Role of Visual Languages in Developing Image Analysis Algorithms

M. S. Atkins, T. Zuk and B. Johnston, Simon Fraser University and TRIUMF
T. Arden, Logical Vision Ltd., Vancouver B.C. Canada

Abstract

Developers of new image analysis algorithms typically require an interactive environment in which the image data can be passed through various operators, some of which may involve feedback, synchronization, merging and conditional control strategies. This paper describes the requirements of visual languages for developing new algorithms, and how several visual languages meet these requirements. We then show how the data flow methodology of a visual software development tool called WIT has greatly simplified the prototyping and testing of our new image registration and segmentation methods. We describe the main features of WIT, and show data flow graphs for two medical image analysis algorithms.

1 Introduction

Developers of new algorithms and techniques for the analysis of images typically require various operations to be performed on the image data. The data is passed through various operators, some of which may involve feedback, synchronization, merging and conditional control strategies. In a data flow visual language, data typically flows through operator modules connected together with links to describe the dataflow algorithm. This eliminates the need for the programming to start from scratch, because low-level image operators and graphical user interfaces are provided. One especially useful feature for algorithm development is the ability to place a probe to verify and analyze the output of any operator.

We use all of the above features in a visual dataflow software development tool called WIT^{TM} [1] which has been designed for image processing applications. WIT is a commercially available product implemented on SUN SPARC computers and on PCs. It has been in use in our research environment since October 1992, and we are an alpha test site with the developer, Logical Vision Inc. of Vancouver. The WIT dataflow environment provides not only a large library of I/O and image processing operators, but also a library of data flow control operators. WIT, like other visual data flow languages, allows the user to draw a graph of the dataflow through named operators using "point-and-click" icons and links; in WIT this graph is called an *igraph*. The links connect portals between operators that allow the data to flow through the igraph. Figure 1 contains a simple igraph utilizing the *if* control operator. Images A and B enter into independent *stats* operators that calculate the mean ($\bar{x}$), standard deviation (σ), floor ($\lfloor\ \rfloor$), and ceiling ($\lceil\ \rceil$) of the pixel values. The standard deviation portal from each operator is connected to the *conditional* operator whose condition parameter is set to ">" so that it outputs a 1 if the top input is greater than the bottom input, and 0 otherwise. This result then goes into the *if* operator which uses it to direct the flow of data C. The *see data* magnifying glass represents a probe which lets the user examine the data passing through the link. A probe can be put on any link by clicking on it with the mouse.

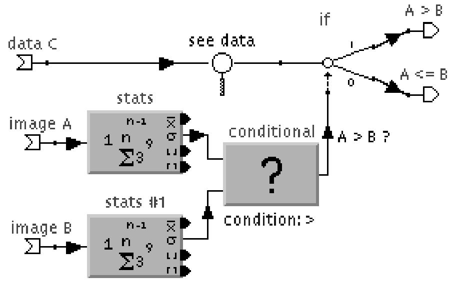

Figure 1: A Simple Igraph

The components of an igraph are taken from libraries of icons which constitute the elements of the dataflow language.

Section 2 of this paper describes requirements for visual data flow programming languages for image processing.

Section 3 discusses how these requirements are met by several current systems. Section 4 describes two applications, registration of two images and segmentation of an MRI brain image into several different tissue types, using a powerful new visual programming language, WIT. Section 5 presents a summary and future work.

2 Visual Dataflow Programming Language Requirements for Image Processing

In Hils' comprehensive survey paper on Dataflow Visual Programming Languages [7], he describes a data flow visual programming language as *powerful* if it provides the necessary programming constructs and predefined functions so that a user of the language can tackle large, complex problems in the language's application domain [8]. Otherwise the user must make a mental leap from the visual data flow environment into the textual sequential-programming environment offered by C or Fortran, in order to program new algorithms or to customise existing library modules. Hils claims (informally) that iteration, procedural abstraction and a large and diverse library of predefined functions for use as building blocks, are all *necessary* for a data flow visual programming language to be powerful. Furthermore, higher-order functions (operators whose parameters may be other operators) are considered *desirable* for making a language more powerful. This definition differentiates visual programming languages from visual languages; you can only program in a visual language if it provides the powerful features.

We would like to add further desirable features from our experience with using different systems:

1. the ability to speed up computation through specifying certain modules may execute on certain machines (placing compute-intensive calculations on fast machines), and through exploiting the inherent parallelism in data flow graphs.

2. the ability to conveniently view the data as it flows along a link, in whatever form is appropriate (eg. to see an integer or an image).

3. the ability to view the presence of data flow along a link and view execution of modules through user-controllable animation of the graph (especially useful during debugging).

4. the ability to perform *demand driven* execution (described below)

5. efficient execution of the visual program – that is, minimal performance penalty for using the data flow visual program environment.

For demand driven dataflow computing, execution of a node occurs when a downstream node requests data from upstream nodes over its input arcs. All data still flows downstream, but requests flow upstream. This leads to more efficient but also more complex implementations. It is not essential for a visual programming language, but it certainly is desirable to provide for demand driven execution as it is very quick and efficient.

The performance efficiency depends on on the implementation language (for example, Smalltalk is proverbially slower than C) and on the dataflow computing model used in the implementation. Herath has described various dataflow computing models [6] which include static computing, recursive dynamic computing, tagged token dynamic computing, eduction computing and not(operation) computing.

In the static computing model, concurrent reentrance of nodes is forbidden, whereas in the dynamic models several instances of a node can be fired at a time, thus increasing the parallelism. Yet further parallelism is allowed in the "tagged token" model, where several data values can exist on a link at any time. It appears that all the dataflow visual languages discussed here use the static computing model, as they are typically implemented on a single processor which cannot take advantage of the inherent parallelism in dataflow graphs. Eduction computing combines dataflow with demand-driven execution; this might provide useful performance enhancements, but has not been used to implement the dataflow of any of these visual languages.

In the not(operation) computing model, parallel computations use special conditional constructions consisting of two parallel complementary computations, where the data flow is split into two parts, one passing to the operation and the other to the not operation. Only one of these operations succeeds to give an output value; the other operation "freezes" its input data. Hence exactly one data stream is preserved. This technique exploits the data flow parallelism and reduces computation time.

These implementation techniques have a profound impact on the ultimate usability and performance of the system, but in this paper we confine ourselves mostly to a discussion of the visual programming language features – the qualitative not quantitative issues.

3 Visual Dataflow Languages for Image Processing

There are several visual language toolkits available for image processing such as AVS [2], Explorer [4], WIT [1], HI-VISUAL [9], KBVision Version 3.0 [14] and Khoros [16] (which uses the Cantata Visual Programming Environment). The IDL system [10] is also popular for image processing, but is not visual, in that the data flow is programmed textually (like the earlier version of KBVision). The 3DViewnix software [17] provides a demand flow environment for displaying images, but it is not a visual language. All these toolkits provide facilities for pixel-based image processing, but several do not contain the powerful control flow operators necessary for creating complex dataflow programs.

All of the above languages provide a procedural abstraction mechanism, meaning that an entire graph may be considered a procedure and condensed into a single node; arcs going to and from the node may then be considered arguments to the procedure. As Hils notes, procedural abstraction is important for two reasons: first, it saves screen space, making a visual program more compact; second, it allows the programmer to view the program at a higher level, without numerous details. In most systems it is provided through "zooming in" on a function icon in order to see its internal structure.

Furthermore, all of the above systems provide a comprehensive library of predefined functions for image processing and display, including libraries for I/O, GUIs, point and image transformations, morphological processes, filtering, transforms (Fourier and others), measuring, and segmentation. However the focus varies between them (e.g. some provide better region-of-interest definition and analysis features than others).

The major difference between the different systems with respect to the power of their visual programming language arises in the ease of use and availability of facilities for iteration.

The language AVS is not powerful in Hils sense, as there are no facilities for iteration within the network editor. Hence it is difficult to tackle complex problems in the visual mode. In AVS the data normally flows from top to bottom, except in certain special circumstances where such *upstream data* is necessary. In AVS upstream flow is described as being necessary for mouse control for data highlighting objects being displayed, and for picking structures being viewed. Therefore only the viewing and display operators allow upstream flow. With this limited, restricted view of upstream flow (the upstream flow is even displayed in the graph only through extra clicking on the by-default-invisible ports), iteration can only be provided *within* a module rather than *between* modules in the graph. It is clear that for developing any new iterative algorithm such as a new image segmentation algorithm, or for an extension of an existing algorithm, the AVS user must leave the visual language environment and write the new code using a textual control flow language like C or C++. With respect to the desirable features for a visual programming language, AVS does include the ability for the user to place particular modules to execute on particular hosts, and it does not appear to have a significant performance overhead. However, it is not especially convenient to view data in transit along links; the user has to split the link and add another viewer module to display the data.

Explorer (available on Silicon Graphics machines) is like AVS, in that it lacks iteration, so visual programming cannot be used to develop new algorithms.

The systems WIT, HI-VISUAL, Khoros and KBVision all allow iteration through while-loop or for-loop constructions, so at first glance these languages might all be considered "powerful" visual programming languages. However, both the Khoros and KBVision systems are actually control flow, not data flow based, and using the visual language to perform upstream programming is not easy. For example, in Khoros, the data is in files and data objects are not persistent; as soon as they are read, they are gone from the link unless they are re-read in from the files. The icons are implemented as stand-alone programs and are executed by a UNIX fork, which is why Khoros is so large (160MBytes) and slow. Hence the user has to be aware of the nature of the implementation in order to achieve the desired execution and to achieve acceptable performance, and it is tough to use the visual language provided by both Khoros and KBVision to write any low-level code.

HI-VISUAL is an interesting visual programming language for image processing (and, more recently, for office work applications). It is more thoroughly object-oriented than any of the other systems, in that the original "box-line" visual interface has been supplemented by icons which may represent objects (both data and code). The "algorithm" to be executed may be represented by overlapping icons on the graph; data may flow between overlapping icons if appropriate matching class methods are found. This language may prove to be a very powerful visual programming language, but it exists only in a prototype form, and its performance may be slow because of the characteristics of the implementation.

A powerful low-level visual language is provided by Hils, who designed a prototype dataflow visual programming language called Datavis for scientific visualisation applications [8]. Datavis is based on a general visual programming language, Show-and-Tell [15]. Datavis overcomes the difficulties of using the visual programming mode to program low-level actions, through integration of the visual level with the predefined library functions in a truly data flow environment. However, Datavis is very slow, and like HI-VISUAL, exists only as a prototype language.

WIT's control operators provide capabilities closer to the nature of an entire dataflow programming language such as NL [5]. Figure 2 illustrates some of the operators from various libraries.

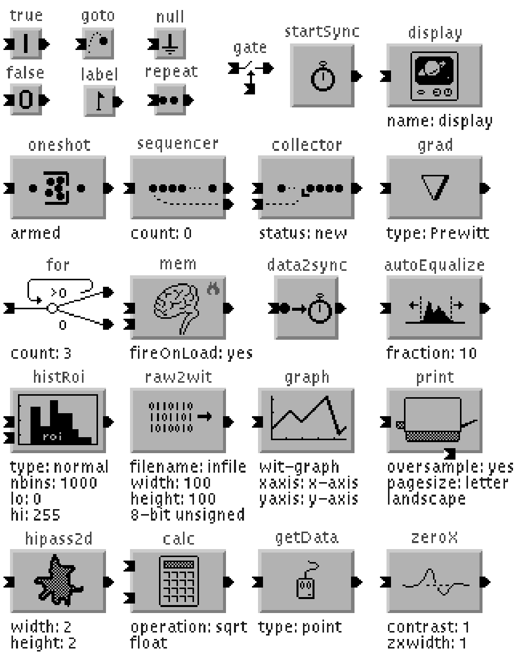

Figure 2: Some WIT Library Operators

Users may create their own operators hierarchically from other existing operators, hence programming within the visual language.

With respect to the desirable properties, there are some predefined higher-order functions such as the CONDITIONAL and ALUOP modules which allow the user to select the operator to be performed on the operands, from a predefined list. The CONDITIONAL operator performs conditional testing on two inputs, and the output type is an integer and is set to 1 if the conditional is TRUE and 0 if FALSE. The condition can be chosen from $==, >, <, !=, \leq, \geq$. The ALUOP operator provides a kind of higher order function, in that it performs certain binary operations on a pixel-by-pixel basis, given two input images, A and B. Supported operations include +, -, *, / and logical operations. Furthermore, X and Y can be used to access the pixel's particular column and row, and an expression can be used to determine the result. For example, the expression $(Y \geq 5)?A : B$ will produce an output image where the first 5 rows are identical to the first 5 rows of the image B, otherwise the output image corresponds to A. It is also possible to define a new higher order function within the visual language, although not entirely straightforward.

One of the greatest advantages of WIT for prototype development arise because very low-level actions can be programmed in the visual mode. For example, the recursive FACTORIAL function can be programmed as shown in Figure 3. Note that as recursive igraphs are not directly allowed, a dummy module is required which expands to the factorial module shown in the figure.

It is very easy to monitor data flowing along links, through clicking on the link and selecting a probe, shown as a magnifying glass in the igraph. When data flows along the link, it is displayed in a new window, in an appropriate way for its type. Execution of the dataflow algorithm can be animated at user-controlled speeds, which is very useful for debugging. WIT allows efficient execution of igraphs in parallel by distributing computation across a network of workstations, so that compute-intensive programs can be accelerated. WIT inherently provides parallelism that is a major benefit of dataflow programming [3]. The parallelization can be scheduled automatically by WIT, or the user can specify a machine for an operator to execute on. WIT's provisions for parallel processing, along with the constructs to create low-level to high-level programs form a powerful visual programming environment.

4 Applications using WIT

4.1 Registration

One of our applications is the 3D registration of PET images into a common coordinate space using surface matching [18]. The approach is similar to Jiang et al.'s hierarchical technique [12]. The igraph in Figure 4 represents the entire registration process.

The *read raw data* loads the voxel data for a scan, and the *Scan Info* operators specify the inter-voxel distances. The slices of each scan are individually processed in the hierarchical operator *thresh closeBin* to extract contours representing the surfaces to be matched. The internal structure of this operator is illustrated in Figure 5. The threshold creates a binary segmentation of the data which is smoothed in the *closeBin* operator which performs a dilation followed by a constriction the input number of times (*nIterations*). The outline of the resulting blob is then taken as the surface contour. The "highlighting"

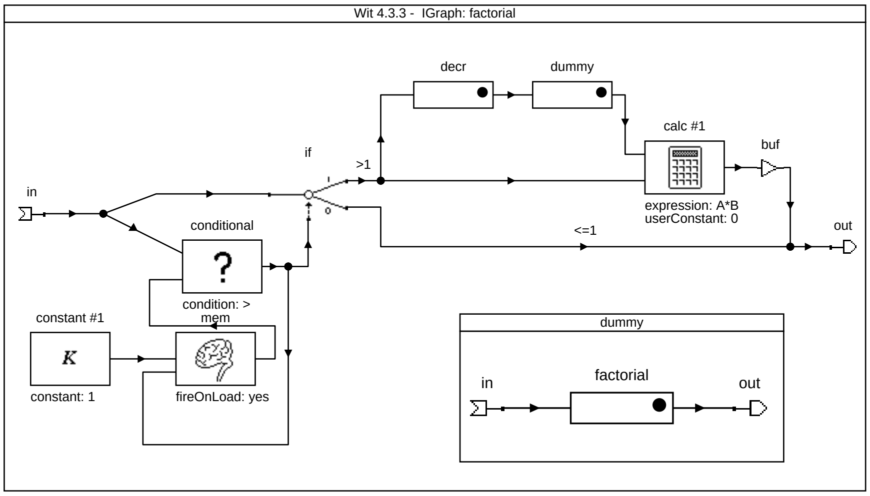

Figure 3: Recursive FACTORIAL Operator

of the operator parameters indicates that they are propagated up to a higher hierarchical level. The *Registration* operator then utilizes a point representation of each surface and performs univariate relaxation on rigid body transformation parameters. The transformation (RBT) that creates the closest match between the two surfaces is then used by the *resample* operator to extract slices from Scan B that correspond to those of Scan A. This represents only one possible configuration of the registration igraph. Different operators for segmentation, matching, or verification can easily be interchanged to better accommodate specific cases.

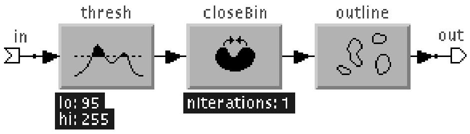

Figure 5: Hierarchical Operator *thresh closeBin*

4.2 Segmentation

Another application is tissue segmentation in MRI images where we are looking at lesion volumes in both tumour and Multiple Sclerosis patients. Figure 6 shows the igraph representing our implementation of an algorithm we have adapted from Besag's iterated conditional modes (ICM) in 3D [11, 13]. We have used dual echo MRI images obtained from scans of Multiple Sclerosis patients for our experiments. The *icmInitialize* operator allows the user to select regions of interest from a single specified starting image. Following the initialization phase, each image spectrum, together with histograms of tissues selected in the initialization phase, and the total number of tissues given to be present in the volume, are passed to the segmentation operator (*Segment Spectrum*). Following separate segmentations of each spectrum, the results are combined in the operator *combinePVolSpectra* to form the true classification for this data set.

Our segmentation technique determines, for each voxel, a set of probabilities that a given voxel x, is of tissue type j. This segmentation is obtained through analysis of tissue histograms and neighbourhood interactions for each voxel. Expansion of the *Segment Spectrum* operator, shown in Figure 7, illustrates how the image volume is iterated through the segmentation routine. The *createPVol* (create partial volume) operator computes an initial partial volume segmentation based on the input histograms. The *icmSeg* operator accepts the image vector, the initial segmentation, a set of neighbourhood interaction parameters, and the histograms describing tissue profiles as input. The neighbourhood interaction parameter matrix, describing relations between tissue types, is created by operator *icmGetBij*. Note how the input parameters are maintained in memory operators for future iterations of the algorithm. The *icmSeg* operator repeatedly applies the ICM equation to each voxel in the input volume until convergence is achieved or until the number of iterations of the algorithm reaches the input parameter maximum (*icmIter*). When convergence has occurred, or the number of iterations of ICM reaches the level of the *icmIter* parameter, a recalculation of neighbourhood interaction parameters and histograms may be performed in order to refine the segmentation. Following recalculation of these parameters the *icmSeg* operator is reactivated and the process is repeated. The computational burden of applying this algorithm in 3D is reduced in WIT by assigning

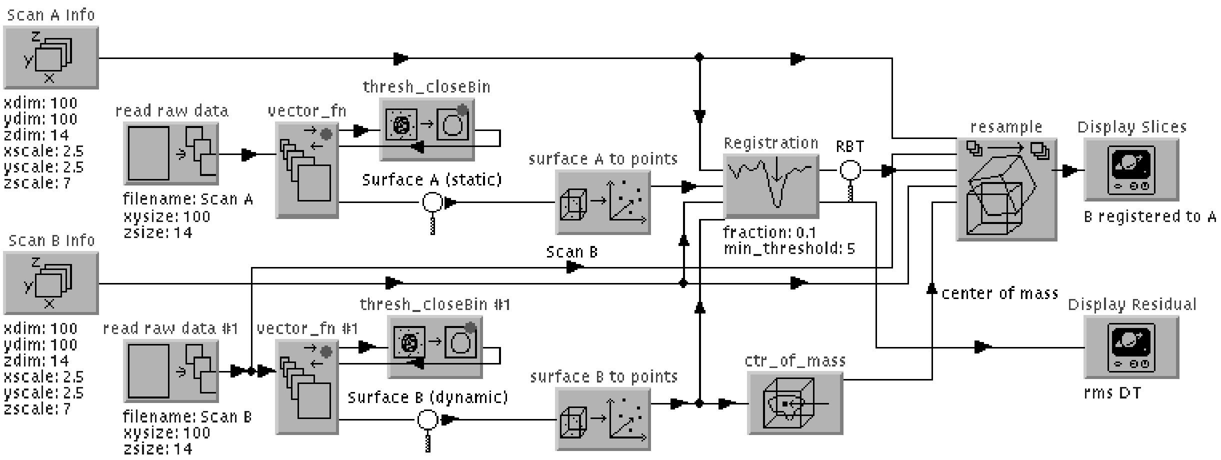

Figure 4: The Registration Process

multiple processors to the segmentation of individual image spectra.

5 Summary and Future Work

Essential features for algorithm development in the visual environment include iteration, a large library of predefined operations and procedural abstraction mechanisms. We have found that visual programming in a dataflow language is very suitable for analysis of medical images, enabling rapid prototyping of new algorithms by the ease of manipulation of links and the easy sharing and re-use of software operators.

We are actively investigating ways to improve the use of screen space, through a variable zoom graphics technique, which enlarges parts of the igraph while shrinking others. This will allow hierarchical operators to be expanded "in-line", hence maintaining the context of hierarchical nodes during their expansion.

The underlying computational model determines the degree of parallelism, and we are experimenting with designs from the visual language down to the implementation level to determine how best to implement control flow constructs like a CASE statement, and efficient scheduler algorithms. We are investigating methods for implementing the dataflow automatically on a distributed network of workstations, and on workstations with multiple cpu's, to speed up compute-intensive problems further. We are also investigating how to make WiT operations more object-oriented.

6 Acknowledgments

Partial funding for this work was provided by the Natural Sciences and Engineering Research Council of Canada, and by the TRIUMF nuclear facility.

References

[1] T. Arden and J. Poon. *WiT User's Guide.* Logical Vision Ltd., 3700 Gilmore Way Burnaby, B.C., Canada, 1993.

[2] AVS. *AVS Technical Overview.* Advanced Visual Systems Inc., 300 Fifth Ave., Waltham, MA 02154, 5.0 edition, 1992.

[3] A. L. Davis and R. M. Keller. Data flow program graphs. *IEEE Computer*, 15(2):26–41, February 1982.

[4] Explorer. *IRIS Explorer User's Guide.* Silicon Graphics Inc., 1.0 edition, 1991.

[5] N. Harvey and J. Morris. NL a general purpose visual dataflow programming language. Technical report, Department of Computer Science, University of Tasmania, Hobart, Tasmania, Australia, October 1993.

[6] J. Herath, Y. Yamaguchi, N. Saito, and T. Yuba. Dataflow computing models, languages, and machines for intelligence computations. *IEEE Transactions on Software Engineering*, 14(12):1805–1828, December 1988.

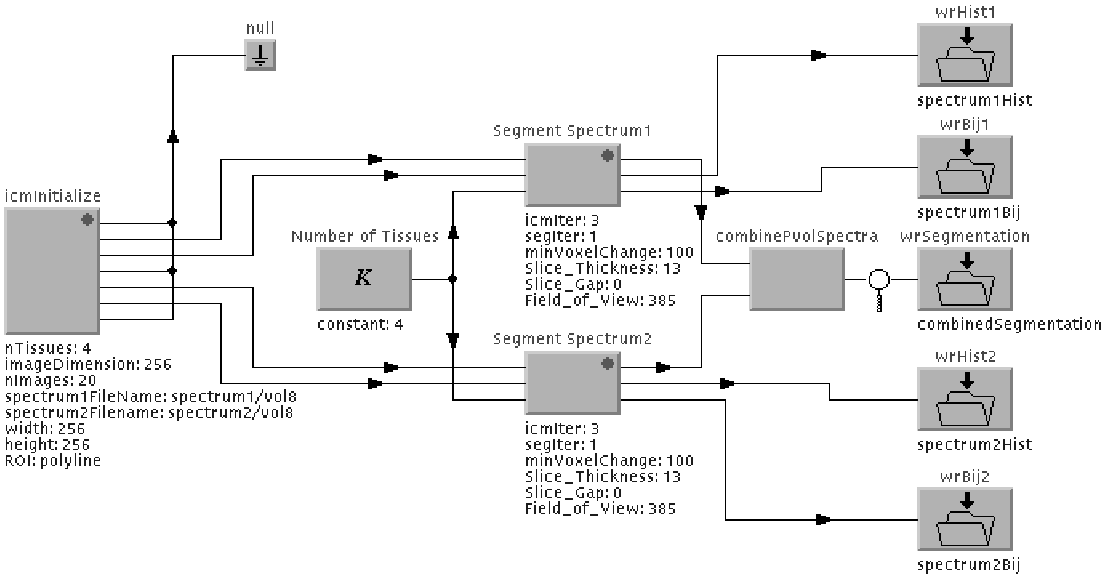

Figure 6: The Segmentation Process

[7] D. D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages and Computing*, 3:69–101, March 1992.

[8] D. D. Hils. *A Visual Programming Language for Visualization of Scientific Data.* PhD thesis, The University of Illinois at Urbana-Champaign, Department of Computer Science, July 1992.

[9] M. Hirakawa, M. Tanaka, and T. Ichikawa. Hi-visual iconic programming environment. In T. Ichikawa, E. Jungert, and R. R. Korfhage, editors, *Visual Languages and Applications*, pages 121–145. Plenum Press, New York, N.Y., 1990.

[10] IDL. *Interactive Data Language.* Research Systems, Inc., 2995 Wilderness Place, Boulder CO 80301, 3.0 edition, 1993.

[11] Besag J. On the statistical analysis of dirty pictures. *Journal of the Royal Statistical Society*, 48(3):259–302, 1986.

[12] H. Jiang and R. Robb. Image registration of multimodality 3D medical images by chamfer matching. In *SPIE - Biomedical Image Processing and Three-Dimensional Microscopy*, volume 1660-35, pages 356–366, San Jose, CA, USA, 1992.

[13] B. Johnston, M. S. Atkins, and K. S. Booth. Three-dimensional partial volume brain tissue segmentation of multispectral magnetic resonance images using stochastic relaxation. In *IS&T/SPIE 1994 International Symposium on Electronic Imaging: Science and Technology*, pages 268–279, 1994.

[14] KBVision. *Knowledge-Based Vision Systems: Visual Programming Environment.* Amerinex Artificial Intelligence, Inc., 409 Main St., Amherst, MA 01002, 3.0 edition, 1993.

[15] T. D. Kimura and P. McLain. Wucs-86-4: Show and tell user's manual. Technical report, Department of Computer Science, Washington University, St. Louis, MO 63130, 1986.

[16] J. Rasure and C. S. Williams. An integrated data flow language and software development environment. *Journal of Visual Languages and Computing*, 2, 1991.

[17] J. K. Udupa. *3DViewnix.* Medical Image Processing Group, Department of Radiology, University of Pennsylvania, 418 Service Drive, Blockley Hall, Philadelphia, PA 19104, 1.0 edition, 1993.

[18] T. Zuk. The registration of multimodality medical scans. Master's thesis, The University of British Columbia, Department of Computer Science, October 1993.

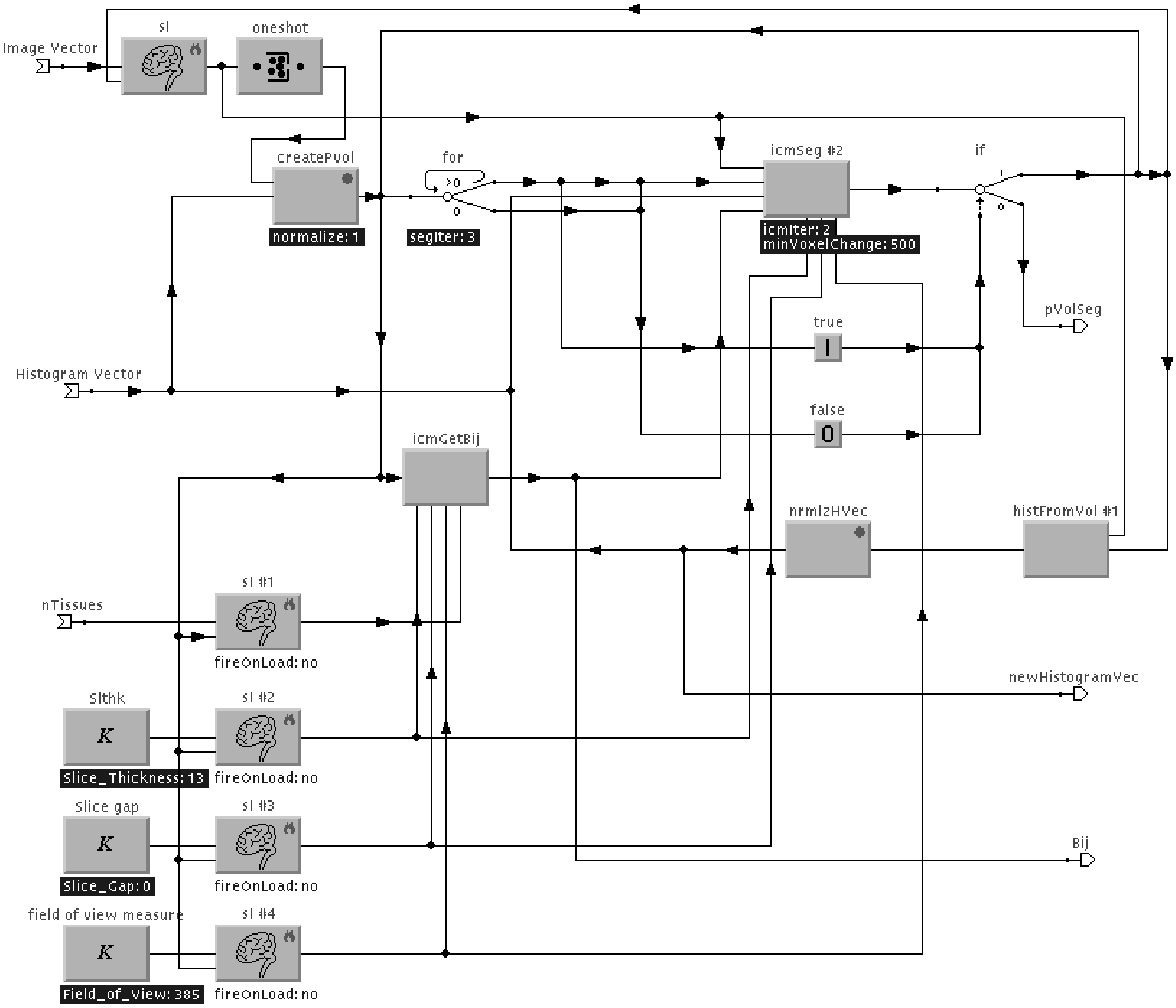

Figure 7: Hierarchical Operator *Segment Spectrum*