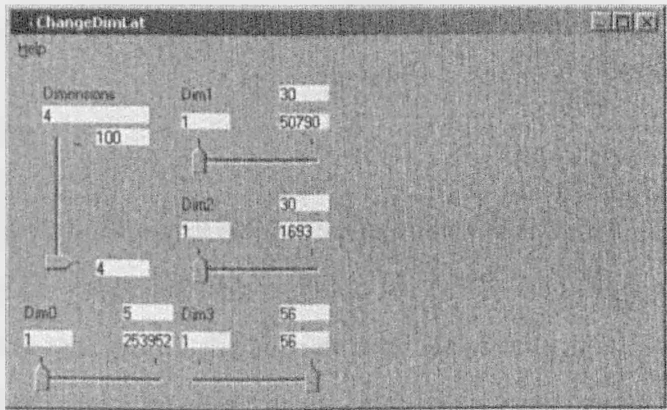


Figure 5.12: ChangeDimLat's user interface; the rank of the array is specified with the 'Dimensions' parameter, this enables parameters 'Dim0' ... 'Dim9' which are otherwise hidden. These parameters are then be used to specify the shape of the array

structure in IRIS Explorer and raises the number of variables in the array from one to the shape of that dimension, whilst reducing the rank of the array by one. For example if a *rank* 3 array has the dimensions: 2, 50 and 45, converting the first dimension into a variable dimension would mean that there would now be two variables, in an array of rank 2 with dimensions of 50 and 45. DimToVar's output can be used as a scattered or gridded multivariate dataset.

Three other modules provide array manipulations. StackDimLat sequentially combines arrays of *rank* n with the same shape into a new array of *rank* $n+1$. The shape of this new array is that of the source arrays with the additional dimension describing the total number of arrays merged. This is particularly useful for sliced volumes where each 2D cross-section of the volume data is held separately. StackDimLat has a clear parameter which frees the currently accumulated data and an output option which sends an accumulated dataset every time the module is fired. The final parameter defines which dimension data is stacked. SliceDimLat has the opposite functionality; for any *rank* n array input it outputs a *rank* $n-1$ array taken as a cross-section through a chosen dimension of the array. Its parameters select first which dimension the cross-section passes through, and second, which slice in that dimension to extract. Finally, CropLat allows a portion of an

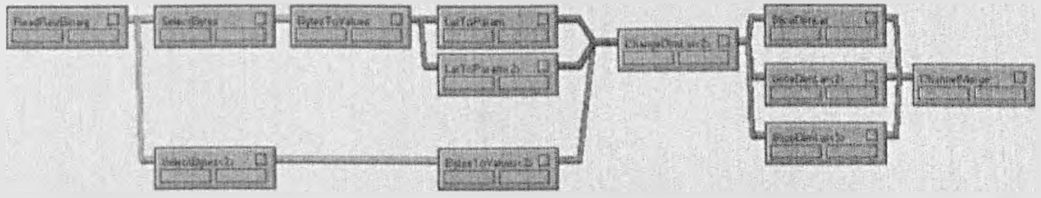
array to be selected by choosing minimum and maximum index ranges.

Variable definition

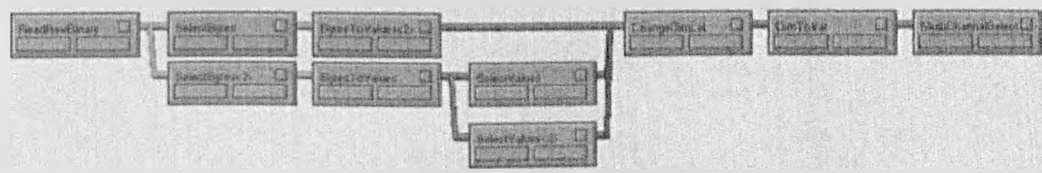
The semantic stage of the file input architecture specifies selections and conversions which alter the meaning of arrays and values within the MVE. IFIT supports this stage with a range of modules that enable the user to combine and manipulate data to use IRIS Explorer's own conventions for a wide range of different variables.

Any module after the value interpretation stage outputs sequences of values that can be used as a variables. Therefore, an output from BytesToValues, TextToValues or TextRecordToValues can be used as a variable; as can any outputs from modules which process their data. Multivariate arrays can be described using IFIT with two different approaches. First, the array's rank and shape, including the variable index described in chapter 3.1.4, is defined using ChangeDimLat. DimToVar is then used to specify which dimension s of the array's shape refers to the variables in the array, where $1 \leq s \leq rank$. This is usually the first or last dimension of the shape, however, DimToVar can turn any dimension into the variable index. The output for DimToVar is a $rank - 1$ data structure with a number of variables corresponding to the size of dimension s. Second, multiple univariate arrays which have the same shape can be merged into a single multivariate array using ChannelMerge. Variables can be selected from these multivariate arrays using the existing ChannelSelect and MultiChannelSelect modules, which respectively output a one or subset of the variables in the multivariate array which was input.

Variables can be separated from a multivariate array using IRIS Explorers existing ChannelSelect or MultiChannelSelect modules. To use some types of variables their order must be altered, for example BGR colour channels cannot be directly interpreted in IRIS Explorer, instead they need to be arranged into an RGB order. Changing the order of variables can be done in two ways, first if the data has not had its variable index converted with DimToVar then it can be sliced along the variable index and then ChannelMerged,



(a) Slicing the array to swap and then combine the variables



(b) Converting the array variable index and swapping the channels by reverse selection

(c) An example of the output these networks produce

Figure 5.13: Two networks that can input a Windows bitmapped picture (BMP) file. These solutions illustrate how values can be taken from the file and wired in as parameters for the file input process. The height and width of the image are taken from the 54 byte header and then used with the knowledge that this file has three values (blue green red (BGR)) per node to dimension the array. The solutions show the different way in which colour channels can be swapped from BGR to RGB

forming a multivariate array as shown in figure 5.13(a). Alternatively, DimToVar can be used with MultiChannelSelect to re-order the variables in the array as shown by the module network in figure 5.13(b).

Variables like coordinate data can be specified in a number of different ways. Many gridded datasets have uniform rectangular coordinates; these can be specified within different

file formats in several ways, including bounding boxes and sample spacings. SetUniform-Coords enables the user to set uniform coordinates in one of three manners, providing a flexible way in which they can be defined. Body-fitted coordinates can be attached to data values using SetCurvCoords which takes as inputs a univariate or multivariate array and combines it with an array of coordinate values to output a body-fitted IRIS Explorer data structure. Finally for coordinates, the SphereToCartes module fits into the semantic stage of the modular approach. It can convert spherical coordinates into Cartesian coordinates, illustrating how variables with the same type of information can have different semantic meanings and therefore need to be converted.

Variables like identifier data are format and not data related. Identifier data can be handled either by ignoring it or, if it shows that the nodes are stored out of order in an array, it can be mapped to the nodal data using VarIdentifierMap. This module specifies, first, which variable in the incoming array of data values is the identifier variable and then, with another input, the connectivity data is mapped to the correct nodes. Such data will often be cell regular, the mapped connections, (or unmapped of there was no identifier information) can be composed into a cell regular data structure using IRIS explorer's ComposePyr module.

5.5 Using visual feedback

IFIT's support of visual feedback through TextView, ImageView and VolumeView allow important diagnostic information to be gleaned from a file. This section will detail how the user can gather and discover different items of information by using these tools.

5.5.1 TextView

TextView (illustrated in figure 5.14) provides the user with a plain-text presentation of their file. This view initially contains text characters mapped to each byte, filling the

screen space in rows: it can be altered to make it more readable by interpreting common control characters or through specifying delimiters and line wrapping options. Finally selections can be taken with the mouse in a similar manner to existing text editors like emacs and notepad, by clicking and dragging over the area desired. In TextView this outputs the start and end of a selection which can be wired into SelectValues to acquire a selection of a file's content.

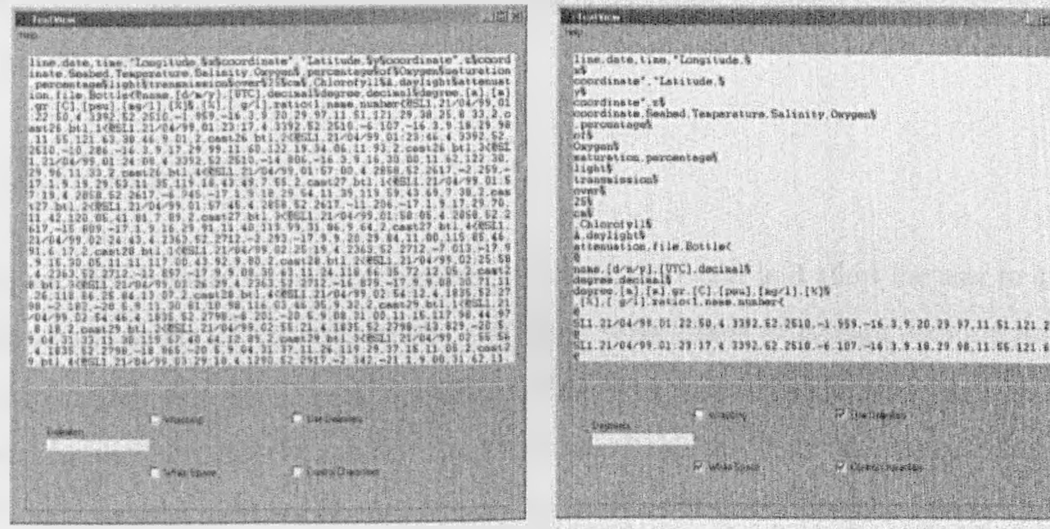


Figure 5.14: TextView in action; two views of the same file using different plain-text character interpretations, the first illustrating a direct view of the file's content as plain text, the second interpreting using control characters and white space to separate values onto different lines.

The view which TextView presents allows the user to discover if a file contains binary or plain-text (ASCII) values. In addition, it enables the following information to be gathered for files containing plain-text:

- the different types (like floating points and integers) of values present;
- header information can define the meaning of other values in the file;
- structural details of the file and a determination of whether it contains a data held using a DDL or using contiguous arrays of values;

- explicit input parameters like the dimensions of the data or the number of variables and their bounds;
- delimiters and tags present in the file;
- selection parameters for data desirable to the user.

TextView's utility over an external text editor is its ability to output parameters directly into other modules.

5.5.2 ImageView

ImageView's visual output and the artefacts which can occur in it allow the user to identify many forms of array-based data, including some forms of storage, like run length encoded data, that IFIT cannot import at present. ImageView offers a powerful diagnostic tool for the purposes of forensic file analysis. It allows any user who has worked with or seen the data or the phenomenon it describes to search for its location in the file and find its dimensions. This ability differentiates this solution from existing input tools and facilitates forensic examination by providing effective feedback. Even users who have not seen the dataset but have experience of identifying artefacts and a little experience using the feedback can find data they have never actually viewed before. ImageView's user interface consists of an interactive viewing area which supports mouse drag interactions and has a menu to set the zoom factor. The user has control over the interpreted image's width, this, with successive interactions with the view, enables the user to 'tune' the image to the shape of the selected sequence of values without knowing any parameters a priori. Figure 5.15 illustrates a sequence of different images taken from ImageView which illustrate how changing the width of the image affects its visual output.

ImageView's visual outputs contain certain patterns that are indicative of correct or incorrect input parameters. However, there are often several possible causes for each pattern,

this leads to an element of 'visual debugging' when interacting with the view. These artefacts can be used as important guides towards discovering file input parameters. Several common patterns that have been identified through the course of this work will now be described in detail:

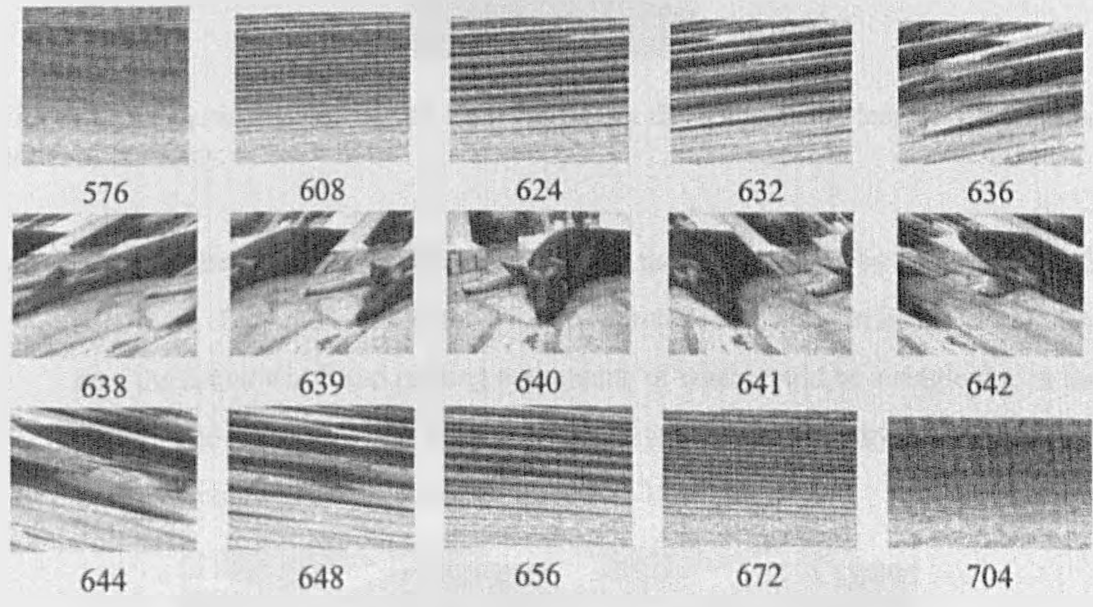


Figure 5.15: The visual effect of trialling different widths for an array whose actual width is 640

Skewing and diagonal lines or non vertical bands Skewing and diagonal to near horizontal bands illustrated in figure 5.15 are caused by the width parameter in ImageView not matching the correct value for the array. When adjusting this parameter, the changing thickness of these bands indicates if the user's interactions are in the appropriate direction. Thickening bands indicate the user is closing upon a factor or multiple of the correct value.

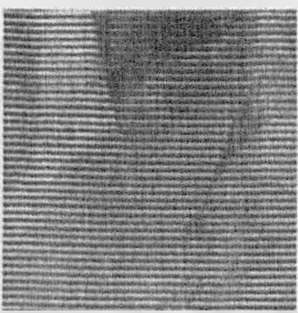


Figure 5.16: Vertically interleaved lines casued by the width being less than the actual width of the array.

Vertically interleaved lines are the same effect as the bands caused by tuning the width parameter. They indicate that the width parameter of the array in ImageView is less than the actual width and causing a wrapping of what would be a single line in the image. The resulting effect is that the values which are vertically adjacent are not vertically continuous as illustrated in figure 5.16.

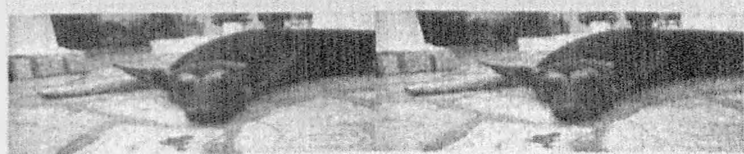
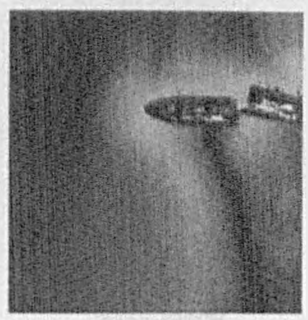


Figure 5.17: Horizontal repeats caused by the trial image width being a multiple of the correct dimension

Apparent repeats of data If there appear to be exact horizontal repeats of the data this can indicate that the trial width is a multiple of the actual width of the dataset. The solution is to reduce the trial width until no repeats are present in the view. Shown in figure 5.17.

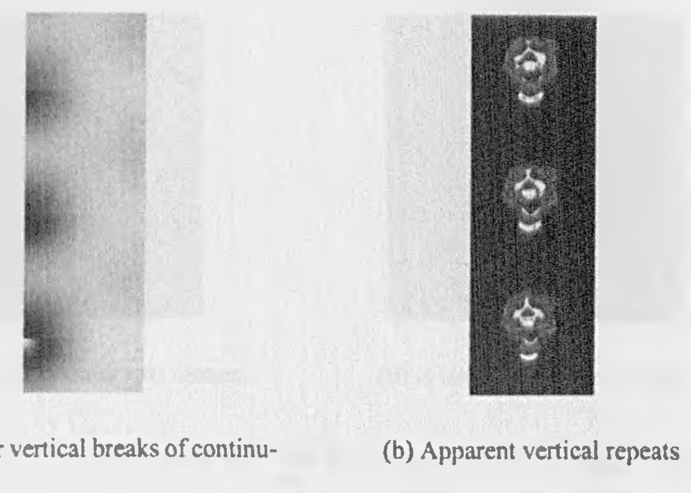
(a) Same DEM exhibiting 'psychedelic' banding



(b) Correctly interpreted DEM

Figure 5.18: DEM data comparison illustrating 'psychedelic' banding

'psychedelic' banding figure 5.18(a) and 5.18(b) show the difference between an ImageView of a sequence of values exhibiting this artefact and a correctly interpreted sequence of values for the same data. The patterns illustrated in figure 5.18(a) are caused when the binary primitive type has been interpreted with incorrect parameters for its byte order or alignment. The effect is caused by the value range being misinterpreted due to the either an incorrect byte order caused by the users choice of endian parameter or the selection start for the bytes which were interpreted into the sequence of values. Psychedelic banding only occurs in integer values of eight bits or more. The effect results in the number range of the binary primitive type becoming divided up into multiple smaller repeated ranges, in effect creating 'contour like' bands through what is a single continuous value range. This interpretation can be corrected either by trying different byte alignments or by adjusting the start point of the selection by $1..n-1$ bytes in a positive or negative direction (where n is the size of the primitive type).

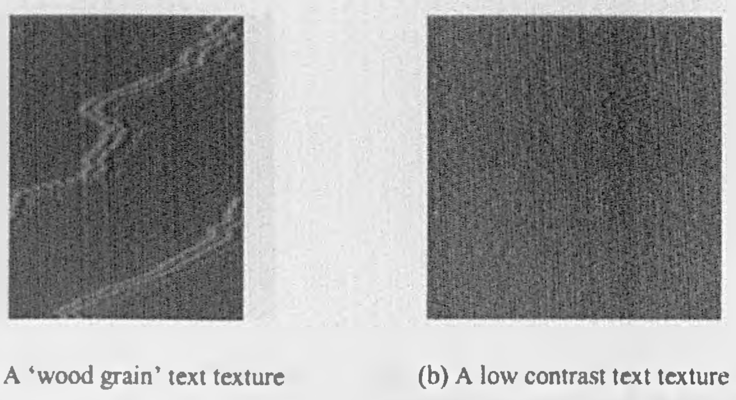


(a) Regular vertical breaks of continuity

(b) Apparent vertical repeats

Figure 5.19: Two examples of artefacts caused by the actual rank of the array exceeding ImageView's *rank* 2 output

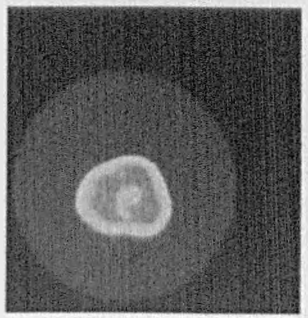
Regular vertical breaks of continuity or apparent vertical repeats Both of these artefacts occur in the ImageView outputs shown in figure 5.19(a) and 5.19(b). They can indicate that the dataset has a further dimension which has not been interpreted at this stage. The correct course of action would be to try to interpret the sequence of values using VolumeViewer, described later. The repeat or break effect can be strong or relatively weak depending upon whether the variable changes subtly or dramatically between slices in that axis. Data which has a strong continuity or pattern, which makes the 'top' edge of a slice different from the 'bottom' edge, will usually have a strong break of continuity effect. Data which is closely sampled or has similar edges (e.g. blank or noisy) in all slices will tend more to a repeating effect; data with sparsely sampled slices and similar edges will exhibit an output in ImageView with a vertical 'movie reel' effect. All these effects occur when the user has discovered one dimension (or a factor or multiple of it). The effect causing the breaks of continuity is caused by adjacent values in the interpreted array not having adjacency in the raw data. The repeating or frame effect is caused by the way slices of volume data are stored adjacently in an array.



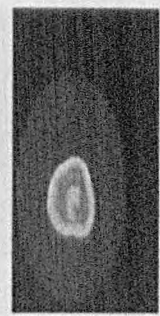
(a) A 'wood grain' text texture (b) A low contrast text texture

Figure 5.20: Text values in a file as viewed through ImageView

Low contrast texture or a 'wood grain' texture for 8-bit values These textures, shown in figure 5.20(a) and figure 5.20(b), are both likely indicators that the array contains text values. To verify this the user sends the data to TextView; if text numbers are present they will need to be interpreted using TextToValues or TextRecordToValues. The low contrast effect is caused because numeric values in a file utilise only a small subset of the possible range of an 8-bit number. The vertical line texture is caused because successive lines of numeric values will have numerically similar digits in similar spacing patterns, although the differences will often cause enough irregularity to usually prevent complete vertical lines from forming.



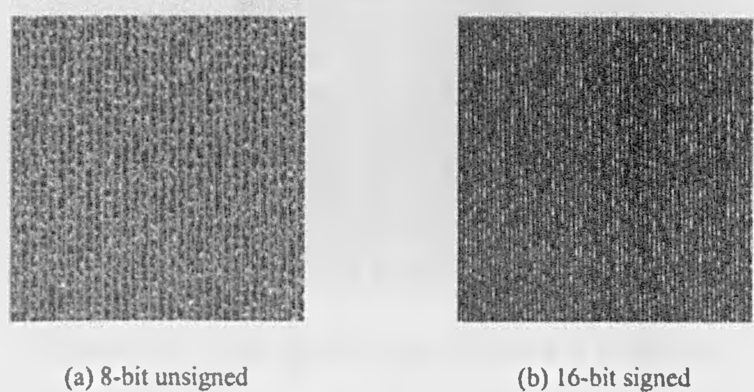
(a) A correct view of a CT dataset



(b) A similar view of the same dataset, however its aspect ratio distorts the image which may lead the user to check the parameters they have used

Figure 5.21: Aspect ratio as an indicator of incorrect input parameters

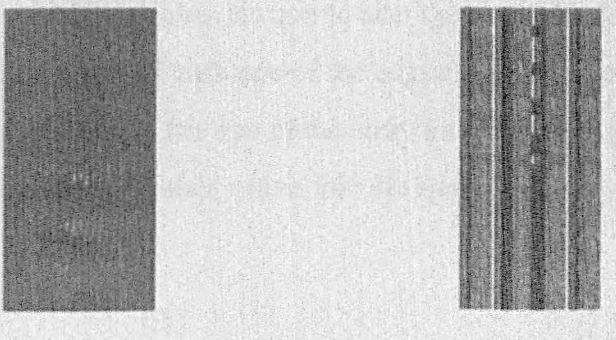
Incorrect aspect ratio Sometimes the data may appear to have dimensions that differ greatly from observations taken elsewhere. This can indicate an incorrect dimension which is a factor of the actual dimension has been used in conjunction with a similar but incorrect binary primitive type. Together these input parameters have produced a view which looks like the original data, but may have a different range of numeric values and dimensions. Both parameters will usually need changing, the width will be a factor of the data's actual width. The difference between the correct primitive type and the current primitive type's size in bytes will determine the factor that the incorrect width must be multiplied by to obtain the correct dimension, once the correct primitive type is chosen.



(a) 8-bit unsigned (b) 16-bit signed

Figure 5.22: Two different binary interpretations of 32-bit values that result in line artefacts

Vertical lines through data Vertical lines can indicate that either an incorrect binary primitive type has been chosen or that there are multiple variables in this array stored adjacently in a dimension which has not been taken into account. Figure 5.22 illustrates the effect of choosing an incorrect binary primitive type and figure 5.23 shows the effect of multiple variables.



(a) Binary data with multiple variables (b) Text with fixed width records

Figure 5.23: Textures illustrating the effect of multiple variables in an array

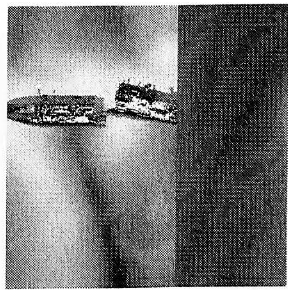
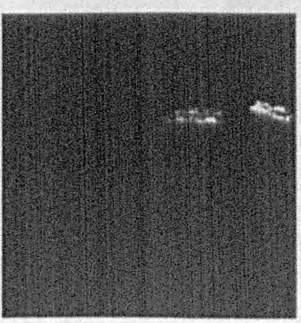
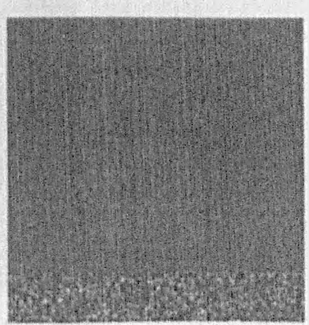


Figure 5.24: Single distinct vertical break of continuity

Single distinct vertical break of continuity This artefact, illustrated in figure 5.24 can occur when the start of the array is incorrectly positioned in the file. The effect is produced by either unwanted values leading into the array or by missing some of the array's data. This in turn alters the interpretation of the values in the array, moving what would be the wrapped edges of the array's first dimension to another location in the first dimension of the interpreted array. This results in the edges of the original array becoming adjacent values at some mid point in the array, and because the edges are not continuous, they will form a visible break in continuity. Correcting this artefact requires the user to alter the start point for the selection until the discontinuity coincides with one of the edges and all the array data is present. It is important to note that the size of the array must be checked to prevent loss of data or the addition of non-data values into the array.



(a) High contrast texture caused by error values



(b) Low contrast caused by an incorrect selection including non-data values

Figure 5.25: Contrast problems found when using ImageView with an automatic range generation for mapping data values to greyscale values

A high or low contrast image This texture can be caused by error values and unwanted values in the data selection. Unusually bright or dark areas in the middle of the texture can indicate error values have been used in this dataset. Such values usually lie at the limits of the binary primitive type used for the data set. Their value can make the use of the data's minimum and maximum limits ineffective for use in mapping the data's values into greyscale pixels for presenting the data with ImageView. An illustration of this contrast effect is presented in figure 5.25(a) and figure 5.25(b). Error values can be tested for by manually setting the range for greyscale mapping in ImaveView. An alternative cause can be found when the start or end of the selection is incorrect and including values which are not part of the array. The bottom and top of the view can be used to discover if this has happened. Figure 5.25(b) illustrates data preceding the array in the file which has been accidentally included in the data selection. Unwanted data values can be remedied through altering selection's start and / or end.

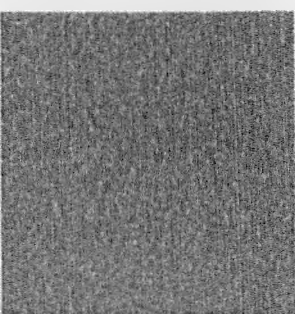
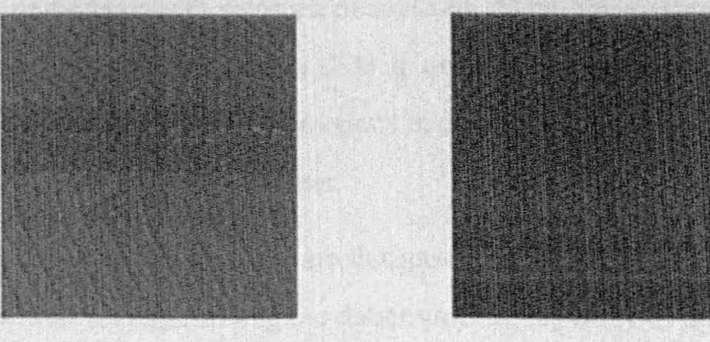


Figure 5.26: 'White-noise' like texture taken from an ImageView of a JPEG image

White noise texture This effect, illustrated in figure 5.26, can indicate one of three possible underlying causes. First is that compression or encryption has been used on the selected values. Second it can indicate the area of the file contains many different binary primitive types stored in no regular order. Lastly it can, like 'psychedelic banding', indicate that the alignment is incorrect for the binary primitive type, which can be solved in the same manner by adjusting the selection's start.



(a) Different blocks of text values (b) Diffferent blocks of binary values

Figure 5.27: ImageView textures that show different blocks of data, each block occuring where the texture changes dramatically.

Irregular horizontal discontinuities This can point to data present in the selection which is undesired. Changing the selection parameters is the appropriate course of action. Examples of this can be seen in figure 5.27(a) and figure 5.27(b).

5.5.3 VolumeView

VolumeView enables the description of 3D arrays and presents the interface shown in figure 5.28. The three windows each present to the user a cross-section through a different axis of the selected data. Each window accepts two-directional mouse drag actions to change the shape of the dimension in the two axes the view presents. These actions can be limited to one dimension allowing the user to concentrate on adjusting one dimension at a time: other parameters include a menu choice which changes the speed at which the mouse affects the shape. Finally there is control over the zoom factor and an option animation, which are both accessible from the menu. Zooming offers the same functionality as presented in ImageView; the animation option pushes the presented slice in each axis through the array, effectively producing an animation of each axis' content.

VolumeView's feedback is caused by the same factors as the feedback from ImageView, except in more dimensions. However, when animated, a directional movement can be detected for some datasets with incorrect dimensions. Much like a cinema film reel which is run at an incorrect speed, the frames slide up or down at a speed depending upon how much the parameters are in error. Interactions in the opposite direction to the movement can lead the user to the correct parameter.

Neither VolumeView nor ImageView are designed to cater for non-gridded data. However, they work well for determining the dimensions of any array of data values in a file, including those which hold multiple variables, and discontinuous data which involves multiple variables. Although non-gridded data cannot be trialled or verified using any of IFIT's current visual tools, it can be passed through the full visualization pipeline. This is currently the only way in which nodal data which is scattered or cell-regular can be visually trialled to verify its interpretation.

Some data is stored in a manner which prevents the user from finding the dimensions of the array because they cannot find a trial width where the data values apparently align

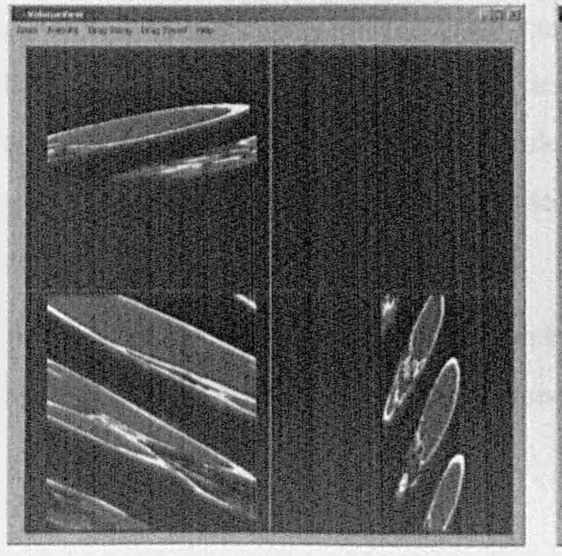
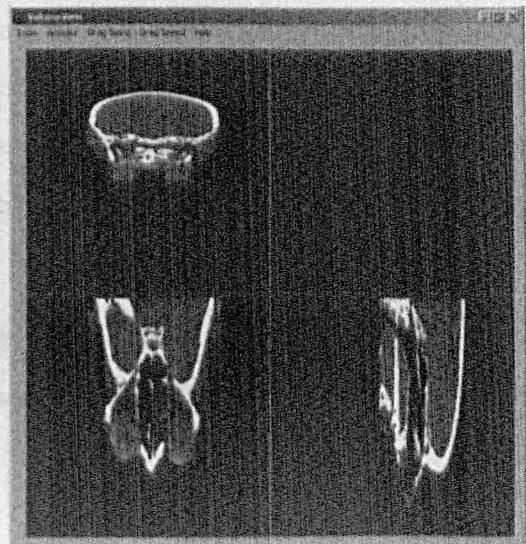



Figure 5.28: VolumeViewer displaying an incorrectly dimensioned array, and the same array when correctly dimensioned

into an interpretable image. This effect can be caused when one or more dimensions of the array, in effect, has a variable length. This prevents ImageViewer from ever producing array dimensions because there is no regular dimension. Such effects can indicate text fields or text value representations, variable length binary record data or finally run length encoding (RLE). If a text interpretation is not the cause then IFIT will be unable to describe the dataset; variable length records introduce another level of complexity which IFIT is currently unable to handle, as does RLE. RLE can be detected with ImageView because it does not change the meaning of all the values in a dataset, instead it replaces sequences of repeated values with markers, thus some data is still viewable, although not accessible.

5.6 Summary

This chapter has described a forensic approach to file input, the requirements for an interactive file input toolkit and a software implementation which supports the forensic

approach. IFIT implements the processes found in the file input architecture presented in chapter 3. Its structure and usage has been described from a design level and on an application level using a range of example module networks. This chapter has also described how the visual feedback modules can be used to discover, debug and verify a file input solution.

The next chapter will evaluate how IFIT compares with the existing solutions and how effective it is at acquiring a user's data and meeting the initial requirements.

Chapter 6

Evaluation of IFIT

The previous chapter described a new forensic methodology and a toolkit (IFIT) supporting the solution of file input problems. This chapter will present a range of test cases and then evaluate IFIT over a range of criteria to show its utility with respect to supporting file input for ViSC.

6.1 Test case selection

All the solutions presented in the following section are of successful test cases and have been selected to demonstrate the range of different data types which can be input using IFIT. The unsuccessful cases and the limitations they have raised will be discussed in section 6.3.2; they have not been included in the example test cases because the module networks that they comprise tend to simply illustrate the use of the forensic tools up to the point where it was discovered that the raw values could not be extracted from the file in a meaningful manner using IFIT.

Each solution was constructed by the author. The test cases in the next section have been chosen from a suite of 34 successful IFIT solutions taken from 43 different file input problems. These problems were provided by a range of users and user groups, and all the test cases are instances of 'real world' data sources. All the solutions which are presented

have been solved based on the user's visualization requirements. This means that not all the information, particularly metadata, stored in the file was necessarily needed to fulfil the user's requirements. However, in each case all the data the user requested was located and visualized.

The diversity of examples illustrates some of IFIT's capabilities with data from different applications and fields which have a mixture of proprietary, commercial and user-generated file formats. The majority of the test cases presented contain different forms of gridded data. Most of the solutions required one or more input parameters to be discovered using the visual tools. Finally, all the solutions have been validated against data values, visualizations obtained by loading the same file into the source software, taking a hardcopy output and discussions with the user. The next section will present the test cases and discuss them on a per-case basis.

6.2 Test cases

6.2.1 Case 1: Medical imaging data

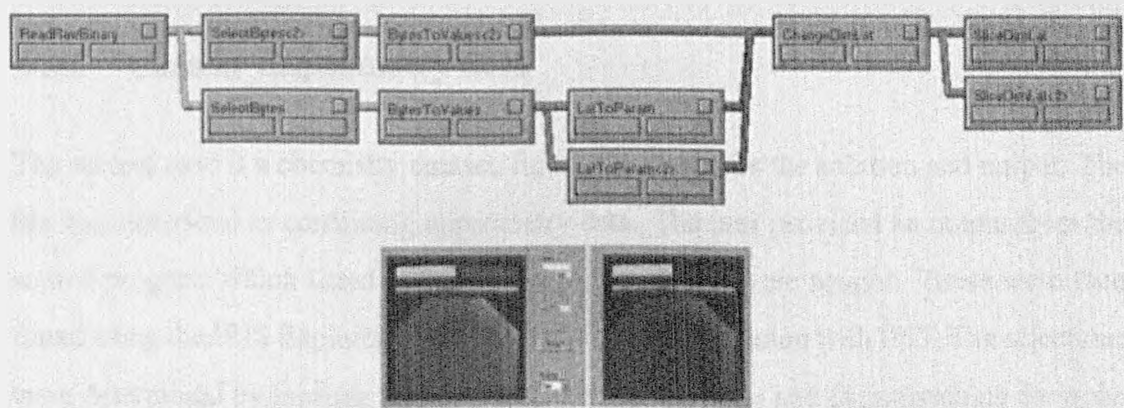


Figure 6.1: A file containing two X-Ray images held in a single array.

The first case is a medical imaging dataset; figure 6.1 illustrates the solution. The file was described by the user as containing two X-ray images, one for high energy and one for low. These were the user's desired output for this case. After an investigation using TextView

the file was found to contain binary values. ImageViewer was then used to locate the two images and their dimensions. They were found to have the same physical dimensions and held in a single *rank* 3 array. The horizontal image dimension was found to vary the fastest, the vertical image dimension next fastest and the variable dimension slowest. The header structure was then searched using the values of these dimensions. SelectBytes and BytesToValues were used in conjunction with an IRIS Explorer print module to display the value at a given location in the file. This search located both the horizontal dimension value and vertical dimension value in the header; the selection was then fixed and the parameters where interpreted and wired into ChangeDimLat. As a result, assuming the header of this format has a fixed structure and will not change size or content, other files in this format with differently sized images can be input.

The medical imager's output is a proprietary file format and as such it lacks a published description. The software related to this imaging equipment also lacks any output that is compatible with the MVE. The IFIT network successfully met the user's requirements for this case in two steps, first by allowing them to see the data, second by giving them useful information about how their data was stored.

6.2.2 Case 2: Elipsometry data

The second case is a chemistry dataset; figure 6.2 illustrates the solution and output. The file was described as containing elipsometry data. The user provided an output from the source program which listed in a table the values which were sought. These were then found using the IRIS Explorer graphing modules in combination with IFIT. The selections were determined by locating values which were in the table and then searching from the ends of the array until values, which were out of the range or at the limits of the numeric representation, were found. These indicated that the values were not part of the data but either from another array or of a different binary primitive type. This provided a good estimator of where the different arrays in the file started and ended. The first graph

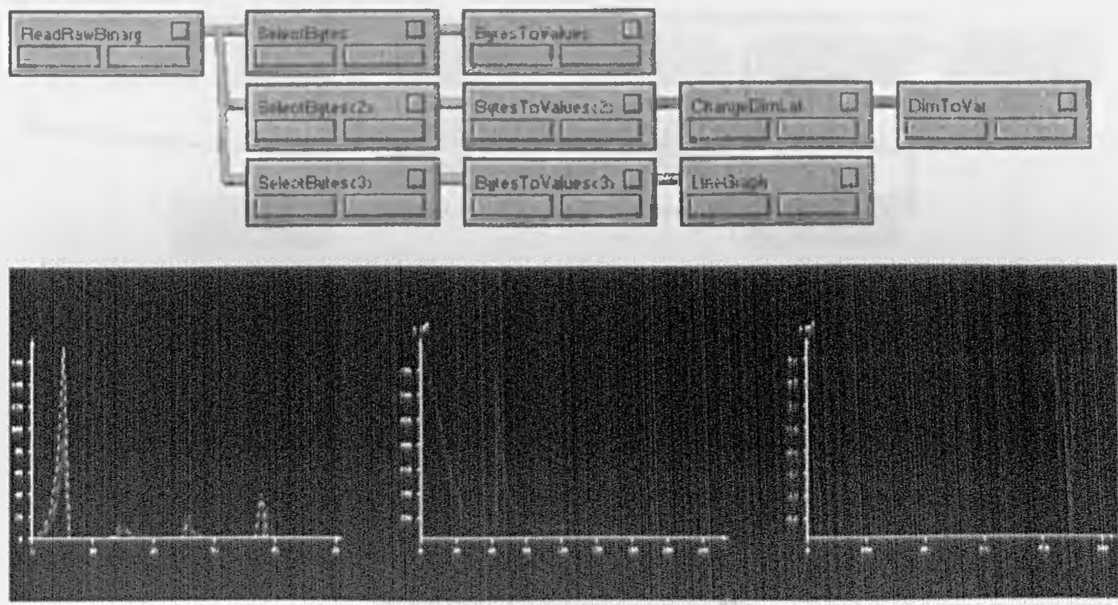


Figure 6.2: A proprietary chemistry file format containing several variables.

illustrates the data desired by the user, the second two illustrate further data which was discovered and, when shown to the user, were identified as related variables which they did not know were stored in the file. The desired data output had a higher accuracy than the user expected and more values than the original listing contained, although these were of no consequence. From this information a file reader was produced which extracted the required data from this type of file and was successfully applied to the user's archive of these files.

6.2.3 Case 3: CT data

Computed Tomography (CT) volume data is often held in individual slice files. The case shown in figure 6.3 highlights how each slice can be loaded and accumulated into a single array. Figure 6.3 shows how a volume made up from sliced can be input using StackDimLat to combine each slice into a *rank* 3 array. DICOM (found in section 2.3.1) is not directly supported by the version of IRIS Explorer used for this project. DICOM files can be compressed: however the data for this case was in an uncompressed form which

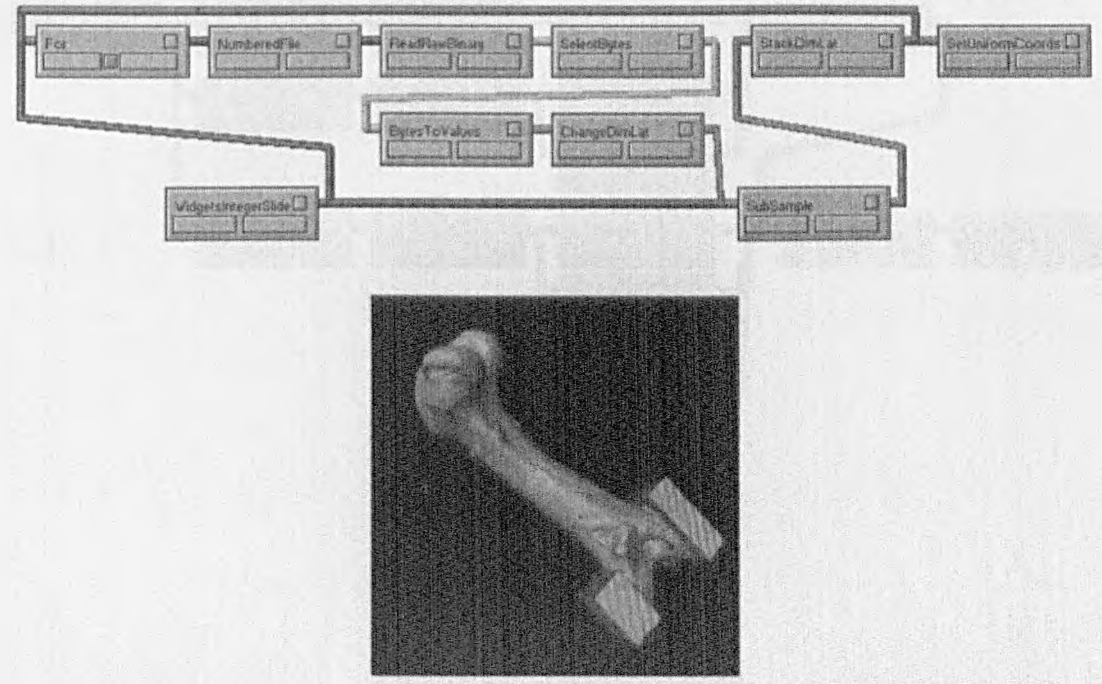


Figure 6.3: A DICOM CT dataset containing slices of a pig femur.

allowed the discovery tools to be used. In terms of discovery the headers of DICOM files are variable in nature. They contain string data and numerous other items of information, which in this case were of no interest to the user. As a result, once the slice dimensions were discovered using ImageView, the design of the network ignores the header and has a one-time specification of the slice dimensions. The data portion is selected and then all the slices are loaded in the same way.

Each file contains a rank two array of 16-bit binary values. As all the filenames follow a set pattern the IRIS Explorer's NumberedFile module can specify the filename for ReadRawBinary, the data is then extracted and described in a fixed manner as all the files are the same size. Finally the data is accumulated and given a set bounding box using SetUniformCoords.

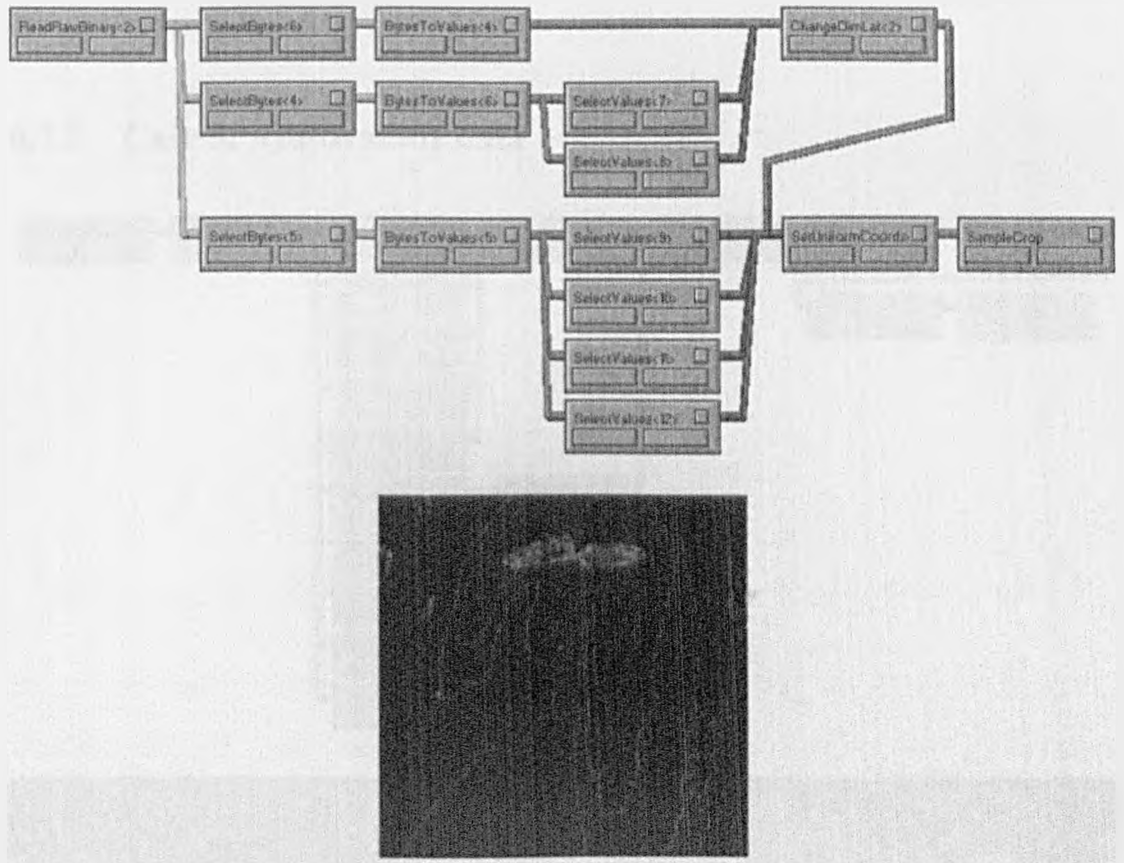


Figure 6.4: A file containing bathymetry (sea depth measurements) in the form of a gridded DEM.

6.2.4 Case 4: Bathymetry data

Case four was provided by a user who requested a visualization of their undersea depth field data (bathymetry). They provided a comprehensive specification of their file format including its structure, binary content and dimensions. They also noted that there was a border in the array of values used for overlapping tiles of this data. In figure 6.4 the module network reads a binary undersea digital elevation model. The file contains a *rank* 2 array of height values and a header containing the shape of the array and physical bounds of the grid. Once the array was interpreted it needed to be cropped, to account for the border. As many files of this format exist, repeated usage was desired; as a result the bounding box and dimensions were located in the header portion of the file and wired to

the appropriate modules.

6.2.5 Case 5: Multivatiate data

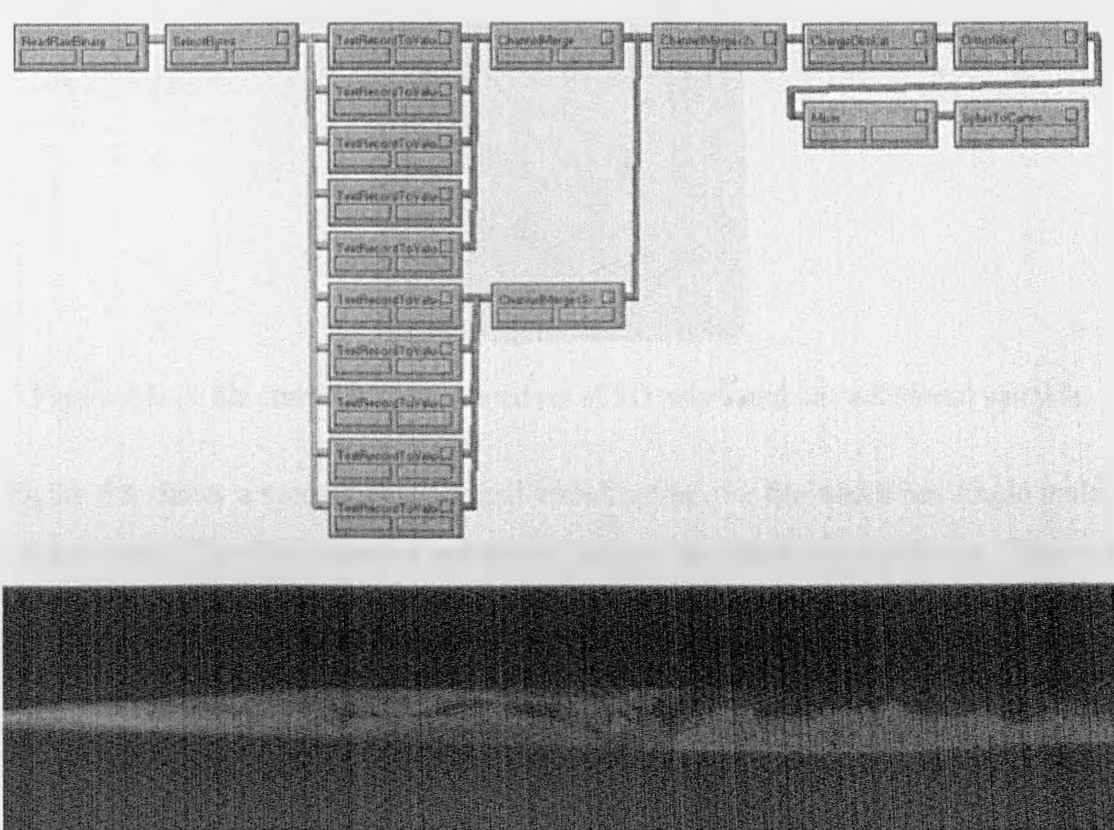


Figure 6.5: A file holding a scattered array of sample points with many variables of different binery types.

Case five highlights a file containing text records. TextRecordToValues is used to extract each numeric variable and make it accessible in the MVE work area. The numeric variables have been individually extracted from the array in the file and then combined into a single array with ten variables. This allows variables which contain text strings to be skipped and other variables of no interest to be excluded. Finally some of the variables are spherical coordinates; these have been converted to Cartesian coordinates so they can be rendered by IRIS Explorer.

6.2.6 Case 6: Scattered data array

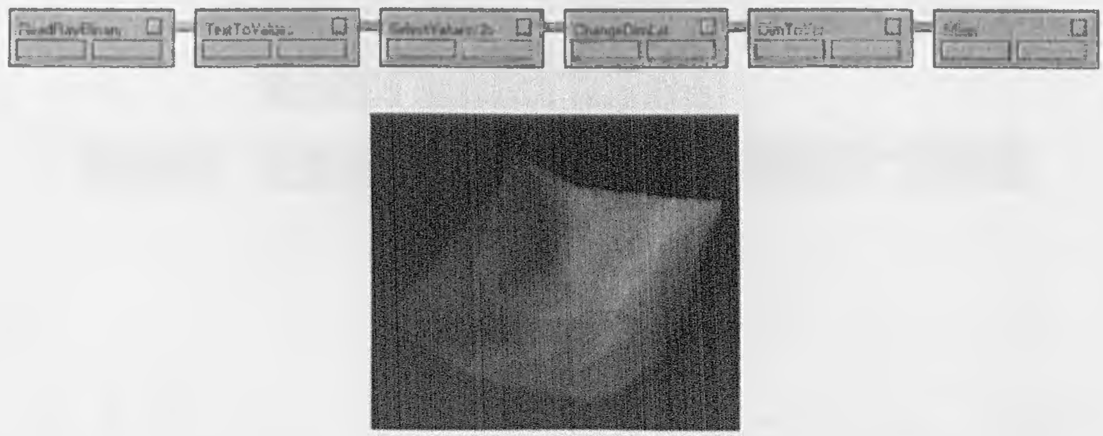


Figure 6.6: A file containing a scattered set of 3D points and one additional vairable

Figure 6.6 shows a module network and visualization of a file which has single multivariate array. The four variables are stored using a text value representation. Three of the variables are coordinate data and have been interpreted as such using the mixer module. The nodes in this dataset are scattered, so triangulation has been used to generate the surface shown in the rendering. The surface has been coloured using the fourth variable.

6.2.7 Case 7: Computational flow dynamics data

The CFD case shown in figure 6.7 comprises two files, both containing values stored using a text representation. One file contains just metadata, the other file contains the actual data. The first file, interpreted by the top four input pipelines, inputs four parameters that describe the dimensions of the array held in the data file. The second file is interpreted by the bottom input pipeline. The parameters in the top pipeline are wired into the ChangeDimLat which describes the rank and shape of the incoming data array. Finally, the data array is a multivariate array, and the variable index is converted using DimToVar. The image below the module network shows a rendering of the output.

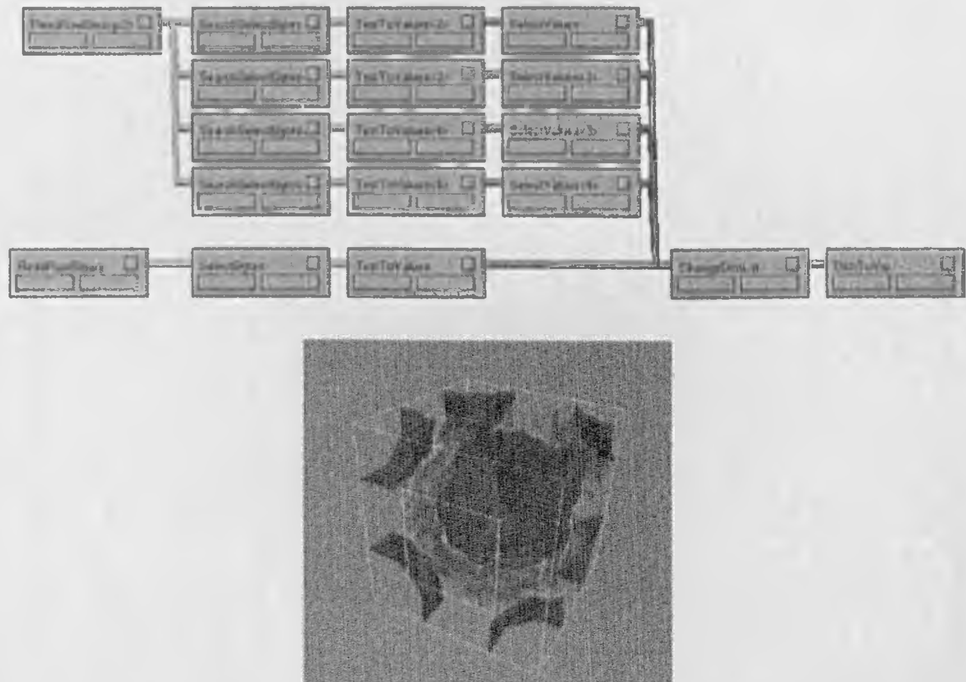


Figure 6.7: A volume containing flow and pressure. The metadata is held in an external file (top pipelines). The loading network combines the metadata providing a reusable file reader for different sized volumes with this number of variables

6.2.8 Case 8: Computational flow dynamics data

The CFD case shown in figure 6.8, is a similar gridded volume to the previous case but, has a single file with the descriptive parameters as a text header followed by a binary data array. The metadata include the dimensions of the array and its physical bounds. These are all input by six input pipelines: the topmost pipeline interprets the binary array, the next three input the dimensions of the array and the lower two pipelines input the physical bounds for the array. Again this case illustrates a multivariate array which is handled using DimToVar: this time the physical bounds for the array are set using SetUniformCoords and the six parameters taken from the last two input pipelines.

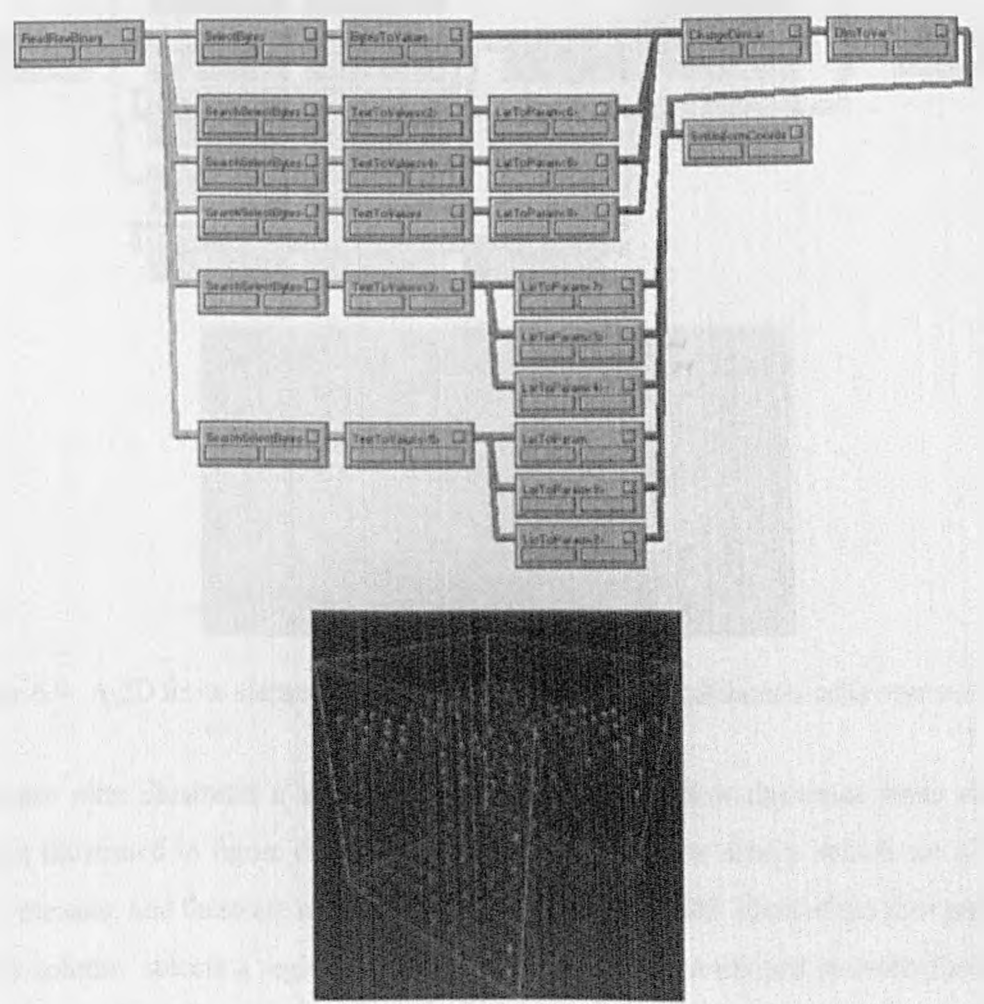


Figure 6.8: A file containing simulation flow data

6.2.9 Case 9: Finite element data

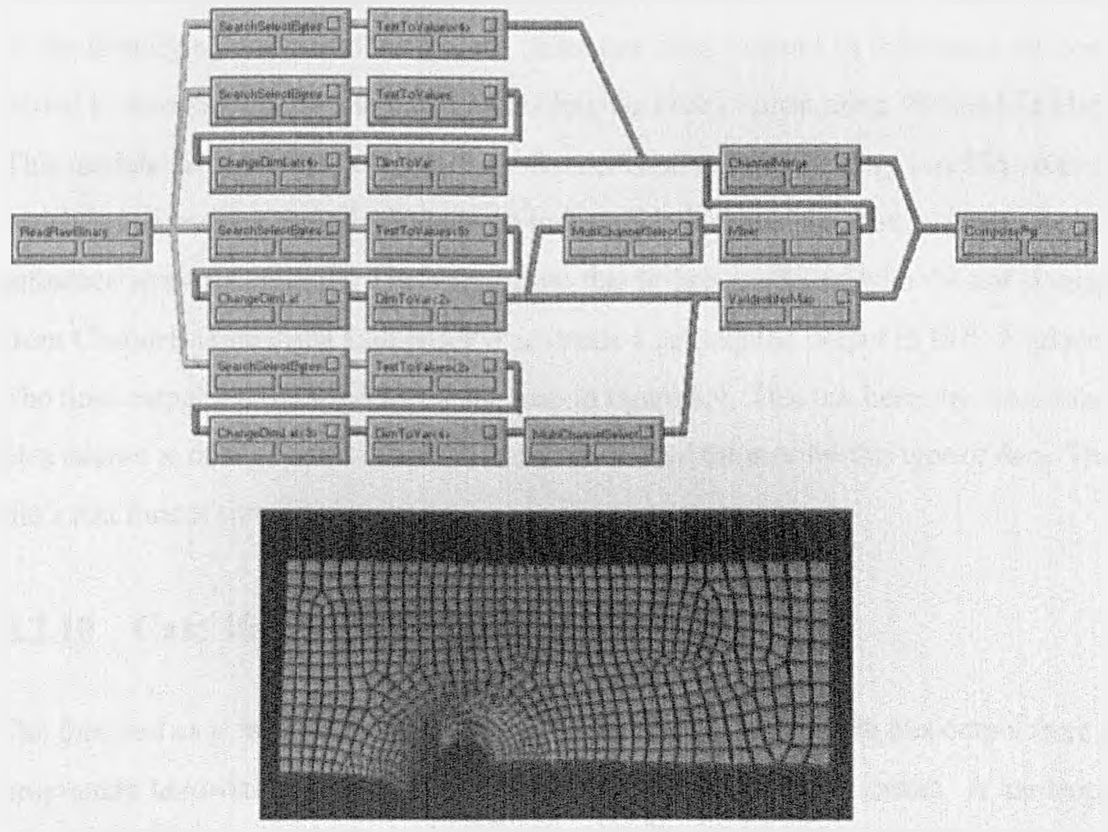


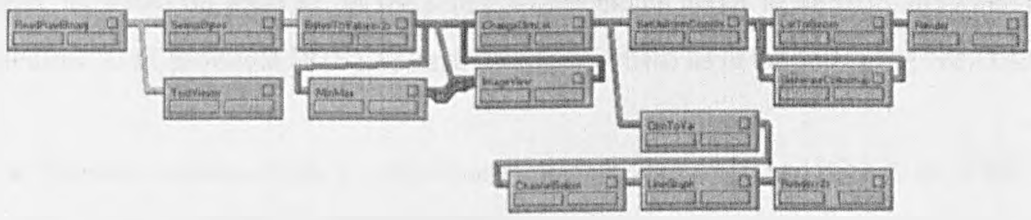
Figure 6.9: A 2D finite element CFD dataset comprising quadrilateral cells over a surface.

The case nine illustrates a solution for a computational flow dynamics finite element dataset illustrated in figure 6.9. The file holds four separate arrays, which are of interest to the user, and these are marked using keywords in the file. Each of the four pipelines in this solution selects a region of the file using these keywords and converts the values held in the region from a text representation to binary. The topmost pipeline inputs an array of floating point vales for pressure at each node. The next one down inputs a 3D vector variable for each node. The third pipeline down inputs 2D positions for each node, each with an identifying number. The coordinates have their identifier data removed and go through mixer which is used to interpret them as coordinate data. They are combined with all the other nodal data into a single multivariate nodal array using ChannelMerge.

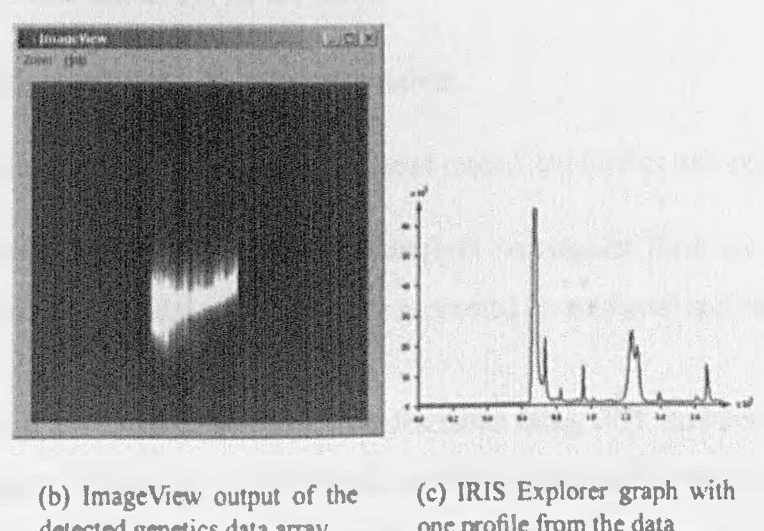
The pipeline at the bottom of the network inputs an array of references. The first dimension of the array is four and specifies quadrilateral cell connectivity. The references relate to the identifying number of each node. After this array is input its references are converted to zero index offsets to the array holding the nodal values using VarIdentiferMap. This module takes an array of identifier references and an array holding variables, one of which is an identifier variable. It finds the index offset for each identifier and converts the reference array accordingly. The output from this is then combined with the nodal array from ChannelMerge using ComposePyr to create a cell-regular output in IRIS Explorer. The final output is visualized below the map in figure 6.9. This has been the most complex dataset to debug because the visual tools do not aid the user for this type of data. The file's text format simplified matters.

6.2.10 Case 10: Gel electrophoresis data

The final test case involves gel electrophoresis DNA profiles. The file was output from a proprietary hardware device, and the user had no knowledge of its format. A hardcopy of a graphical output from the package that was used in conjunction with the device was provided. This hardcopy illustrated processed tracks of genetics data and was used as the target output in a visual search of the file using ImageView. The search found the binary array of values illustrated in figure 6.10(b). The array contained 40 variables of genetics data. The solution shown in figure 6.10 still has TextView and ImageView connected into the input pipeline in order to discover the input parameters and data held in the file. The resulting network is a single solution which will work for this file but is less likely to work for other files of this format and will require one or more input parameters changing and discovering for other files of a similar type.



(a) Genetics data input map with discovery modules still attached



(b) ImageView output of the detected genetics data array

(c) IRIS Explorer graph with one profile from the data

Figure 6.10: This data was previewed on printed output. The file format was proprietary and from old hardware. This prevented the user from accessing it for further analysis and visualization purposes. What was found was an array containing an 40 variables each of a genetics plot

6.3 Test case evaluation

This section will now describe the scope of IFIT in terms of successful applicability and limitations to application.

6.3.1 Successful application

IFIT can be successfully applied to files that store the user's data as contiguous arrays of raw values that use the supported binary primitive types or text interpretations. IFIT's

output, as stated on page 87, in the initial requirements needs to be IRIS Explorer data structures. IFIT produces IRIS Explorer structures which have the following attributes:

- Numeric values of two's complement integers with 8, 16 and 32-bits or IEEE 754 floating point values of 32 and 64-bits.
- Multidimensional arrays up to *rank* 9.
- Multivariate arrays of one or more variables.
- Data point connectivity of either scattered (none), gridded or cell-regular type.
- Coordinates in either body-fitted or uniform rectangular form are supported for gridded data and nodal coordinates are supported for scattered and cell regular data.

User data which has these attributes can be described using IFIT, however its successful input will largely depend upon how data is organised and stored within the user's file. Files with the following range of attributes are supported:

- Multiple files can be described and combined to produce an IRIS Explorer data structure. This includes slices of data in files, and separated metadata and raw data contained in different files.
- Binary value unsigned integers, two's complement integers and IEEE 754 floating point values are supported with stated conversions and accuracy reduction.
- Plain text values are supported for both floating point and integer representations.
- Multivariate data is supported either in a single array or merged from several separate arrays.
- Multivariate arrays containing different values of more than one type are supported in text.

- Named references are supported which enable the creation of cell-regular data from nodal data with identifiers.

- Spherical coordinates are supported through conversion to Cartesian coordinates using an existing IRIS Explorer module.

This enables many field-specific and user-defined file formats to be input using IFIT, including DICOM, DEM and FITS which were described in section 2.3.1. Within the test suite, IFIT can solve 34 of the 43 test cases giving a success rate of 79.1%. The majority of these cases contain different forms of gridded data which include images and volumes with multiple variables of different physical phenomena.

IFIT solutions are produced on a per-file basis, as opposed to the per-format basis of the hard-coded solutions. Reusable solutions can be constructed for some file formats. However, this depends on the user's data input needs and the complexity of the file format. The test cases highlight three types of file input solution, namely complete-use, single-use and discover-use.

Complete-use solutions have described all the content of the file and therefore should be robust enough to handle all files in the same format. This type of solution can only usually be produced for simple file formats and has the same functionality as a hard-coded solution.

Single-use solutions have described enough of the file to access the data repeatedly for one specific file. However, the solution has not described all the parameters in the file and therefore may not work with other files of the same format.

Discover-use solutions require the user to discover some input parameters every time they input the file, therefore there is flexibility but also an increase in the amount of time and user interactions needed to input the data.

6.3.2 Limitations to application

IFIT is limited in its application by the following attributes of datasets and file format, each of which individually prohibits the input of a usable dataset. Each attribute is listed below:

- Binary primitive types of less than one byte, floating points which do not adhere to the IEEE 754 standard, fixed point values, binary coded decimal, and values which do not have a byte order that is little- or big-endian.
- The user's data is stored parametrically, resulting in no raw data which IFIT can extract from the file. Additionally, no feedback can be gleaned from these files because of the lack of raw data values and as such they are an intractable problem. Without extensive user knowledge and a way of enabling them to perform their calculations on these parameters they cannot be input. Compression and encryption are both subsets of this group.
- Data values which are stored in a file which uses a data description language (DDL) to assign values to variables.
- The format uses a language to assign values and give meaning to values.
- The file contains arrays that have a variable index which is neither the first nor last dimension of their shape, no such arrays have been found in over the course of the research they are however plausible.
- Variable length records or any type of data structures that can have a different number of values held at each node or in each record. This includes mixed binary and text records where the text fields affect each record's length in a local manner.
- Variable rectangular coordinates for gridded data.

- Nodal data (e.g. colour table images) where the references are gridded and the data is a list of numbered values.
- All cell-variable connectivity.
- Cell-centroid or point data is not supported.
- Cell-regular data structures which are not specified using separated nodes and connections.

One or more of these attributes were present in the nine failed cases from the 43 file input problems that the test suite was selected from. While this provides a figure of 20.9% representing problems that are intractable using IFIT, it cannot be considered a representative figure for all file formats. The author suspects that the figure is higher, given the usage of file formats containing cell-variable connectivity and the usage of DDLs in various scientific fields.

For some types of file format, particularly self-describing formats like HDF and XML, the format does not necessarily prevent IFIT from producing a solution. Instead the way that data is arranged in these files becomes the limiting factor. The same can be said for file formats which have optional compression schemas. Although the use of compression was excluded in chapter 1.6 from the scope of the project, its use must be noted. Scientific file formats, mostly found in the fields of medical and satellite imaging, use several different compression schemes. These include run length encoding (RLE), image quantization and the LZW algorithm. While IFIT cannot currently input data in this form, several file formats have options for storing data without compression. Therefore, for some cases, data can be re-saved without compression from the source software or using an external tool. The result in some cases will be a file of the same format, albeit much larger, that will enable the user to access their data with IFIT and with their source software. This is currently the only option for users with compressed data.

Complex binary structures can prove too hard to describe using IFIT. If the data is held using complex binary structures or cannot be visualized without a large number of values taken from such structures, it is likely that IFIT will be able to solve that particular file input problem.

Each limitation can be addressed by future developmental work to make IFIT a more complete solution. The essential problems of how any data can be accessed and manipulated have been given proof of concept, as have the utility of visual tools for file input. Every process in the file input architecture has one or more modules which show its utility with respect to file input and inputting a user's data.

Within the scope described in chapter 1, the following file formats outlined in chapter 2 cannot be described by IFIT without additional programmed extension: mmCIF, GRIB and BUFR. In the case of mmCIF, while a field-specific file format, it is also a langage-based file format. In the cases of GRIB and BUFR, IFIT's ability to input data from them depends upon how they have been configured by the user or the source software.

6.3.3 Factors affecting the utility of the visual techniques

The visual feedback techniques presented in section 5.5 (page 106) have some limitations which are inherent with any information system. The amount and type of 'information' present in the dataset can affect the user's ability to elicit a recognisable view of data in a file.

In order for any meaningful parameters to be discovered, visual feedback requires a relationship to exist between the values which are to be examined. This relationship can result from a single continuous variable, multiple continuous variables or multiple discontinuous variables. For a single continuous variable the relationship between values which are adjacent in the data, but not adjacent in the user's interpretation of the data, causes a selection of visual artefacts to be produced which indicate incorrect parameter choices.

The same can be said for multiple continuous variables in addition to the visual artefacts caused when the variables are adjacent in the first dimension of the array. The artefacts are produced by the relatively weak relationship between different variables values.

Finally, data sampled at irregular spacings, like scattered or cell data, can still generate useful visual feedback if there are multiple variables at each node which are stored adjacently in the array. The visual effect cause by this will enable the user to perceived the number of variables in such cases; other dimensions in the array will not be obvious using the visual techniques due to the data's lack of continuity.

The availability of values is a prerequisite for using visual feedback; if raw data values cannot be extracted from a file due to compression, encryption or an unknown binary primitive type, no meaningful feedback can be generated. The ability to extract single byte values is always available to the user: however, if the type used by the data cannot be converted then IFIT has no interpretation which will enable the user to input their data, and the feedback may only indicate attributes of the data.

The amount of 'information' present in an array affects the user's ability to elicit a recognisable view. Arrays devoid of information or those with a high noise content can prevent any recognisable feedback from being generated. Repeated patterns in the dataset can also either prevent or hinder the discovery of an array's shape. The user requires a minimum recognisable feature to elicit a useful response cycle. This feature is something they know from an external source which is in the dataset and shows up in the raw values clearly. From the cases in the last chapter, the bathymetry test case on page 127 has a ship which shows up clearly and aids the user in finding the correct dimensions. Conversely an experiment to input an image containing rasterized text took much longer to discover the dimensions, because the data mislead the user's perception of how close they were getting to the correct parameter. Most continuous datasets are easy to discover using the visual methods, and most of the artefacts described in section 5.5, if communicated to the user, enable quick trials and parameter changes to lead to the answer. Overall, the

amount of data involved improves the feedback effect; very small array dimensions leave little scope for finding an answer. VolumeViewer's animation effects are equally limited in their usefulness by the third dimension of the array: if it is very small then it is unlikely to produce any guidance other than the image artefacts. VolumeViewer's fixed frame rate for animation can also prevent arrays with a small third dimension from producing perceivable animation artefacts. The key factors affecting the utility of visual feedback are listed below:

- Data contains continuous domains or multiple variables.
- Dimensions are large enough to enable the user to recognise a pattern.
- The information content is not masked by high levels of noise.
- There is a minimum of information content, i.e. the data is not constant.
- There are no strongly repeated patterns or totally identical adjacent areas in arrays.
- Values are accessible and in their raw form.

6.3.4 Factors affecting software performance

IFIT's ability to solve a file input problem can be affected by both the platform on which IRIS Explorer, and hence IFIT, is executed and the size of the file or dataset which the user needs to input. The main limitation is IFIT's ability to produce real-time interactive feedback.

Creating a file input solution with IFIT and a forensic approach requires the whole file to be loaded in order to generate the appropriate feedback. This can cause problems for users with large datasets that exceed the hardware resources which are available, either in terms of memory requirements, or in terms of prohibitive execution time for a particular display algorithm, e.g. marching cubes for isosurfacing.

Space complexity The hardware must have enough available memory or virtual memory resources to store whole files in their entirety. On common desktop systems this can prove problematic for files with a size of over 100MB. At a design level, because many IFIT modules convert data whilst requiring a copy of it to be left for other possible interpretations, there can be several copies of a dataset at different stages in the pipeline. Future developments may be able to simplify this but in the current version this can lead to a dramatic increase in the amount of data being handled by the system. IFIT can be used to solve problems with a high space complexity, however, the hardware must have enough available resources to allow storage of the whole dataset.

Time complexity only becomes an issue when using modules in the host MVE which have a high order of complexity. The visual output which is produced solely by IFIT uses minimalistic computation, with most processes having $O(n)$ complexity, so only modules from outside the toolkit like triangulation or isosurfacing can cause time complexity issues.

The platform which IRIS Explorer runs on does affect the performance of some IFIT modules. Visual modules in particular do not transfer well to different operating systems and graphics systems. OpenGL acceleration is an a priori requirement, however, as this is also a need for the MVE, this does not present a concern. The features of OpenGL which have been performance-enhanced for the graphics card in use also effect the performance of the visual tools.

Group compilation IRIS Explorer maps can be grouped and compiled into a single module with only the required parameters visible. This feature allows faster inter-process communication and a simplified interface to be generated for file formats for which the user has produced a solution or part solution. As a result of IRIS Explorer's ability to provide this functionality, module network solutions will not necessarily be slower than hard-coded solutions or programmed extension, although

this will largely depend on how optimised they are for their specific file input problem.

6.4 Usability evaluation

The user requirements outlined in section 5.2.1 and an assessment of the complexity of producing IFIT solutions will be used as the core of this usability evaluation. Section 5.2.1 described the following requirements for a file input toolkit:

- Consistency of interface.
- Unambiguous terminology.
- Clear feedback and outputs.
- Transparency and accuracy.

Two skills are required to input data using IFIT in IRIS Explorer. First, competency in building module networks within the IRIS Explorer work area, changing module parameters and wiring inputs and outputs. Second, the less quantifiable problem solving abilities needed to discover any unknown parameters, choose the appropriate modules and wire them together for a particular file input problem.

The first skill should be known to novice users of IRIS Explorer because it is part of the system's general usage. In this respect IFIT offers no additional difficulty in its usage than the usage of the MVE. Moreover it is a consistent way of using the system, rather than switching to some monolithic tool or wizard. In this way IFIT meets the requirement for a consistent user interface.

The second skill is less tangible and only aided by the methods associated with the forensic methodology of file examination and description, and experience or examples. A selection of tutorials and worked cases may be one way of instilling this knowledge and

providing the user with a mental model of how their problem relates to others and thus how it can be solved.

IFIT attempts to use unambiguous terminology for naming conventions and user parameters as set out in the requirements. It departs from computing terminology like stride and offset where possible to use more generally understandable terms like selection, start and end. In defining byte order, the user does not need to know the make of their machine or what byte order the file uses. If the data does not display correctly, a 'swap byte' option is suggested as a simple way of handling the byte order.

IFIT's modular structure mirrors the file input architecture and therefore enables data to be extracted at any point and examined. This examination is facilitated through the visual feedback modules which provide the user with a range of options for visualizing semi-interpreted data values. The feedback IFIT offers is effective for diagnosing a range of different incorrect file input parameters. There is a definite need for guidance to aid the user in interpreting what they see. Given no aid, some of the textures which are generated by ImageView and VolumeView are counter-intuitive and will lead the user away from the correct answer. However, with guidance, in the form of a selection of explained texture swatches or a tutorial, many file input parameters can be discovered quickly and easily.

Modular networks provide a type of file input solution which shows the user what has been performed upon their data at every step since it entered the system, including any conversions and interpretations which have been made. This is important for scientific integrity, as the user is aware of the processes which have been performed upon their data prior to passing into the visualization pipeline. In this respect modular solutions offer more than the 'black boxes' which other file input solutions often represent. Equally important is the ability to validate a solution. With all file input tools, it is entirely possible to produce an output from a file which looks similar to the original dataset but has radically different numeric values. For previously unvisualized datasets, it is entirely possible to create a visual mapping which has a compelling view that is incorrect. In this respect, IFIT is no

different from any other file input solution which has been user generated. Some solutions warn or terminate if some parameters fail to produce reasonable results, like tags are not found or dimensions are out of bounds. IFIT does not, it will always produce an output, but will warn when errors have occurred about which the user should be informed. Every module logs events that do not seem to offer useful results and the user is notified of these events. IFIT modules enable the user to check at any point in the file input pipeline what the values of the data are and if they match their expectations. Therefore, the requirement for transparency is met through IFIT's modular approach.

Overall, the complexity of any solution made using IFIT is dependent upon the following five factors:

- The data's complexity.
- The file's complexity.
- The data's visualization requirement.
- The user's knowledge.
- The solution's intended usage.

One method for measuring the general complexity of a file input solution is to count the number of modules which are required to input the user's data. The number of modules in each of the successful cases in the test suite was totalled, as were the number of explicit file input parameters and arrays needed from each file. On average, three modules are required for each item that the user needs to input from a file. Using this metric, the total number of items needed from a file increases the complexity of creating a solution. As a result, files containing many variables of different binary primitive types will need a much larger number of modules to extract data from, making them, by this metric, the most complex. However, the data involved can play a much larger role in the complexity of making a solution. This is because the gridded data is best supported by the current range

of visual feedback modules. The construction of file input solutions for univariate gridded data is the most simple, followed by multivariate gridded data and then cell regular data which is the most complex type of data structure that can be input using IFIT.

The complexity of the file format also contributes to the complexity of producing a solution. Some files contain a lot of information which describes aspects of the data which are not needed by the user, equally some file formats separate or group data in a manner which makes their extraction for use harder than others which store similar data.

The data's visualization requirement describes the amount of information the user needs from the file. If they do not need all the information in a file then this can simplify the construction of a solution, conversely, if they need all the data then this can add complexity to the construction of a solution.

Another factor in the complexity of making a solution is how much the user knows about their problem, in terms of metadata, data and file format. If they lack any information the problem will require the user to discover the missing information as opposed to just specifying what they know. When a user knows little about their file, the complexity of making a solution is mostly determined by the format and data's complexity. A file input problem may be therefore be intractable if the user knows little and the file format is complex or contains data which provides no useful view using ImageView or VolumeView.

Finally if the user needs a specific type of solution from one of the three outlined in section 6.3.1 then that can also add complexity to the creation of a solution.

Complete-use solutions are the most complicated as they require the user to have firstly discovered what all the values in the file mean, and how they are described, and secondly to have been able to use them in constructing a modular network solution. However, they are the easiest to use subsequently for loading many files of the same format.

Single-use solutions are moderately simple to construct. Once the input parameters are

discovered and set they are then saved with the module network. These solutions will repeatedly load the file for which they were constructed.

Discover-use solutions are the simplest type of solution to construct, and, once they are made, can load different files by discovering their parameters with visual feedback. However, they are the most time consuming solution with which to input files.

For the experienced user of IFIT, file input problems revolve around information discovery and identifying the structures in the file that IFIT can input, followed by constructing an appropriate network. For the novice user, the utility of IFIT modules needs to be highlighted by example and by tutoring them in the structure of an IFIT solution. There is anecdotal evidence of novice users who have created a file input solution with a minimal explanation of IFIT and only an image to guide them in their search for the file's data.

IFIT most benefits the experienced visualization user, visioneer or visualization expert. For any of these users it will enable them to input a wide range of formats which are within IFIT's limits. IFIT minimises the need for the visualization expert to have extensive and iterative communications with the user, reserving such time-consuming activities to verifying the output.

6.5 IFIT compared with existing solutions

The AVS file access objects (see 2.4.1) are the only comparable case of a file input tool which uses the modular approach. AVS's File Access Objects (FAOs), field mappers, extractors and combiners provide a range of modules, functions and AVS objects which, in theory, offer a similar utility to the transformation and specification range of modules present in IFIT. They enable the user to build AVS fields by extracting data values and arrays from files and combining them into AVS structures. Table 6.1 compares AVS FAOs with IFIT in terms of the types of data and file which they can input and their user interface.

File input attribute	AVS FAO	IFIT
Visual feedback via	Pipeline only	Pipeline and visual tools
User interface	Text type-in	Widgets
Supports very large file input	$\surd$	×
Solution restart for changes	$\surd$	×
Binary primitive types supported	5	11
Supports little- and big-endian	×	$\surd$
Extensible	$\surd$	$\surd$
Scattered data	$\surd$	$\surd$
Uniform-rectangular gridded data	$\surd$	$\surd$
Variable-rectangular gridded data	$\surd$	×
Body-fitted gridded data	$\surd$	$\surd$
Cell-regular data	$\surd$	$\surd$
Cell-variable data	$\surd$	×
Axial Systems supported	3	2

Table 6.1: Comparison between AVS's File access objects and IFIT's input facilities

In some respects AVS's modules cover a much wider range of problems because they have implemented functions and data interpretations for many different high-level mappings, handling data of different axial systems and types of structuring. However, IFIT has some better low-level functionality for handling values, particularly binary. IFIT's ability to convert binary primitive types which are not directly supported by the MVEs data structure and handle issues like byte ordering are not supported in AVS's file access objects.

The interface and GUI usage of AVS's file access objects, functions and modules makes them difficult to use. The File Access Objects require typed text inputs which can contain V script as their control parameters. While flexible, this is not a simple user interface. When viewed in relation to the terminology and description of some of these parameters, the usage of file access objects can leave users opting for the technically harder routes of programmed extension and use of AVS's own scripting language V.

There are several major differences in the way IFIT and AVS's FAOs operate and their

intended use. FAOs handle the file using a file pointer, and no data is accessed from the file until the data structure, which has been described with them and the field mappers and combiners, is accessed. This shows their intended usage as a descriptive tool for large datasets where inputting the whole dataset may result in system resource problems. Their technique instead is to construct a solution, fill in the parameters and then load in the portion of the dataset the user requires. It enables the user to load and process the dataset in small sections.

By contrast IFIT caches the whole file, and will operate correctly if parameters are left blank. When parameters are changed, IFIT will be able to alter the interpretation from that point forward, as opposed to the FAOs which require the whole process to be restarted (probably to reset the file pointer, although this is an assumption).

Overall, the way in which AVS FAOs operate is less useful for forensic examination because it hinders the process of trialling input parameters. This is because file access using them operates in a similar manner to the way traditional programming language perform file input. Parameters for AVS FAOs need to be filled-in in advance otherwise wholly incorrect outputs are produced which have no apparent relation to the raw values in the file.

AVS lacks any feedback tools for file input. As a result any parameter changes need to be sent through the whole visualization pipeline. Without specific tools for file input, the interactions with these views produced by AVS and the actual visualizations AVS supports will be less useful than the output from IFIT's visual feedback modules. The file access objects and other tools present in AVS simply were not designed to discover information from files, instead providing the user with the ability to describe file content. IFIT provides a way of discovering information, AVS's modular approach, without the ability to effectively offer forensic analysis of file content, will not be able to solve some file input problems where IFIT has been successful.

IFIT, as a modular network solution, cannot be compared directly with other different

	Simplicity of use	Flexibility
Most	Hard-Coded	Programmed
	Monolithic tool	Modular
	Modular	Scripted
	Header files	Header files
	Scripted	Monolithic tool
Least	Programmed	Hard-coded

Table 6.2: The modular network approach in relation to other types of file input approaches

forms of file input solution. However, from anecdotal evidence we can form the comparative assessments shown in table 6.2 for both the simplicity of use and flexibility of each particular technique.

The level of flexibility shown in table 6.2 describes the ability to change a solution once it has been created. If the user cannot change the implicit file input parameters or the description of the file's content then the solution can be regarded as inflexible. Changes can be made at compile-time, run-time or real-time. In IFIT, parameters can be changed in real-time as opposed to hard-coded solutions where, for the vast majority of these solutions, implicit parameters can only be changed at compile-time.

Simplicity of use relates to the level of autonomy and the simplicity of the user interface for solving a file input problem with the given type of approach. Hard-coded readers are usually automatic and have the simplest interface, with usually just a file name to enter. Conversely, programmed solutions require familiarity with many different aspects of the system, file and MVE.

The modular approach and therefore IFIT, as shown in table 6.2, can be seen as easier than using a programmed approach, and marginally simpler than the other scripted and header approaches. This can be suggested because the other techniques all require additional knowledge and skills over those needed to use a modular approach which is an extension of the general use of an MVE system. The modular approach can be seen as more flexible

and adaptable than all but the programmed approaches because it is extensible and easily reconfigurable. However it is less flexible than a programmed approach which provides the user with the maximum level of control over the input process.

By learning a programming API, a set of dialogs, or a header description format the user is engaging in something which is not consistent with the rest of the MVE's usage. IFIT allows the user to apply their knowledge of modular programming which is used elsewhere in the MVE and instead learn about the applicability of the IFIT modules to their particular problem.

6.6 Discussion

This chapter has reviewed a selection of cases where IFIT has been applied successfully. The lessons from these cases and others have been used in evaluating IFIT. The evaluation has focused upon the attributes of both user file formats and user data which enable and limit successful application. These attributes have been grouped into those which directly affect the use of the toolkit, and those which affect software performance and the usage of the visual techniques. Finally IFIT has been evaluated for its simplicity of use and it has been compared with present forms of file input solutions.

The test suite contained 43 different file formats, of these IFIT was used to solve 34. While this results in a success rate of approximately 79.1% for the test suite, the translation of this into an overall figure for all file input problems faced by users is unquantifiable. This is because the test suite is a combination of problems received by visualization experts and NAG for IRIS Explorer. The actual number of users who have access to MVE software but either cannot use it because of file access problems, or choose to solve their problem without consulting visualization experts is hard to judge. The quantity of gridded data in the test suite was high compared to the quantity of nodal and cell-based data. Given the usage of nodal and cell-based data in fields like finite element analysis, computer

aided design and geographical information systems, file formats containing such data are probably more numerous than the number found in the test suite.

Unlike other solutions, which are designed as purely descriptive tools, IFIT enables the user to experiment freely with the interpretation of their data and visually trial different file input parameters in real-time. Its visual outputs can offer much more to the user than just text views of file content or hexadecimal views. If the user lacks knowledge of the file format or dataset IFIT provides them with facilities to discover the information they need. It requires no a priori knowledge in order to input the content of a file. IFIT's data-orientation allows a user who knows nothing about the header or structure of the data of their file to extract raw data values, provided the file is within IFIT's envelope of description. Given sufficient experience, IFIT can allow a user to input what would be an intractable problem with other file input solutions.

Like other solutions, IFIT can input a wide range of files containing scattered, gridded and cell-regular data in a range of binary and plain text numeric representations. It can also produce reusable solutions, which once created, act like hard-coded input solutions.

Chapter 7

Towards autonomous data input

Towards the end of this project, the use of ImageView and VolumeView provided insights which implied the possibility of an algorithmic method for discovering the shape of an array. As most file formats store values as arrays, this functionality would be of tremendous use. This chapter documents the resulting research into finding an autonomous solution to this problem and illustrates one clear advance toward an automatic system for data input, one of the main aims of this research outlined on page 10 in chapter 1.

The interactions required to discover the dimensions of an array using ImageView were the first inspiration that there could be an automatic method for finding the shape of an array. Initially, the search looked toward systems that could analyse the output of ImageView and then provide corrective control parameters. Such a system would essentially perform the same task as the user by iteratively correcting the parameters until the array's shape was found. This notion led to existing work in the field of machine vision and optical character recognition. Algorithms in these fields are able to provide parameters like the angle of skew for objects in an image and the direction of texture flow. These parameters in turn could be used in either a formulaic correction to the image or used in an iterative solver as a comparative measure of correctness.

During this search for a greater level of autonomy, a much simpler examination of arrays from a range of data files was undertaken. This search used slicing and graphing to

find simple patterns which could be used to detect the 'edges' of an array and, therefore, determine parameters for its shape. During this work, it was noted that graphing a multi-dimensional array appeared much like a graph of a periodic phenomena. When flattened to a sequence of values, arrays exhibited periodicity caused by their different dimensions. The notion of an array's dimensions creating wave patterns which could be detected and applied in an algorithmic solution lead towards the use of the Fourier transform.

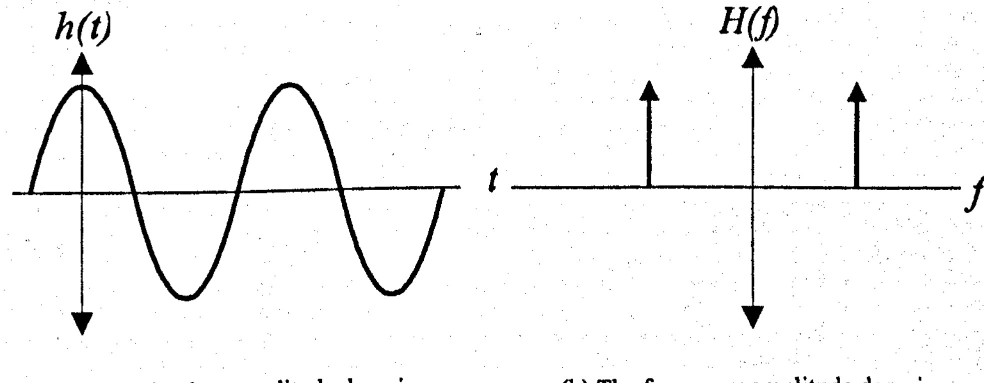
7.1 The Fourier transform

The Fourier Transform (FT) equations provide a forward and inverse transformation of values in the time domain to values the frequency domain. The forward FT decomposes a waveform into sinusoids of different frequencies which, when summed, will combine to produce the original waveform. It distinguishes different frequency sinusoids and their amplitudes. The FT stated in equation 7.1 can take the values of physical process h as a function of time t ($h(t)$), or an amplitude H as a function of a frequency f ($H(f)$) and transform them. The FT is widely used (Bracewell 1978), often in the physical sciences. Figure 7.1 illustrates a sine wave in both the time domain and the frequency domain. Figure 7.2 shows the relationship between the time amplitude and frequency amplitude domains.

$$H(f) = \int_{-\infty}^{\infty} h(t)e^{2\pi ift}dt$$

$$h(t) = \int_{-\infty}^{\infty} H(f)e^{-2\pi ift}df \tag{7.1}$$

The Discrete Fourier Transform (DFT) and the time-efficient Fast Fourier Transform (FFT) both transform from the time amplitude domain to the frequency amplitude domain for discretely sampled data. For the purposes of this research, it is more convenient



(a) The time amplitude domain (b) The frequency amplitude domain

Figure 7.1: A sine wave represented in both domains

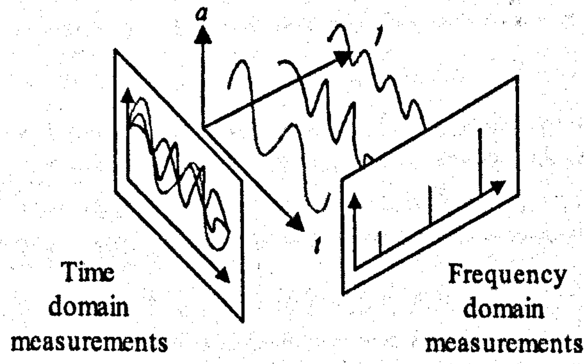


Figure 7.2: The relationship between three different measurements in the time domain and the frequency domain

to work with the real quantity output by the power spectrum density function. Parseval's theorem states that the power of a signal represented by function $h(t)$ is the same irrespective of which space it is measured in, that is,

$$Total\ power \equiv \int_{-\infty}^{\infty} |h(t)|^2 dt = \int_{-\infty}^{\infty} |H(f)|^2 df \tag{7.2}$$

To find the power in the frequency interval f and $f + df$ we require $P(f)$, or the one-sided

Shape	Peak frequency spacing
3	3.3333×10^{-1}
128	2.6041×10^{-3}

Table 7.1: The expected spacings between peaks in the frequency domain for a *rank* 3 array

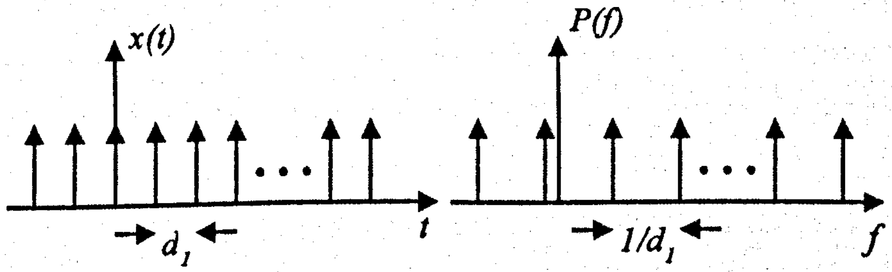
power spectral density function. For the real data of this application this is defined as

$$P(f) = 2|H(f)|^2 \qquad 0 \leq f \leq \infty \tag{7.3}$$

7.1.1 Application to data input

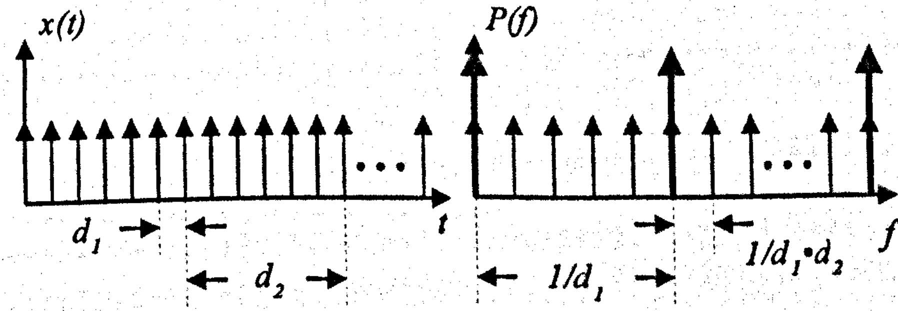
The hypothesis was that an array, when unfolded (or flattened) to 1D, will produce a periodic pattern. This periodic pattern will have frequencies in it which relate to the wavelengths of the underlying data, which in turn are directly related to the dimensions of the transformed array. A power spectrum will show these frequencies as peaks in the frequency amplitude domain, enabling peak detection heuristics to evaluate them and determine the shape of the array.

The periodicity formed by flattening an array from *rank* n to *rank* 1 appears in the power spectrum as $n - 1$ pulse trains with different spacings. Figure 7.3(b) illustrates the power spectrum of the *rank* 2 array displayed in figure 7.3(a), and figure 7.3(d) shows the power spectrum for a *rank* 3 array illustrated in 7.3(c). Each pulse train has a separation between its peaks which is related to the size of the dimension it represents and the position of that dimension in the shape of the array. The frequency spacing for a given dimension is the product of the size of that dimension and all those which vary faster in the array. Table 7.1 illustrates the expected separation between peaks in the two pulse trains when given the first two dimensions of a *rank* 3 array.



(a) $x(t)$ with periodicity d_1 in a *rank* 2 array

(b) The power spectrum pulse train $P(f)$ of $x(t)$ with a peak separation of $\frac{1}{d_1}$



(c) $x(t)$ with periodicity caused by the dimensions d_1 and d_2 of a *rank* 3 array

(d) The power spectrum pulse train $P(f)$ of $x(t)$ with two sets of peaks with separations corresponding to $\frac{1}{d_1}$ and $\frac{1}{d_1 \cdot d_2}$

Figure 7.3: The pulse trains in power spectra for a *rank* 2 and *rank* 3 array

Given the shape of the array D the expected spacing s of the nth pulse train in the power spectrum will be

$$s(n, D) = \frac{1}{\prod_{i=1}^{n} D_i}$$

where

$$1 \leq n \leq rank \qquad D = [d_1, \ldots, d_{rank}] \tag{7.4}$$

The rank of an array can be determined the number of distinct pulse trains. Each pulse train is superimposed in the power spectrum and has very different average amplitude as illustrated in the example spectrum shown in figure 7.4. All but the last rank of an array will create periodicity in the array when it is flattened. A power spectrum will exhibit $rank - 1$ pulse trains. Figure 7.4 also shows how measuring the spacing between adjacent peaks in each pulse train can be used to find an average spacing that can be used to find the arrays dimensions.

7.1.2 Limitations to data input

MatLab was chosen as a testing environment and was used to produce power spectra of thirteen datasets in the same manner. The tests progressed over four phases.

- The first phase was a feasibility test. It contained correctly interpreted arrays of values. These were flattened into sequences of values. This test was to discover if an array's dimensions would show up as peaks in a power spectrum.
- The second phase was an evolution of the first phase. Its aim was to determine if the length of the binary primitive type could also be discovered for correctly interpreted arrays of values. Again the arrays were flattened into sequences of values, but this time the individual values were also broken up into their constituent bytes. These byte values were then tested.
- The third phase took whole files as sequences of byte values, with the intention of discovering if additional information in the selection would affect the test.
- The fourth and final phase took synthetic arrays and tested for the effect of mirrored and repeated data.

The analysis of the first phase found that the full rank and shape of an array could be determined for real world data when the incoming values were correctly interpreted and

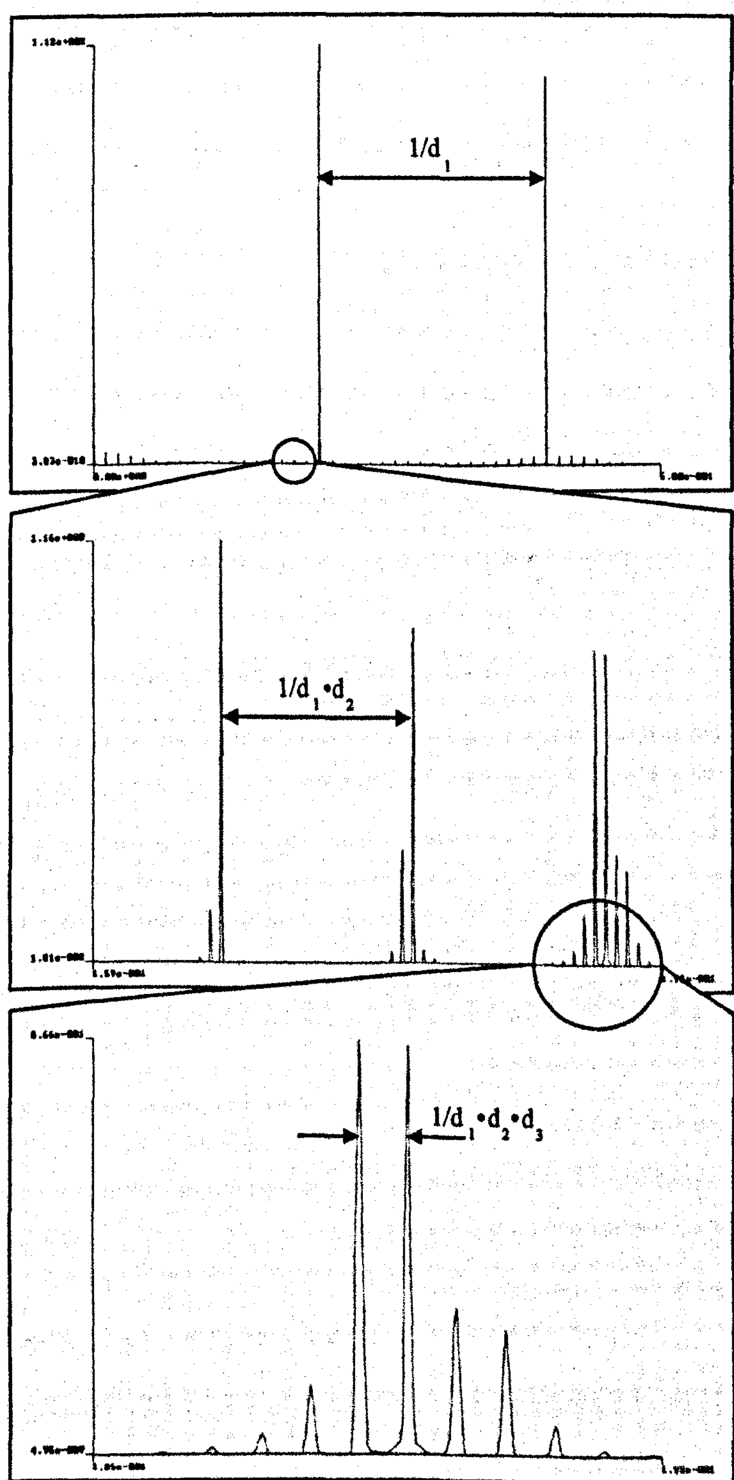


Figure 7.4: The pulse trains in the power spectrum of a real world dataset which was a *rank* 4 array. The spacings which can be used to acquire dimensions $d_1 \ldots d_3$ are marked on each view of the spectrum

the array was correctly selected. If the array was not correctly selected then only $rank - 1$ dimensions of the shape could be determined; the last dimension would not be found from the spectrum because the total number of values input determines the frequency at which the powers are output. Therefore if the selection was incorrect, the value for the last dimension, as taken from the smallest spacing in the spectrum, would be invalid.

The analysis of the second phase concluded that the length of the binary primitive type could be found for univariate arrays, but not for multivariate arrays. Arrays containing values which have a binary primitive type of more than 8 bits can be considered to have an additional dimension. For example a $rank$ 3 array of 32-bit values can be considered to be a $rank$ 4 array of 8-bit values. The size of this additional dimension is the number of bytes in the binary primitive type. The additional dimension of the binary primitive type will always precede the existing shape of the array, for example if an array of 32-bit values had a shape of 22, 22, 15 then the same array analysed as byte values would have a shape of 4, 22, 22, 15.

Univariate arrays when flattened to 1D and interpreted as bytes produce a power spectrum which has a peak of significant magnitude at the frequency which corresponds to the length of the binary primitive type. If the data was originally 8-bit, no such peak will appear.

Multivariate arrays do not produce a peak corresponding to the length of the binary primitive type. Instead they produce a pulse train for the first two dimensions of the shape whose spacing fits equation 7.4 for $n = 1$. The variable index and the length of the binary primitive type effectively forming one pulse train instead of two. For example a $rank$ 3 array of 32-bit values with a shape of 5, 22, 22 can be considered to have a shape of 4, 5, 22, 22 when interpreted as byte values. When viewed as a power spectrum the first two dimensions will appear as one which has a size of 20. Byte values will have a strong relationship with similarly positioned byte values in the same variable. However, when byte values are compared with either byte values at a different position in the same variable

or different byte values from different variables there is no relationship. Therefore, when an array of multi-byte values is interpreted as just bytes there is no strong relationship between adjacent byte values in the array. A strong relationship is only found between bytes whose separation is the product of the number of variables and the length of the binary primitive type.

The analysis of the third phase concluded that a spectrum of whole files input as byte values provided similar information as for arrays interpreted as byte values. The main difference, as found in the first phase, is that without a correct selection of the array only $rank - 1$ dimensions can be found reliably. When other arrays are present in a file, peaks are present for $rank - 1$ dimensions of each array. Arrays sharing common dimensions result in much larger peaks at frequencies corresponding to those dimensions. The effect caused by two or more arrays with different dimensions is much less distinct than for a file containing a single array.

The analysis of the fourth phase looked at mirrored data and repeated data. Mirrored data, where there is complete symmetry through one or more dimensions of the array, did not affect the spectrum in an adverse way. Repeated data on the other hand, where the content of the array in a particular dimension is exactly repeated multiple times, causes the peak spacings for the dimensions containing the repeats to increase by a factor corresponding to the frequency of the repeat.

While a power spectrum of an array can provide meaningful results for many datasets there are some limitations. If the all the dimensions of the data are required the array must contain continuous information. Discontinuous data lacks the periodicity needed to generate an output. However, discontinuous multivariate data may still yield one output, namely the number of variables. This can be found because there is a strong relationship between individual values taken from the same variable when compared with the weaker relationship between values taken from different variables. It must be noted that this effect depends entirely on a strong relationship between values in a particular variable.

Continuous data also has limitations; arrays of constant value have no periodicity and hence no information about their dimensions can be gathered from their spectra. Equally the spectrum of very noisy data or random data will hold little value for dimensioning an array.

For any array with continuous data and sufficient information content $rank-1$ dimensions of the array may be found. If the data is raw file content, i.e. byte values from the file, then the product of the length of the binary primitive type and the array's first dimension comprise the first major frequency.

Some datasets do not yield the correct shape because of artefacts in the data including dimensional repeats, too much noise or too little information content.

7.2 Summary

Power spectrum analysis offers a new avenue of research into automatic techniques for determining file input parameters. This work relates directly to the automatic discovery of the rank and shape of a dataset. Furthermore, it can also be used to detect either the number of variables in an array or the product of the number of variables and the number of bytes in the binary primitive type.

This work has illustrated how some areas of IFIT and the file input architecture could be automated given detailed analysis. Fourier analysis allows a sequence of values to be taken from a file and probed for its shape, raising the possibility of an interaction-free shape detection algorithm. As arrays are a predominant form of storage for scientific data, if this is researched to fruition a wide range of file input problems could be simplified.

Chapter 8

Conclusions and further work

8.1 Summary of achievements

This thesis has focused upon the area of data input for visualization in scientific computing. The problem's scope was outlined in chapter 1 on page 10. Within this area, many of the relevant models for data transformation and storage have been reviewed. In addition, a range of file formats found in scientific computing and the different techniques MVEs provide to input non-standard file formats have been reviewed.

In chapter 1 the aims and scope of this project were stated. The main aims were to:

- simplify the problem of creating solutions to file input problems for ViSC systems;
- find a solution which can be applied to a broad range of scientific file input problems;
- work towards an automatic solution for user file input.

The first aim has been met by the forensic approach and the supporting dataflow model put forward in chapter 4. The forensic approach uses the visual techniques presented in chapter 5 as an aid to discovering file input parameters and facilitating the application of a user's knowledge. The model and accompanying file input architecture illustrates

the interpretations and steps necessary to take raw file content and transform it into the target application's data structures. Together they are an alternative to the present 'ad-hoc' approaches. They simplify the process of solving file input problems because they take the emphasis of constructing a solution away from the format, source application and discipline, instead focusing upon the data and user knowledge available for each problem.

This theoretical work enabled the creation of IFIT, a software toolkit for creating file input solutions. IFIT enables the user to input many different scientific datasets from a wide range of file formats that store data values in one or more arrays of values. IFIT provides the user with both investigative and descriptive tools for specifying how data is stored from numerous file formats that use arrays as the predominant means of storing data values. This enables many problems which could not be solved due to a lack of user information to be solved through the discovery process outlined in chapter 5 as well as those which simply require the specification of input parameters.

Power spectrum analysis of arrays and files has been presented in chapter 7. This work stemmed from the development of the visual tools, and provides a theoretical basis for the production of modules to automatically determine an array's rank and shape using power spectra. This novel method of array metadata determination has been tested using MatLab. The further work section describes how the results from chapter 7 could be developed into a new module in IFIT.

IFIT provides the user with a distinct form of modular network solution. As a result, it has the flexibility of programmed solutions combined with the ability to rapidly develop prototypes afforded by the use of visual programming. It differs from current approaches because, not only can it describe a file in a flexible manner, but it can also be used to discover a file's content. This ability enables an IFIT user to input a wide range of problems which using the present range of file input techniques will be much harder or intractable when information about the file's content is missing.

IFIT's novel visual tools TextView, ImageView and VolumeView can all be used to inter-

actively trial different parameters for interpreting a file's content. Their use in the forensic examination of a file's content can enable the user to discover a wide range of input parameters that can aid them in visually programming a solution to their problem.

Solving and debugging file input problems using visual programming requires the user to be presented with a visual form of feedback. However, using the standard filter map render sequence of an MVE to test input solutions provides several problems for the user. First, they need to know how to choose the appropriate set of modules to visualize their data. Second, the interactions they can have with the system are usually external to the view of the data; they cannot easily steer the appropriate parameters in their solution. Finally, the performance of using a filter map render sequence can prevent the interactive trialling of parameters for large datasets. IFIT provides the user with modules which offer them direct manipulation of file input parameters and visual feedback using simple display algorithms. This allows them to rapidly trial different interpretations and metadata. This use of visual feedback provides a method for diagnosing problems in a solution, whilst also allowing the user to discover unknown metadata values and gain insight into a file's content. These visual modules provide similar functionality to the work found in computational steering, where the user adapts a parameter values and views the results in real-time. The insights found by these modules can lead to a swifter file input solution by offering unique views of the data in a timely manner.

The usage and applicability of IFIT has been demonstrated through a range of examples presented in chapter 5 and test cases which where presented and evaluated in chapter 6. The test cases illustrate the three different types of solution which can be implemented in IFIT: 'discover-use', 'single-use' and 'complete-use'. Finally, the examples show how the complexity of the file, the data and the user's needs within the visualization system effect the complexity of creating a file input solution.

IFIT is implemented in IRIS Explorer, however, any of the ideas found in IFIT can be easily implemented in MVEs capable of programmed extension. many of the algorithms

would be easily transferred to other MVEs. AVS with its comprehensive range of field mappers would the easiest MVE to extended to produce IFIT-like results.

In terms of platform dependence, IFIT's visual modules are the only modules that would require major reimplementation for other operating systems as they link directly to windows control and display routines and therefore would require changing. However, their design and use of OpenGL would minimise this to a rebinding and movement of the calling and initialization functions to the alternate operating systems callbacks, reducing the difficulty of such a platform conversion. All the other IFIT modules are standard C, or C++ and can be simply recompiled for the other platforms.

To summarise, the ideas offered in this research propose a unified method for dealing with file input as opposed to present 'ad-hoc' and format-oriented approaches which are of a purely descriptive nature. IFIT and the forensic approach presented in this thesis illustrate how file input solutions can be solved by using visual programming and interactive discovery.

The visual techniques for finding the dimensions of an array and the Fourier analysis methods both represent novel contributions to the field of scientific visualization. They prove the power of both the software architecture and model by illustrating how they have enabled processes to be solved using a range of different techniques.

File standards have often been suggested as the ultimate remedy to the problems of data compatibility and file access problems. Formats like CDF and data description languages like XML have made inroads into this problem. However, there are many reasons why these powerful methods of providing data compatibility may not be used. Equally, the new standards for scientific data like XML can be used just like traditional binary formats; producing data descriptions which cannot be correctly interpreted by other software. The notion of tools to interpret and discover file content rather than just describe file content is a powerful notion that can be applied to many fields where multidisciplinary data sources need to be accessed.

This work has solved a range of questions related to file input and provided a viable model for file input. In addition, this work has put forward several proposals for automating aspects of file input. A fully automatic method to file input may, one day, be achievable. However, this will require additional research into the other aspects of file input which have been discovered and classified during the course of this work.

The next section will describe future research and development which expands on some of the discoveries made during this research.

8.2 Further work

1. The visual feedback modules provide a format independent view of the data and so provide the user with a powerful method of discovery and error detection. ImageView and VolumeView both work best with gridded data of low a dimensionality. The other three types of structural connectivity (described in chapter 3) and the other semantic meanings for data that can be found in an array, like vector or colour data, would be more intuitively supported through visual tools which present a more appropriate visualization for the type of data involved and can discover the additional parameters which are relevant to that data.

   - Nodal and cell data would benefit the most from new visual feedback modules. A visualizer that could display both the source nodal data values and positions, as well as the linkages between nodes and cells would dramatically improve over ImageView in examining such data. Trialling this type of data would involve searching for several parameters including the linkages between nodes and the type of coordinates involved. This could take the form of directly wiring a template for cell connectivity or just plotting all the nodal points and then enabling the user to trial the different combinations of connections.
   - Flow data is another type of data which would be better supported through a

different visualization technique. Vector lines and spot noise (van Wijk 1991) can both be used to generate views of vector components that provide an excellent feedback method for the forensic discovery of arrays containing vector data. Modules similar to ImageView and VolumeView could be used to generate interactive flow field feedback that could enable the user to discover input parameters in a similar manner to ImageView. This has been trialled using existing IRIS Explorer modules for spot noise generation. While lacking the interactive element found in ImageView the visual feedback did provide similar artefacts which could be used to determine file input parameters.

- Graphs, tables and other simple graphic feedback tools to illustrate the content of an array are also useful tools for investigating the content of a file. Adding modules which enable the file to be examined using a range of different interpretations would offer useful functionality for data input.

2. Combining all these interactive tools with a view manager that can present different sections of the file to the user may provide performance benefits in discovering file input parameters, especially if it offers comparative views of the same data under different interpretations.

3. Chapter 7 described the use of power spectrums as a means of finding the dimensions of an array. This offers the possibility of an automatic solution to the problem of determining the shape of an array. By linking a power spectrum of the data with a peak detection algorithm to find peak magnitudes and spacings; the resulting information could be used to find the dimensions of an array with little or no user interaction. This could then lead to a module which either replaces ChangeDimLat or sets its rank and shape parameters.

4. Monolithic input tools while inflexible are simple to use. A 'novice friendly' file input tool could be created using IFIT by adding an extra interface layer that enables users to specify their data in a step-by-step manner and then generate appropriate

IFIT networks for their problems. The interface to this tool should aim to describe as many aspects of the user's file in a visual manner using icons and as few parameters as possible for any given problem. The module network generated dynamically or taken from a library of solutions that can be described by the user's choices. Using IFIT as the underlying system for such a tool would enable the range of solutions to be extended, the resulting networks would be user customisable and unlike existing tools unknown parameters could be handled through connecting visual feedback modules into the network at the appropriate places.

5. Language-based file formats and those using a data dictionary, or data description language (DDL) (as described in chapter 2, present a different methodology for storing data. While this methodology fits into the dataflow model for file input, and the file input architecture, IFIT has no modules which can effectively deal with this form of storage. These files often contain the equivalent of fixed record data, although the DDLs can describe much more in a flexible manner. There are several requirements for modules to handle this type of storage. These include the ability to identify separate values in the file, group them into variables and convert them into a usable binary form.

   One approach would be to produce 'file compiler' using techniques taken from existing compilers (See Trembley and Sorenson (1985) for a description on the theory of compilers). This approach would use regular expressions and production rules to first discern different types of numeric value and formatting tag, and then compiled the discovered data values into a binary form which could then be accessed by IFIT. Such a tool could include some primitive rules, which define how values like floating point numbers and integers are represented. Other simpler approaches include producing simple parsers, which when meeting a user-defined tag can perform a specified operation. One example would be gathering a list of values that match a particular tag, e.g. "COORDINATE =".

6. Future tools should aim to enhance the forensic approach by providing tools to search a file to find the location and description of file input parameters and metadata that have been discovered. The resulting set of matches could allow the user to then specify how the file is laid out in a reusable manner, overall this would have the effect of simplifying the production of reusable module networks.

7. Compression was excluded from the scope of this project in the early stages due to both the additional complexity and the generally lower usage of compression in the user-defined, field-specific and non-standard file formats found in the scientific community. Different compression techniques can have very different performance with different types of data. Their effect on the information content also differs with some techniques 'lossy' and others 'lossless' in their effect on the data involved. Compression can also radically change the content, structure and meaning of a dataset. For example, RLE data still has the same primitive type and values meanings as the source data. RLE replaces repeated values in the dataset with tags which define regions of constant value. Conversely Lempel-Zif (LZW) encoding radically alters the meaning of the bytes from their original form and changes the files values meanings into a selection of references into a dictionary of values. The interpretation pipeline and dataflow model for file input do not explicitly prevent the interpretation of compressed files. It may be possible to take some commonly used compression schemas and provide them as modules operating at their relevant place in the pipeline.

8. The model and architecture as they stand provide a unidirectional pipeline which flows from a file's content to the application's data structures. This could be extended to produce a bi-directional model, including file output as well as input. This would use a range of processes which offer the inverse functionality of those found in the file input pipeline. Starting with metadata and data structures from the application and interpreting them into file content which can then be directly out-

put. This could potentially solve a range of data exchange problems which occur between scientific systems.

9. Finally, the techniques applied in the design of IFIT may be applied in the gathering of digital evidence. Future work in this field would involve the development and feasibility testing of tools to enable the discovery of 'hidden data' within files on suspects hard drive. For example, data hidden in media files. Methods of discovery found in the dynamic feedback element of IFIT could be developed to enable a user to search for the structures which would be present in such files that would occur in addition to their 'visible' content.

References

Adobe Systems Incorporated, Taft, E. and Walden, J.: 1990, *PostScript language reference manual*, 2nd edn, Addison-Wesley.

Bergeron, R. and Grinstein, G. G.: 1989, A Reference Model for the Visualization of Multi-dimensional Data, *Proceedings of Eurographics '89*, Elsevier Science Publishers B.V. (North-Holland), pp. 393–399.

Berges, J. C.: 2002, Support of WMO binary format (BUFR and GRIB), Internet : `citeseer.ist.psu.edu/berges02support.html`.

Bracewell, R. N.: 1978, *The Fourier Transform and Its Applications*, electrical and electronic engineering, Second Edition edn, McGraw-Hill Book Company.

Brodlie, K.: 1993, *Animation and Scientific Visualization*, Academic Press, chapter 8: A classification scheme for scientific visualization, pp. 125–140.

Brodlie, K., Carpenter, L., Earnshaw, R., Gallop, J., Hubbold, R., Mumford, A., Osland, C. and Quarendon, P.: 1992a, *Scientific Visualization: Techniques and Applications*, Springer Verlag, pp. 37–85.

Brodlie, K., Carpenter, L., Earnshaw, R., Gallop, J., Hubbold, R., Mumford, A., Osland, C. and Quarendon, P.: 1992b, *Scientific Visualization: Techniques and Applications*, Springer Verlag.

Brodlie, K., Gallop, J., Grant, A., Hanswell, J., Hewitt, W., Larkin, S., Lilley, C., Morphet,

H., Townend, A., Wood, J. and Wright, H.: 1995, *Review of Visualization Systems*, number 9 in *Technical Report Series*, Advisory Group on Computer Graphics.

Butler, D. and Pendley, M.: 1989, A visualization model based on the mathematics of fibre bundles, *Computers in Physics* 3(2), 45–51.

Carey, R., Bell, G. and Marrin, C.: 1997, ISO/IEC 14772-1:1997 Virtual Reality Modeling Language (VRML97), *Technical report*, The VRML Consortium Incorperated.

Chatzinikos, F. and Wright, H.: 2003, Enabling Multipurpose Image Interaction in Modular Visualization Environments, *in* R. F. Erbacher, P. C. Chen, J. C. Roberts, M. Grlin and K. Brner (eds), *SPIE: 5009, Visualization and Data Analysis*, p. 455 462.

Core AVS/Express and the Object Manager: 2004, Internet: `http://help.avs.com/Express/doc/help_63/books/dr/drfo.html#26321`.

Dyer, S.: 1990, A Dataflow Toolkit for Visualization, *IEEE Computer Graphics and Applications* **10**(4), 60–69.

Extensible Markup Language (XML) 1.0: 1998, *Technical report*, WC3.

Felger, W. and Schroder, F.: 1992, The Visualization Input Pipeline - Enabling Semantic Interaction in Scientific Visualization, *Eurographics* **11**(3).

Fitzgerald, P., Berman, H., Bourne, P. and Watenpaugh, K.: 1993, The Macromolecular CIF Dictionary, Internet: `citeseer.ist.psu.edu/fitzgerald93macromolecular.html`.

Foley, J. D., van Dam, A., Feiner, S. K. and Hughes, J. F.: 1996, *Computer Graphics: Principles and Practice*, The systems programming series, Second edition in C edn, Addison Wesley, pp. 334–335.

Gallop, J.: 1994, *Visualization In Geographical Information Systems*, New York: John Wiley & Sons, Ltd., chapter 6: State of the Art in Visualization Software, pp. 42–48.

Goucher, G. and Mathews, J.: 1994, Common Data Format CDF, Internet: `http://nssdc.gsfc.nasa.gov/cfd/html/tech_brief.html`.

Haber, R. B., B.Lucas and N.Collins: 1991, A Data Model for Scientific Visualization with Provisions for Regular and Irregular Grids, *Proceedings of Visualization '91*.

Haber, R. B. and McNabb, D. A.: 1990, Visualization Idioms: A Conceptual Model for Scientific Visualization Systems, *Visualization in Scientific Computing* pp. 74–93.

Hall, S. R., Allen, F. H. and Brown, I. D.: 1991, The Crystallographic Information File (CIF): a New Standard Archive File for Crystallography, *Acta Crystallographer* pp. 655–685.

Hamming, R. W.: 1962, *Numerical Methods for Scientists and Engineers*, McGraw-Hill New york.

Hierarchical Data Format: 2000, Internet: `http://hdf.ncsa.uiuc.edu/`.

IRIS Explorer User's Guide (Windows NT/2000): 2000, Internet: `http://www.nag.co.uk/visual/IE/iecbb/DOC/html/nt-ie5-0.htm`.

Johnson, C. R. and Parker, S. G.: 1994, A computational steering model for problems in medicine, *In Supercomputing '94*, IEEE Press, pp. 540–549.

Kendrew, J. C., Bodo, G., Dintzis, H. M., Parrish, R. G., Wyckoff, H. and Phillips, D. C.: 1958, A Three-Dimensional Model of the Myoglobin Molecule Obtained by X-ray Analysis, *Nature* (181), 662–666.

McCormick, B. H., DeFanti, T. A. and Brown, M. D.: 1987, Visualization in Scientific Computing, *Computer Graphics* **21**(6).

Network Working Group: 1987, XDR: External Data Representation Standard, *Technical Report RFC 1014*, Sun Microsystems Incorperated.

Network Common Data Form: 2000, Internet : `http://unidata.ucar.edu/packages/netcdf/index.html`.

Nielsen, J.: 1993, *Usability Engineering*, AP Proffessional, chapter Usability Heuristics.

Open Inventor: 1993, Internet : `http://oss.sgi.com/projects/inventor/`.

Osland, C.: 1992, *FRAMEWORK*, Springer Verlag, chapter 2, pp. 15–35.

Pearsall, J. (ed.): 1998a, *New Oxford Dictionary of English*, Oxford University Press, pp. 2066–2067.

Pearsall, J. (ed.): 1998b, *New Oxford Dictionary of English*, Oxford University Press, p. 995.

Richards, F. M.: 1968, The matching of physical models to three-dimensional electron-density maps: A simple optical device, *Journal of Molecular Biology* **37**, 225–228.

The: 2000, *IRIS Explorer Module Writer's Guide (Windows NT/2000)*, 5 edn.

The Jargon Dictionary : Terms : The M Terms : middle-endian: 2003, Internet : `http://info.astrian.net/jargon/terms/m/middle-endian.html`.

Trembley, J.-P. and Sorenson, P. G.: 1985, *The Theory and Practice of Compiler Writing*, Computer Science Series, McGraw-Hill International Editions.

Tufte, E. R.: 1983a, *The Visual Display of Quantitative Information*, Graphics Press.

Tufte, E. R.: 1983b, *The Visual Display of Quantitative Information*, Graphics Press, p. 28.

Tufte, E. R.: 1983c, *The Visual Display of Quantitative Information*, Graphics Press, pp. 40–41.

Upson, C., Jr., T. F., Kamins, D., Laidlaw, D., Schlegel, D., Vroom, J., Gurwitz, R. and van Dam, A.: 1989, The Application Visualization System : A Computational Environment for Scientific Visualization, *IEEE Computer Graphics and Applications* 9(4), 30–42.

van Wijk, J.: 1991, Spot noise texture synthesis for data visualization, *in* T. W. Sederberg (ed.), *Proceedings of Computer Graphics SIGGRAPH 91*, Vol. 25, pp. 263–272.

Webster's Master English Dictionary: 2002, Midpoint Press, p. 454.

Wells, D., Greisen, E. and Harten, R.: 1981, FITS: A Flexible Image Transport System, *Astronomy and Astrophysics Supplement Series* 44, 367–370.

West, J. J.: 1999, Images and Reversals, Internet : `http://www.siggraph.org/publications/newsletter//v33n1/columns/west.html`.

Woo, M., Neider, J. and Davis, T.: 1997, *OpenGL Programming Guide*, Second Edition edn, Addison-Wesley Developers Press.

Wood, J.: 1998, *Collaborative Visualization*, PhD thesis, School of Computer Studies, University of Leeds.

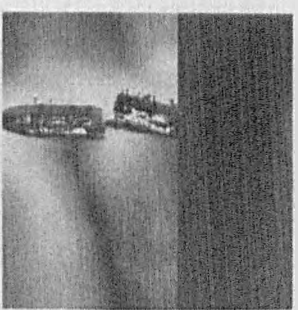


Figure 5.24: Single distinct vertical break of continuity

Single distinct vertical break of continuity This artefact, illustrated in figure 5.24 can occur when the start of the array is incorrectly positioned in the file. The effect is produced by either unwanted values leading into the array or by missing some of the array's data. This in turn alters the interpretation of the values in the array, moving what would be the wrapped edges of the array's first dimension to another location in the first dimension of the interpreted array. This results in the edges of the original array becoming adjacent values at some mid point in the array, and because the edges are not continuous, they will form a visible break in continuity. Correcting this artefact requires the user to alter the start point for the selection until the discontinuity coincides with one of the edges and all the array data is present. It is important to note that the size of the array must be checked to prevent loss of data or the addition of non-data values into the array.