

University of Montana

ScholarWorks at University of Montana

Graduate Student Theses, Dissertations, &
Professional Papers

Graduate School

1994

Animation of a process for identifying and merging raster polygon areas

Katherine J. Kahl
The University of Montana

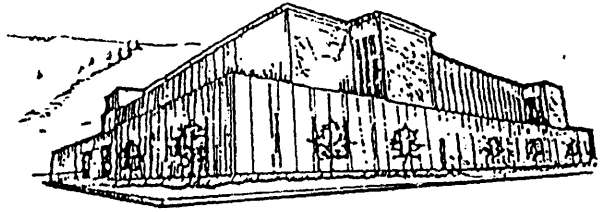
Follow this and additional works at: <https://scholarworks.umt.edu/etd>

Let us know how access to this document benefits you.

Recommended Citation

Kahl, Katherine J., "Animation of a process for identifying and merging raster polygon areas" (1994).
Graduate Student Theses, Dissertations, & Professional Papers. 5109.
<https://scholarworks.umt.edu/etd/5109>

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact scholarworks@mso.umt.edu.



Maureen and Mike MANSFIELD LIBRARY

The University of
Montana

Permission is granted by the author to reproduce this material in its entirety, provided that this material is used for scholarly purposes and is properly cited in published works and reports.

*** Please check "Yes" or "No" and provide signature***

Yes, I grant permission
No, I do not grant permission

Author's Signature Katherine S. Kahl

Date: 6-14-94

Any copying for commercial purposes or financial gain may be undertaken only with the author's explicit consent.

Animation of a Process for Identifying and Merging
Raster Polygon Areas

by

Katherine J. Kahl

B.A. The University of Michigan, 1975

presented in partial fulfillment of the requirements

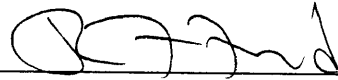
for the degree of

Master of Science

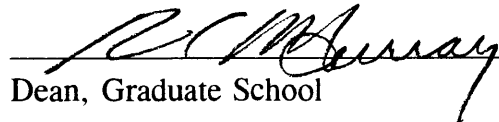
The University of Montana

1994

Approved by



Chairperson, Project Committee



Dean, Graduate School

July 14, 1994

Date

UMI Number: EP40573

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI EP40573

Published by ProQuest LLC (2014). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against unauthorized copying under Title 17, United States Code



ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

J.F. 8-5-94

Kahl, Katherine J., M.S., June 1994

Computer Science

Animation of a Process for Identifying and Merging Raster Polygon Areas
(93 pp.)

Director: Ray Ford

Algorithm animation is designed to reveal the logic, progress, and internal workings of an executing program. The implementation of an effective visualization of an intricate algorithm requires a specialized environment for both programmers and end-users. The potential for discovery of previously undiscovered properties of the algorithm is great, but only if the environment is sufficiently easy and enjoyable to use. This project investigates the requirements for algorithm animation by using a scientific data visualization package to produce animated prototypes of an image classification algorithm.

The *MERGE* algorithm is currently under development to satisfy variable application constraints defined by biologists at the Montana Biodiversity Project. After a satellite image is processed to classify areas of similar vegetation characteristics, *MERGE* analyzes the 2-dimensional data array and identifies raster polygons characterized by similar data values. The algorithm merges and eliminates insignificant areas by applying a set of user-defined rules and criteria based on properties of neighboring cells, effectively operating as a noise elimination function.

The ultimate long-term goal of the *MERGE* algorithm animation project is to be able to demonstrate the algorithm in real time, in animated graphical form, with a specific set of user controls over the algorithm's progress. The goal at the current project level is to propose and investigate a set of alternatives based on related work and background. Detailed intermediate goals and successive stages of implementation are anticipated and described by the spiral model of software development. Prototype animations are implemented using the IBM Data Explorer software on an RS/6000 platform with user modules written in the *C* programming language. The final results of this project include use of the spiral model to document that capabilities for effective algorithm animation are limited by properties of the current visualization system.

Table of Contents

ABSTRACT	ii
Table of Contents	iii
List of Figures	v
List of Tables	vi
Acknowledgements	vii
1 Project Formulation	1
1.1 Scenario	1
1.2 Application Background and Terminology	3
1.3 The MERGE Algorithm	5
1.4 The Spiral Model of Software Development	6
2 Related Work and Background	9
2.1 Scientific Visualization	9
2.2 Visual Programming	11
2.3 Algorithm Animation	13
2.4 Computer Animation	22
2.5 The Data Explorer Software	23
3 Implementation	28
3.1 Introduction	28
3.2 Analysis	28
3.3 Project Details	30
3.3.1 Spiral 1	34
3.3.2 Spiral 2	49
3.3.3 Spiral 3	62
3.3.4 Future Spirals	72
4 Conclusion	76
4.1 What was accomplished	76
4.2 Future Enhancements	78
4.3 Lessons Learned	78
4.4 Summary	79
Bibliography	81
Appendix A: DX Data Model Class Diagram	83

Appendix B: User-written <i>DX</i> Modules	84
Appendix C: Hardcopy Output of Animation Sequences	90
Appendix D: <i>DX</i> Network Descriptions and Instructions	93

List of Figures

Figure 1.1: Generic Spiral Model of Software Development	8
Figure 1.2: Attributes of Algorithm Animation [Brown-88]	16
Figure 3.1: <i>MERGE</i> Algorithm Animation Main Spiral	33
Figure 3.2: Spiral 1.0 Feasibility Subtasks	35
Figure 3.3: Static Visualization of <i>MERGE</i> Data	43
Figure 3.4: Series of Changed Cells	47
Figure 3.5: Spiral 2.0 Development Subtasks	50
Figure 3.6: Sample Network of Caching Modules	57
Figure 3.7: Similarity Matrix	60
Figure 3.8: Spiral 3.0 Interactivity Subtasks	62
Figure 3.9: Interprocess Communication	66
Figure 3.10: <i>MERGE</i> Dataflow Network	67
Figure 3.11: Performance Timings	69

List of Tables

Table 3.1. Spiral 1.1	36
Table 3.2. Spiral 1.2	39
Table 3.3. Spiral 1.3	41
Table 3.4. Spiral 1.4	44
Table 3.5. Spiral 2.1	51
Table 3.6. Spiral 2.2	54
Table 3.7. Spiral 2.3	56
Table 3.8. Spiral 2.4	59
Table 3.9. Spiral 3.1	63
Table 3.10. Spiral 3.2	65
Table 3.11. Spiral 3.3	68
Table 3.12. Spiral 4.0	72
Table 3.13. Spiral 5.0	74

Acknowledgements

I gratefully acknowledge the wise and patient support of Dr. Ray Ford and Dr. Alden Wright of the Computer Science Department, and committee members Dr. Jim Ullrich and Dr. Roly Redmond. I'd also like to thank Mike Sweet, Dick Thompson, Ron Righter, and Saxon Holbrook.

This effort is dedicated to my family.

1 Project Formulation

This project addresses a problem in the application area of geoprocessing which combines the fields of image processing and geographic information systems (*GIS*). The application is a computationally intensive model of a classification algorithm applied to a raster *GIS* data set. The area of interest is the detection of boundary conditions in 2-D raster images and the potential for merging regions of relatively uniform properties. The specific task of this project paper is to address the following problem: we would like to better understand this large, computationally intense, analytical process. Algorithms are the "stuff" of computer science, the specific problem-solving methods. The approach we've taken is *algorithm animation*, which involves knowledge and concepts from the fields of analysis of algorithms, the human visual process, scientific visualization, visual programming and programming environments, animation as an art and as an applied computer graphics technique. A review of the literature on algorithm animation and related topics is covered in Chapter 2, followed by a description of the specific visualization software system. Implementation details are the focus of Chapter 3, and Chapter 4 summarizes the results.

This chapter gives a brief introduction to the different subjects that are necessary to understand the context and significance of this research. The first section defines some image processing terminology and the second section describes the *MERGE* algorithm. The last section outlines the chosen software methodology which determines the entire approach to the project.

1.1 Scenario

Picture a scientist at the console of a graphics workstation. The monitor displays two images, where one image, the "raw" input image, never changes.

The other, a "derived" image resulting from the application of a rule-driven image enhancement process, changes constantly as a sequence of image transforming rules are applied. The scientist is able to observe the sequence of changes and interact with the image enhancement process in a number of ways. At any point he/she can adjust the system controls to speed up, slow down, or replay the sequence of intermediate images. He/she can directly modify the image enhancement program by graphical manipulation of the visual program representation. Finally, the scientist is able to halt the derivation process in a particular state and demonstrate a new rule by simply "painting" a new image illustrating the desired result over the current image. In this mode, the underlying system is directed to deduce the rule that appropriately describes the transformation.

Once a mere "vision" shared by a few ambitious designers, interactive computational discovery is now a reality, achievable in the laboratory. This is specifically due to conceptual and technical breakthroughs in *scientific data visualization*, *visual programming* languages and environments, and *algorithm animation*, which is the visualization of program execution. These areas have advanced rapidly over the past few years to suggest the convergence of an emerging set of paradigms for programming and design that define the conceptual framework and enable discoveries at the intersection of these disciplines by providing us with whole new patterns and modes of thinking [Ambler-92].

This project paper is a contribution to the intersection of these technologies. The application domain is an image enhancement process that is critical in the domain of ecosystem analysis. The goal is to combine the best aspects of visualization, visual programming, and algorithm animation to build an interactive user interface. Appropriate principles of data visualization are applied in order to depict the exact structure and observable properties of the data. The goal is to improve user understanding of the

underlying processes of the algorithm. Our general approach is based on the spiral model of software development [Boehm-88]. Our visual programming environment is the IBM Visualization Data Explorer software on an IBM RS/6000 platform [IBM-93]. For extensibility, the system is based on this general purpose scientific visualization package with a visual programming interface, rather than a customized but obscure collection of graphics and window manipulation components. The results at the end of this project fall short of the concepts described by the scenario regarding complete interactivity, real-time algorithm animation and communication with the algorithm process, and the inferential capabilities. However, assuming that increased functionality is supported by new versions of the visual programming environment, these features can easily be scheduled as iterative enhancements paralleling development of the application algorithm.

1.2 Application Background and Terminology

Image Processing

Image processing techniques include image reduction, magnification, aggregation, subsetting, classification, contrast enhancement, edge detection and enhancement, filtering, etc. There are two main purposes of the methods: 1) to appeal to the senses by improving the visual appearance of the image to the user; and 2) to prepare images for automatic analysis, such as the measuring and detection of inherent features.

A satellite image is similar to a photograph of the earth, but is made up of overlapping bands of data where each band represents a range of spectral reflectance values of the features. Satellite data is a primary source of digital raster images. A digital raster image is a two-dimensional grid of pixels. Each pixel represents a certain size area on the earth and can be denoted by a unique (*i*th row, *j*th column) ordered pair. The size of the grid is determined by the spatial resolution of the data. *Image subsetting* is used to extract target portions of an image. *Image aggregation* is used to transform an image to

a higher level of abstraction by collecting groups of pixels into areas called *raster polygons* based on both their spatial and attribute-dependent relationships. Raw or unclassified satellite images yield the spectral reflectance values of the features on the ground which is not enough to identify the features themselves. *Spectral classification* algorithms contained in some of the commercially-available packages, such as *ERDAS*, process the data to produce classified images which identify features.

Classification

The example of image classification applied here is the identification of classes of existing vegetation and land cover from analysis of satellite imagery. Each cell in a classified image contains one of a set of class values. In essence, the class value identifies the class to which that cell belongs. There are two methods of assigning spectral classes. In a supervised classification, the user defines the classes by identifying areas of interest among the spectral statistics and parameters. Unsupervised classification is where the user allows the classification program to identify significantly different areas by statistical data clustering. Classification details are dependent on characteristics of the remote sensing equipment used, properties of the dataset, and the intended application.

Aggregation

In general, aggregation is a process that combines pixels into polygon areas determined by size and shared properties determined after the initial classification. The *MERGE* algorithm implementation is based on the premise that aggregation details should be under user control in order to address application-specific (e.g. *expert system*) constraints. These details include information such as the following.

- The user specifies minimum area size to merge to (*threshold* value) and any class values where areas should be excluded from aggregation.
- Only areas smaller than the threshold size are to be merged.
- Merge areas in increasing order of size and don't reorder to-be-merged areas when their size changes, unless their new size exceeds the threshold value.

- Merge areas according to a user supplied selection function which indicates the relative similarity of attributes between data values of neighboring areas.

The inherent complexity of the aggregation and the large size of target images leads to any algorithm which implements aggregation being both memory and compute bound. Thus the technical considerations of the tradeoff between memory and the processing time involved in the implementation of an effective algorithm become significant issues. The goal of the *MERGE* algorithm animation project is to clarify some of these issues by illuminating the rule application process.

1.3 The MERGE Algorithm

The *MERGE* program was written as an implementation of a general aggregation process by Prof. Ray Ford of the Computer Science Department. *MERGE* takes input in the form of any remotely sensed image data which has been preprocessed from a 3-D array of raw pixel data into a classified image. The classified image is a 2-D grid of cells, where each cell has been assigned a discrete value based on analysis of the spectral bands from the remote sensing instrumentation. The problem is to "refine" the image by combining adjacent cells with similar class values into regions. For both visual and measurement analysis, regions below a certain *threshold* size are considered "noise", and must be eliminated by combining them with their larger neighbors according to certain ordering, selection, and similarity criteria. After the "uncluttering" process of *MERGE*, the image will only contain regions larger than the threshold size. Thus *MERGE* operates on the data structure as a noise elimination function, efficiently combining adjacent polygons according to user-specified parameters and rules.

The inputs to *MERGE* are a file containing integer values for an $n \times n$ grid, a threshold size, and in cases incorporating a neighbor selection function, an array defining similarity

values within the range of valid cell values. Additionally, the user can specify particular class values to be excluded from the merging process. The outputs of *MERGE* are the updated classified image file that contains only regions larger than the threshold size, and a statistical summary of the results and internal performance.

1.4 The Spiral Model of Software Development

Implementation of the algorithm animation project is prescribed by the spiral model of software development [Boehm-88]. The spiral model is one candidate that addresses a weakness of traditional software process models: that they tend to discourage the effective approaches of prototyping and software reuse. The main characteristic and strength of process models is that they answer the following software project questions.

- *What process should be done next?*
- *How long should the process continue?*

The spiral model is an iterative risk-driven approach to software development. It is graphically illustrated by an outwardly spiralling curve that reflects the concept that each cycle of redesign addresses the same processes as in prior cycles, but with more detail and in the context of new information. The radial distance of the curve from the center reflects the cumulative "cost" to date; the angular dimension measures the relative progress toward completing each cycle. Use of the model removes the negative connotation that the reworking of a product of an earlier stage has under a document-driven or code-driven development model. That is, the spiral model suggests that such reworking is natural and inevitable, rather than the result of errors or failure in earlier stages of development.

A typical spiral cycle involves four phases, represented in two dimensions by the four

quadrants defined by the intersecting x and y axes. A new cycle starts with the hypothesis that there is a software project deserving implementation. The spiral is terminated at any time if the hypothesis fails.

The first phase involves identification of the objectives, alternatives, and constraints of the proposed project. The second phase is evaluation of the alternatives with respect to the objectives and constraints in order to identify areas of uncertainty and the probabilities and costs of potential failures. A required output of the second phase is the formulation of a plan to resolve the causes of these risks. The plan may involve some form of prototyping, specific test plan, user questionnaires, or other appropriate action to target the risk factors. Depending on the relative nature of the unresolved risks after phase two, the third phase may be implementation or another round of design. Each cycle of the spiral culminates in the fourth quadrant with a review involving the people and organizations involved in the product. The main purpose of the review is to assure that all concerned parties are committed to the approach for the next round.

The spiral model is extremely flexible, permitting overlapping curves for concurrent tasks, three dimensional spirals, etc. A generic spiral for the software development process from [Boehm-88] is illustrated in Figure 1.1.

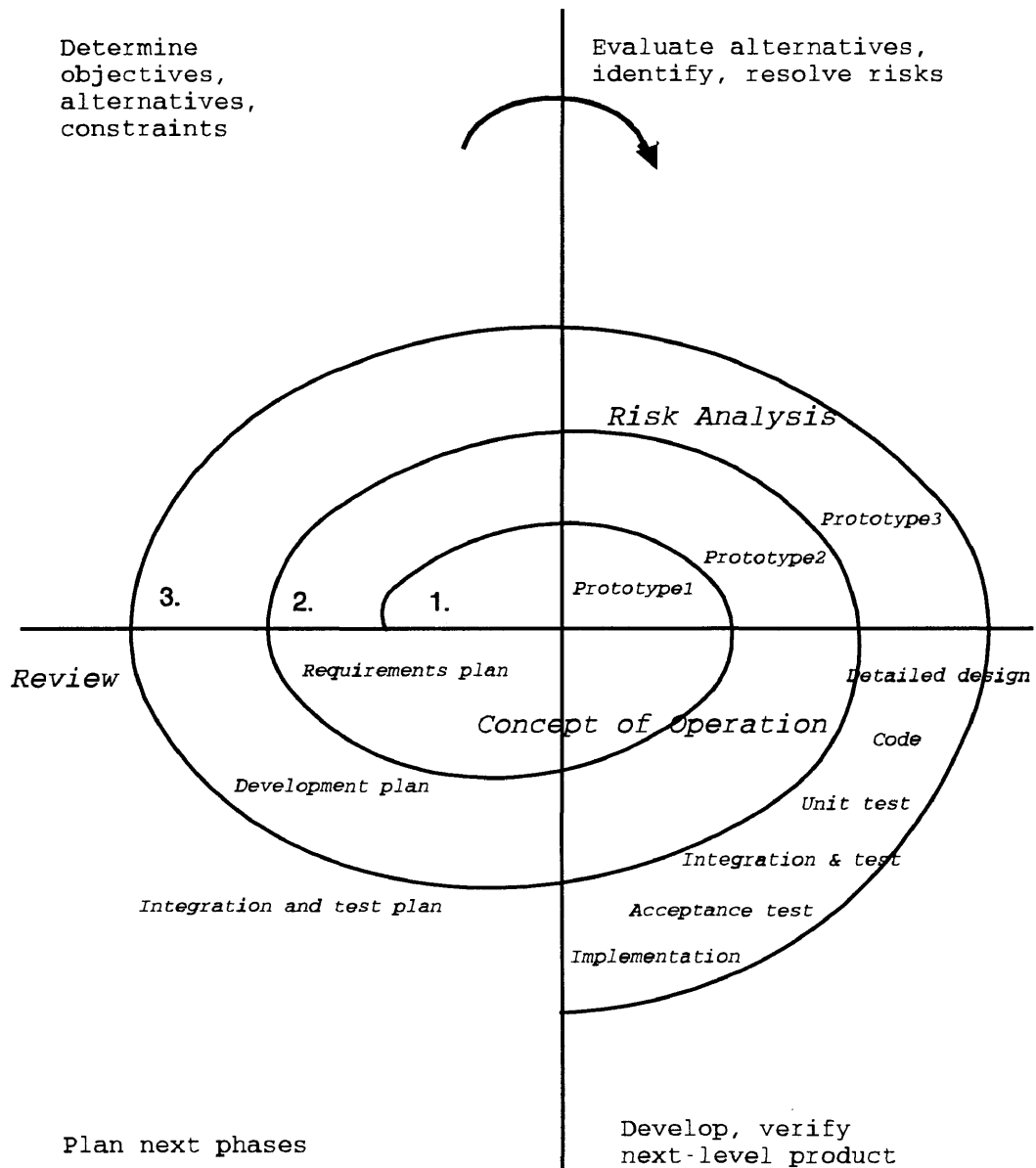


Figure 1.1: Generic Spiral Model of Software Development

2 Related Work and Background

Scientific visualization, visual programming, and algorithm animation share the premise that critical processes of intellectual discovery can be enhanced through the use of visual depiction of appropriate phenomena. New programming paradigms are emerging to structure the visibility of data, program modules, and program execution in these three areas. The goals and paradigms of these three intersecting disciplines are discussed and illustrated by application of the capabilities of IBM's Visual Data Explorer package.

2.1 Scientific Visualization

The desired product of visualization is comprehension, not graphics. Visual images provide the greatest bandwidth to the human brain, with over half the neurons dedicated to visual processing [Earnshaw-92]. The development of written language using letters and numbers as abstract symbols of reality is a much more recent development than the biological evolution of our optic nerve and visual cortex system with its advanced visual processing abilities. Vision gives us direct representations of rates of change, direction, depth cues, light and dark, and surface properties. Researchers who use visual techniques immediately begin to make new discoveries about their data. New details and patterns, or not-yet-discovered errors are more readily apparent in images than from statistically tabulated numbers and textual reports. Vast quantities of data can be summarized pictorially in a coherent image whereas the same data volume would be overwhelming to analyze both in terms of time and organization.

Scientific data visualization provides the scientist with pictures of data. The goal is to promote an understanding of the outputs of some modeling process, in the hope that this understanding will provide the sort of insight required for the author of the modeling

process to enhance it to better match the characteristics of some physical process.

A range of techniques combine to make up scientific visualization: *data representation techniques*, to define data definitions and relationships and facilitate import and export operations between applications, *three-dimensional graphics* using depth cues and perspective to portray spatial relationships, *rendering* using texture, color, and shading to portray multiple values and dimensions, and, increasingly, *animation*, presenting a rapid succession of images to portray the added dimension of time. The advent of data visualization has come with advances in computer storage capacity and processor speed. The entire nature of visualization software is changing. The software problem is no longer just to render a single static image but to create efficient ways to store data with varying characteristics for strategic access to create an effective visualization, particularly one composed of a sequence of images (i.e., a pseudo-animation) to support dynamic modelling.

New data representation techniques have been developed to accept data from any source and automatically translate it to a compatible format for display. Two distinct classes of architecture for general-purpose visualization systems have evolved, and both allow the user to display and manage data easily without having to do any textual programming. The first, exemplified by systems such as Fieldview and The Data Visualizer, provide a fixed set of powerful tools. Pop-up menus and simple icons allow objects to be picked, probed, rotated, scaled, colored, and transformed. This tool-kit approach is limited by the lack of interaction between these tools [Lucas-92]. The second set, which includes *DX* and software such as AVS, apE, and IRIS Explorer, provides a large number of modules which can be connected by a dataflow graph to produce a visualization. Icons for individual modules are provided and applications are built by connecting predefined inputs and outputs. Streams of data flow like liquid through a network of nodes. The scientist controls only the parameterized connections between modules. Evaluation ordering is automatically scheduled by the system based on either the *data-driven* or

demand-driven execution model. In the data-driven model, a node is scheduled for execution as soon as all required input is available. In the demand-driven model, a node is not executed unless its required inputs are all available, and unless its results are directly needed for output by another node whose results are needed.

2.2 Visual Programming

The goal of visual programming is to simplify the programming process. Examples of visual programming language approaches include languages whose syntax is dataflow diagrams, programming-by-demonstration languages in which program logic is demonstrated by manipulating data on the screen, and form-based languages whose syntax incorporates spatial relationships on a form. The effect of visual programming applied to data visualization is to interactively control program flow in order to obtain a more natural expression of the modelling process and its components. A visual programming language can apply more expressive power to the control of data visualization than a textual programming language since the semantics incorporate spatial relationships. Visual programs are not self-documenting, however. The program interface should support the association of additional documentation with the graphical elements.

A visualization system requires support for data import/export operations, data manipulations, and data display. Williams categorizes the state of the art of visual languages for visualization [Williams-92]. Dataflow is the standard programming paradigm, but the entire process is represented by differing data models, execution models, operators, and monitoring facilities. An interactive, dataflow-based visual language has proven to facilitate the rapid prototyping of solutions to complex data analysis and visualization problems. The syntax of a visual language may be thought of as the graphical elements which are the nodes (modules or operators) and their connections. The semantics refers to the definitions provided by the valid constructs in

the language. Several systems allow efficient access to predefined functions through a hierarchical arrangement of operators classified into categories. Sample categories are input/output, transformation, extractors, rendering, and transfer and control, or structuring. Control modules are part of the semantics of the execution model required to solve complex problems using visual languages. They provide run-time decisions to order the execution of operators. Control operators may include for loops, while loops, if-then-else constructs, merge and trigger functions. Data dependent control operators extract information and may set values to change operator inputs to direct the flow of program execution. Facilities for animation in the current set of visual language systems are of two kinds: image-based and geometry-based. Image-based animation involves generating a sequence of images and playing them back to form an animation loop. Geometry-based animation refers to saving geometries and the use of rendering hardware to render them as a sequence.

None of the current visual systems supports recursion. This would require that each instance of an operator have its own local data per execution. Some support graphical hierarchies where a higher level operator can be built as a visual program consisting of lower level operators. The implementation of abstract operators which hide the complexity of the data model from the user rely on polymorphism and overloading to allow the output of any operator to serve as the input to any other operator. This depends on the operators not producing any side effects. Data granularity refers to the size of the data blocks flowing through the system. Fine-grained dataflow operators can manipulate small atomic elements and are necessary for real-time process control. Large-grain dataflow systems are the norm where the block size passed between modules is the size of the data model. Large-grain dataflow systems may not save intermediate results since memory costs are prohibitive. Thus these systems do not save state. Intermediate results must be recomputed to produce new output. Program execution cannot be interrupted nor can partially executed programs be recovered.

Burnett's Taxonomy

Burnett describes a taxonomy of characteristics of visual programming systems that provide increasing levels of power to the scientist to interface with the application in ways that involve much more than simply changing parameter values [Burnett-92]. The lowest level of control is *post-processing*, which implies that data must be generated off-line, prior to the invocation of the visual programming environment. The visual representation of data is available only after the application has completed. There is no opportunity to dynamically change parameters or analyze the results in progress to affect the scientific application. The next level of ability which involves the ability to generate and continually update the visual representation of the data as the application executes is called *tracking*. At this level the scientist's wait loop for feedback is effectively diminished yet the information flow under tracking is still unidirectional. The most powerful systems are those which allow the scientist to provide feedback to the computer during the visualization and computation itself through a two-way flow of information. Two levels of systems with bidirectional information flow are categorized. *Interactive visualization* is a term used to describe tracking with feedback to allow dynamic changes in how the graphical information is to appear. *Steering* is the ability of the scientist to make unanticipated changes to any aspect of the data and logic of the scientific application. It is the ultimate goal of visual programming.

2.3 Algorithm Animation

Algorithm animation focuses on illustrating the process rather than the final results of a complex task, i.e. on *program visualization* as opposed to *data visualization*. The goal is a clear understanding of the structure and form of an implemented program, specifically aimed to edify and enlighten the programmers and algorithm designers. By incorporating the display of program code and system performance documentation, designers gain understanding of the current implementation through a graphical representation of the

imation can
on displays,
n animation
nts in color

resolution, sound, and parallel processing capabilities.

Program Visualization Systems

Program visualization systems can be classified by whether they illustrate code or data, whether the displays are static or dynamic, and if dynamic, either interactive or passive. Static displays of code include flowcharts, scoping, and module interconnection diagrams. They can be automatically generated and animated by highlighting appropriate sections as the code executes. Static displays of program data are more difficult to generate than program structure, since data structures can be implemented in many different ways and have many representations, the most effective of which are only found through experimentation. Ron Baecker's movie *Sorting Out Sorting*, first shown at Siggraph '81, illustrated several sorting techniques applied to large and small data sets and is a classic example of an effective dynamic yet passive program visualization.

Brown's Model

Brown provides a definitive outline of the field of algorithm animation in his dissertation where he developed a model for an algorithm animation system in active collaboration with the teaching of computer science courses dealing with algorithms and data structures [Brown-88]. Animations developed for instruction can be used for research, and vice versa. Whole libraries of useful animations have been developed for well-known algorithms and have led to improved variations. Other applications include the production of technical drawings, performance tuning, program documentation, and systems modelling, especially with respect to multiprocessing applications. Brown mentions one area that has met with limited success. Graphical views of program structure and code, when tied directly to syntax, are limited in their usefulness and do not provide any significant advantage over textual aids in program development environments. He provides a useful history of algorithm animation efforts, ranging from making movies using graphic stills to various models based on annotating code with "interesting events" or using temporal constraints as external triggers. Several approaches used Smalltalk to create an algorithm animation system with almost a trivial amount of effort. Brown points out that the primary drawback of using a general-purpose programming environment for algorithm animation efforts is that end-users are restricted to using only the features found in the environment. Some of these restrictions include not being able to use multiple windows, zoom facilities, script facilities, or playing algorithms backwards.

Brown describes algorithm animation displays using three dimensions, as shown in Figure 1.2 [Brown-88]. First, looking at the content axis, *direct* displays are pictures that could be constructed from the data structures with no additional information needed. *Synthetic* images represent operations causing changes in the data, or abstractions of the data. The *persistence* dimension ranges from displays that show only the current state of information to those that show a complete history of the information. The third aspect of displays is based on the nature of the transitions as either discrete or incremental, characterized by either abrupt changes or smooth transitions. Discrete transformations are

effective on large sets of data, whereas incremental transformations are most effective when running on small sets of data.

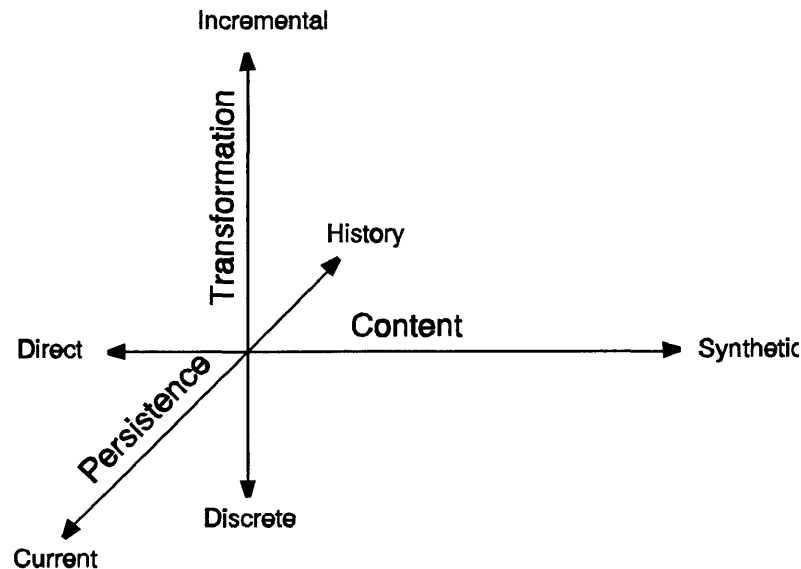


Figure 1.2: Attributes of Algorithm Animation [Brown-88]

The primary contribution of Brown's thesis is the implementation of a dedicated algorithm animation system consistent with his theoretical models. Separate models are defined for 1) programmers creating animations, 2) end-users interacting with the animation, and 3) end-users creating, editing, and replaying dynamic documents, or scripts. Each model is independent of the algorithms, inputs, and views, which makes it easy to animate new algorithms and interact with them consistently. Scripts are semantic interpretations of an end-users session that can be replayed for passive viewing or active exploration. Algorithms are annotated with event stubs to indicate interesting phenomena that drive updated displays. Additionally, these events provide the abstraction for end-users to control the execution in the following ways: 1) algorithm events name segments of code so end-users can refer to them; 2) the end-user can associate a cost with the event; 3) an output event signals the views to update themselves; 4) an input event signals the input generator to provide the algorithm with data. A view defines a synthesized,

dynamic graphical entity. It does not access an algorithm's data structures, but updates the screen image as a result of update messages received from an adapter translating algorithm output events.

Characteristics

Algorithm animation introduces some problems not faced by the classical graphics program designer. Brown provides specific guidelines for effective algorithm animation. Multiple views are recommended rather than a single overly complicated one [Brown-85]. The content of each view should be a picture of a separate process component, such as a tree, queue, or list, with indications of process completion, efficiency, and history. With multiple views on one console, windows are necessarily smaller. Displaying multiple views consistently is a task requiring robust control over timing services, and high-level access to a toolkit for program features of shape, line style, and color mapping. It has been shown that information is lost if the animation is too fast or too slow. The viewer and the purpose of the display determine the best speed. Fast speeds provide high-level intuition; relatively slow speeds are required to understand details or unfamiliar representations of the algorithm. Color is the essential tool for highlighting activity, representing state transitions while tying views together and emphasizing patterns. Recent reports indicate the effectiveness of highlighting subtle changes with sound-producing components. Scientific sonification is a new area where the multiple properties of sound such as tone, volume, and duration are being studied as appropriate mappings to process parameters. Other traditionally useful features include control panels with start/stop buttons, sliders for speed control, customized data menus, and snapshot/restore capabilities.

Algorithm animation involves the high-level paradigms of sequential control, real-time processing, parallel environments, and the simulation of continuous movement, or the appearance of continuous transitions from one state to another, as well as those of graphic representations of standard programming constructs. It also involves principles of spatial data management. Research is providing a better understanding of the role of diagrams,

particularly the role of dynamic diagrams in the role of the software development life cycle. Elements of diagramming, including polygonal and circular objects, connectors of different widths and styles are the basic graphical building blocks, with support for static and dynamic icons, whose representation changes with their attribute values. How they are defined and constrained is critical to algorithm animation. The controlled use of dynamic graphics including highlighting techniques illustrate the need for a more complete dynamic vocabulary to ensure that the dynamic results of algorithm animation enhance clarity of vision.

Roman's Taxonomy

Previous classifications of program visualization systems had focused on degrees of *sophistication*, from pretty printing to complex algorithm animations [Shu-88], static vs dynamic *display styles* of code, data, or algorithms [Myers-90], direct or synthetic *levels of content*, discrete or smooth *transformation capabilities*, *persistence*, with regard to the current state or entire execution history [Brown-88], and finally categories of *aspect*, *abstraction*, *animation*, and *automation* [Stasko/Patterson-92].

The taxonomy of Roman, *et al*, is based on a well-defined model of the field [Roman-93]. "Program visualization is a mapping, or transformation, of a program to a graphical representation." The model is a function of the interactions of three variables, or participants: the programmer who is responsible for developing the original program, the animator who defines and constructs the mapping, and the viewer who observes and reacts to the graphical representation. Although these theoretical roles may not be filled by three distinct real-world personages, the specialized techniques of each stage represent a natural division of identified functions. Based on this mapping, program visualization systems are classified according to five criteria: *scope*, *abstraction level*, *specification method*, *interface*, and *presentation*.

1) **Scope** *What aspect of the program is demonstrated?*
 A program is characterized by its code, data, algorithm, or control state and execution behavior.

2) **Abstraction** *What kind of information is conveyed?*
 This criterion divides the animation into levels of concepts presented much as Shu and Brown do with levels of sophistication and content, transformation, and persistence. Highlighted lines of code would be at the very low level of abstraction, while a module interconnection diagram with active modules highlighted would be a structural representation at a higher level of abstraction.

3) **Specification** *What mechanisms does the animator use to construct the visualization?*
 How flexible is the mapping of program states to graphic events - does code have to be written or modified, or can the mapping be dynamically specified?

4) **Interface** *What facilities does the system provide for the visual representation of information?*
 The interface is how the viewer sees and controls the outputs of the system.

5) **Presentation** *How does the system convey information?*
 What means does the animator use to convey meaning, rather than just visual events?

The Development of Program Visualization Systems

The evolution of algorithm animation and program visualization systems can be summarized by the capabilities of several successful implementations of systems specifically for program visualization: *Balsa*, *Zeus*, *Tango*, and *Pavane*. *Balsa* [Brown-88] represents the first system that attempt to organize the task of constructing complex program visualizations. It implements program annotation by adding procedure calls to a program to capture system event changes and update the associated graphical views. *Zeus* [Brown-91] is directly descended from *Balsa* and inherits many of *Balsa*'s capabilities plus some innovative features such as support for direct mappings of values to graphical objects. The next generation of systems attempted to minimize the effort

required to construct graphic objects and define display events. *Tango* [Stasko-90] incorporates the encapsulation features of object-oriented programming in the implementation of abstract data types for animation such as location, images, paths, and transitions, and demonstrates a significant effort at specialized modelling components for program visualization. Control interaction is provided with facilities to identify the derivation of program data from image attributes. Depending on state variables, the same program event may be mapped to different display events at different times. *Pavane* [Roman-92] represents the latest implementation and involves a radical design shift in order to demonstrate the properties of concurrent programs. In the domain of concurrent events, it is difficult to portray a program in terms of event sequences. Code annotation, for example, is impractical because events are not associated with a specific location in the program code. In *Pavane*, the mapping of program states to images is specified as a set of rules to apply with each state change. Objects are also allowed to have time-dependent attributes. These combination of features enable the abstract visualization of concurrent programs with relatively few rules, display code, or program annotations. Table 2.1 summarizes the taxonomic criteria outlined in [Roman-92] and includes a description of the level of support of the features using IBM's Data Explorer.

Taxonomic Criteria	Examples	DX Capabilities
Scope (what is shown)		
Code visualization	highlighted statements	not directly, w/ debugger?
Data state	graphical representation of data	yes
Control State	highlighted current module	user-written modules
Behavior	event trace of sequence of states	program w/ sequencer
Abstraction Level (level of sophistication or context)		
Direct representation	gauges to display values, colormaps	colormaps, grids, interactors
Structural representation	properly-sized blocks	programmable
Synthesized representation	3D representation of shortest path	"
Specification method (level of automation)		
Predefinition	application specific	programmable
Annotation	program calls animation routines	programmable
Declaration	direct mappings graphical objects	no support
Manipulation	mapping of gestures to program events	no support
Interface (graphical vocabulary)		
Simple objects	points, lines, glyphs, text	yes
Composite objects	predefined collections of objects	groups and fields
Visual events	discrete or smooth animation	sequencer, discrete + morphing?
World (dimensionality)	multiple dimensions absolute vs constraint-based	3+ absolute
Multiple worlds	multiple windows	multiple images
Control interaction	predefined controls	control panels, no nesting
Image interaction†	graphical input	pick, probe
Presentation (semantics of presentation)		
Analytical	program correctness, not mechanics	no support
Explanatory	visual events to convey info	no support
Orchestration	collection of selected visualizations	limited control mechanisms

Table 2.1: Program Visualization Taxonomy Applied to *DX* (adapted from [Roman-93])

2.4 Computer Animation

Sophisticated computer animation techniques have been developed for special effects in the communications industry. The best animation techniques that incorporate interaction are still evolving. Simone provides a breakdown of the features significant to three areas of products [Simone-92].

Business presentations

- speed and ease of generation
- playback ability in a variety of environments
- high quality fonts
- pause, stop, branch ability

Creative animation

- tracing, in-betweening (morphing), looping, keyframing
- defining paths and actors, via splining or interactive editing
- parameterized scripting
- scene description, object, light source, camera parameters
- canned motion and transition effects
- onion skin techniques
- palette optimization (color encoding and enhancement techniques)

Multimedia control

- access various audio and video inputs (MIDI, wavelength audio, CD-based audio, digital and analog video)
- output to video
- video editing capabilities

The evolution of animation for scientific visualization purposes adds a fourth product to this list and involves many of the features. The most significant element is the mathematical model and object definition with appropriate dataflow transformations. Lucas describes four characteristics of scientific data rendering that add to the requirements for algorithm animation [Lucas-92].

Scientific Data and Algorithm Animation

- mathematical and hierarchical data definitions
- variety of data representations
- large quantities of data
- depiction of real-world data require volumetric rendering (3D)

Large quantities of data (even gigabytes with multiple variables and time frames) are the norm, so animation systems for scientific applications need to take advantage of parallelism. The rendering of volumes, i.e. in 3-D, requires a few useful capabilities such as the ability to render translucency for both volumes and surfaces and clip both in render time and object space. This category does require a traditional rendering repertoire with the goal of accurate and appropriate data representation, yet not to the extent of photo-realism or illusionary effects included in commercial packages. Finally, a wide variety of data representations and the ability to correlate modelling methods is necessary for the support of mathematical and hierarchical data representations.

2.5 The Data Explorer Software

The target platform for project implementation is IBM's *Visualization Data Explorer (DX)*. *DX* has several powerful elements that distinguish it from the others in the second set:

- a powerful set of interoperable visualization modules;
- consistent use of an object-oriented data model;
- various levels of system tools; and
- a system execution model that facilitates parallel execution and performance optimizations such as caching.

Modules

DX consists of a large set of highly interoperable visualization modules. Interoperability is the property that modules may be connected in a variety of ways to achieve different effects. Module design and selection is guided by two design goals: that they be sufficiently primitive yet as powerful as possible to aid the model building process. All modules are available in the visual programming interface, where the modelling process is specified entirely by selecting modules from category menus. Each module is represented by an icon that can be dragged and dropped into position in the visual programming environment window (*VPE*). The visual program editor allows predefined

input and output tabs to be preset, connected, or disabled. The object-oriented data model provides strong type checking and error detection at each connection. Execution is controlled by control panel access to *interactors* for setting values of variables, *switches* to control execution flow, and a *sequencer* for controlling a series of objects with different time-dependent attributes. Modules are included for importing and exporting data, transformations such as mappings of one field to another, "rubbersheeting" a surface, and computing slabs, slices, and streamlines. Annotation modules enable captions, axes, glyphs, tubes, and debugging support. Rendering is supported by modules that perform 3-D viewing transformation using properties of camera settings, surface characteristics, color transformations, and lighting models.

Data Model

The *DX* data model is based on the object-oriented paradigm where data is viewed as a hierarchy of abstract data types with support for inheritance, encapsulation, information-hiding, and message-passing. All data that is passed between modules in the system is passed in the form of pointers to objects. Thus objects in this model are the arguments of messages passed between modules, rather than being themselves active entities that create and receive messages.

The data model defines a field object as a mapping from one space to another space. Each field is defined as a set of positions and connections, or interpolation elements. The actual mapping is the set of data values which are in a one-to-one correspondence with either the positions component, implying an interpolation function between values, or the connections component, in which case each element of the implied grid has a constant value. Data fields can be defined on spaces of any dimensionality and connected by primitives of multiple dimensionality. This allows a wide variety of data definitions, from objects defining completely unstructured data components to volumes and surfaces with a variety of regular and irregular structures. A field may contain additional components such as colors, opacities, normals, and statistics, and may be aggregated into groups combined with other fields. The hierarchy of these relationships is represented in the

class diagram of the *DX* data model in Appendix A using Booch's notation to illustrate the data hierarchies [Booch-91].

An object in this system is implemented as an object header in global memory that points to an array of data. The header contains information such as type-dependent attributes and a reference count. Access to objects is controlled entirely by a set of information-hiding access routines. To actually operate on the data, a module calls a procedure to get a pointer to the data. Considerable performance advantages of both time and space are derived from the scheme of passing pointers rather than the "stream-oriented" method used by other visualization systems where data is copied from module input to module output without sharing of unchanged data. Data sharing is enabled and encouraged by enforcing a rule of the dataflow paradigm that a module not modify its inputs. Module output must be recreatable and a strict function of the inputs. Consequently the dataflow network must be implemented as a graph with no cycles. *DX* allows data sharing to be overridden with explicit control over caching the effects of each module. There is, however, no facility built in at the module level for accumulating the results of repetitive transformations.

System Tools

DX system tools provide uniform support for the data model. A sophisticated graphical user interface provides visual programming capabilities with immediate type-checking for consistency as soon as connections are made between module inputs and outputs. Each graphical dataflow program, or *network*, is maintained as a script program available in text form for manipulation as a program or macro. The Application Programmers' Interface (*API*) provides the ability to customize the system with user-written modules in the programming language *C*. The *API* documents the set of access routines which are procedures and functions that take pointers to objects as parameters. Each procedure can pass and receive a variety of object types, implementing polymorphism, and is responsible for type-checking its arguments and processing them recursively. The *application builder*

program and *module description file* simplify the module building process by providing a stub program that includes extensive error checking. The *data browser* and *data prompter* are separate programs that are provided as support tools for the data definition and import process. The import process effectively instantiates the user data as instances of the *DX* data classes. Support is provided for dynamically generated data through Unix pipes or externally linked modules. Thus, the data source is not limited to static arrays generated offline by a different process.

Execution Model

IBM's Data Explorer takes a system-wide approach to supporting parallelism. It is composed of two major subsystems, the user interface and the executive. These two components execute as separate processes that communicate via a socket interface. They may be run on the same or different machines, such as the user's workstation or a departmental server. The user interface consists of the visual program interface, which translates the visual program into a script which is sent to the executive for interpretation. The user controls the executive by the execution pull-down menu options, which include control for starting and stopping execution, control for executing on change, and sequencing both forward and backward through a series of objects either continuously or in step mode. The executive executes the dataflow program by calling visualization modules as needed, passing pointers to objects, caching results, and displaying the resulting images on the user's workstation. Overhead is minimized by combining the modules into a single binary (the executive) rather than dispatching each module as a separate process. This somewhat complicates the process of adding new modules to the system with the necessity of having to relink the executive, but the option is available to specify user-written modules as external *outboard* modules to be dynamically loaded and linked by different dataflow networks as needed.

DX is designed from the start to support multiple processors with high-speed access to global memory at the level of coarse-grained parallelism. Pre-processing support is

provided to designate execution groups of modules to run on separate processors and to define explicit partitioning of data sets. This two-fold approach was taken for the following reasons: most datasets are naturally partitioned; most module sets parallelize rather easily; fine-grained (such as loop-level) parallelism involves expensive inter-process communication; and, inter-module parallelism does not achieve full processor utilization.

The Dataflow Paradigm

By facilitating software reuse, module flexibility, and extensibility, successful dataflow visualization systems are enabling research to progress at a much faster rate than traditional programming environments. The accessibility of the modular system components enables *DX* to address the needs of various levels of users. The novice user can load previously configured dataflow networks to access data visually. Intermediate users are able to arrange modules in different ways to produce a variety of visualizations. Advanced users can challenge system potential in the area of customization, not only by assembling modules into reusable popular arrangements, but by building new system components. High-end users expect consistent design capabilities and performance from any combination of system features, relying on the system to perform well with respect to several issues of scaling. Not only is high performance expected in the area of data management with support for managing hundreds or thousands of modules in a dataflow network, but real-world applications require support for parallelism and explicit control over the execution model at the module level. Embedded applications require control flow mechanisms equivalent to *if*, *while*, and *for* constructs, subroutining, complex event-handling, and other synchronization primitives. A user, for example, may want to redirect an input event to a module higher up in the control flow and reset execution control dynamically. Such capabilities are foreseen but not yet available in current dataflow execution models such as *DX*. Shortcomings in the area of control flow have a significant impact in projects such as this one involving algorithm animation which require facilities for flexible user interaction and inter-process communication.

3 Implementation

3.1 Introduction

The various background areas for this project are scientific visualization, visual programming, algorithm animation, as well as program visualization systems. Each of these areas of research is very new, and as such, has been defined and categorized in the literature only recently. Each new development effort spanning any or all of these areas may match some of the goals and fit some of the categories, or may challenge and refine these definitions. This chapter focuses on using the ideas reviewed in Chapters 1 and 2 to develop a visualization tool for better understanding of the *MERGE* program. The chapter starts with a general discussion of the nature of the problem. The chapter then provides a statement of project goals and an outline of the projected software development schedule. The tasks of the project are formulated as individual spiral cycles, each of which is detailed using the standard spiral template [Boehm-88]. The template lists the goals, risks, and resolution of each spiral in tabular form. Following each template is a narrative of the goals, discussion, and evaluation of the subtask.

3.2 Analysis

The original domain of this project is labelled *algorithm animation*. A standard customizable software package is being developed based on a complex internal algorithm designed to efficiently "match" or "implement" the detailed specifications of a real-world process. The demands of meeting the specifications as they become increasingly detailed require more concrete feedback as to the performance of the algorithm in specific cases. The successful implementation of the process is highly dependent upon the flexibility and appropriateness of the algorithm in handling each individual case. The specific nature of

the individual cases is how to merge each-less-than-minimally-sized polygon area according to both global rules regarding the type of scene analysis desired and individual rules regarding size, position, and value of neighboring areas. The large number of individual cases are derived from the spatial properties of both the target polygon to be merged and each of the polygon neighbors. The combinatorial effect of rule applications and neighbor properties make it impossible to efficiently document each individual case. This provides the motivation for presenting a visualization to view the solution space at several levels of detail.

The nature of the objects and properties being modelled lend themselves to a direct visual representation. Polygons and individual cells have two-dimensional spatial properties of area, size, and proximity. The "target" or "source" data value and neighbor area data values serve as inputs to a selection function which selects the "best" neighbor in which to merge a small "target" area.

"The least abstract type of graphical representations map some aspect of a program directly to a picture ...

The level of abstraction of the graphical representation of programs control complexity during debugging and monitoring and facilitating program understanding in pedagogical settings ...

Direct visualizations are appealing because they can often be constructed mechanically, without knowledge of the programmer's intent. But often a visualization must convey that intent, particularly when sophisticated algorithms are involved" [Roman-88].

A classification of levels of algorithm animation based on abstraction parallels one of the basic concepts of object oriented design and implementation. *Abstraction* and *encapsulation* are complementary concepts: abstraction focuses on the outside view of an object; encapsulation, also known as *information hiding*, prevents users or clients from viewing the inside information, where the behavior of the abstraction is implemented. For abstraction to work, implementations must be encapsulated. The abstraction of an object should precede implementation decisions. "Once an implementation is selected, it should

be treated as a secret of the abstraction and hidden from most clients" [Booch-88]. No part of a complex system should be dependent on the internal details of any other part. While abstraction helps designers and implementors think about what they are doing, encapsulation allows software changes that involve implementation details to be made reliably with minimal effort.

This is the essence of what amounts to a distinct classification of the levels of algorithm animation and program visualization, referred to here as "process illumination". Another descriptive yet unwieldy title might be "specification animation". According to the criteria developed by Roman *et al*, visualizations that involve a low level of *abstraction*, a broad view of *scope*, irregardless of the method of *specification*, level of *interface* or *presentation* categories can provide significant insight to the system designer at the level of specification detail. The viewers of the visualization are not subjected to details of *how* or *why* the programmer chose to implement the specifications, but are directly able to verify the programmer's intent to implement the specifications in terms of accuracy and completeness. By specifically concealing program implementation details, the application process is the focus rather than the details of algorithm selection, appropriateness, or efficiency.

The above discussion previews the final placement of this project in the spectrum of program visualization research. This discovery was evident only after several iterations of project development.

3.3 Project Details

The original project proposal is in the form of the following project statement:

Use *DX* to produce "appropriate" animations of the output of *MERGE*, where "appropriate" is a balance between static visualization of algorithm features, real time linkage with *MERGE*, and user control over input parameters to the *MERGE* program.

The initial concept of the project is based on the four phase approach of the spiral model, restated here as:

- 1) propose a set of capabilities;
- 2) investigate capabilities;
- 3) show what can be done;
- 4) project future directions.

An early breakdown predicted three rounds of the spiral.

Round 1:	Feasibility
Round 2:	Development
Round 3:	Extensions

These three spirals were expanded into subtasks which were developed concurrently as suggested by Boehm. Thus the review stage for this level of task analysis may range from a walk-through of each individual component to a major requirements review encompassing a composition of elements.

"Visualize a series of parallel spiral cycles, one for each component, adding a third dimension to the concept presented in (Figure 3.1). Separate spirals are described for separate software components or increments" [Boehm-88].

The spiral model is both descriptive and prescriptive. A descriptive outline of the elements of each phase serves as documentaton of the software development process. By answering the questions of how long should each process go on, and what process should be pursued next, the model effectively prescribes future activity. The final phase of each subspiral may include recommendations for proceeding with the development of individual components.

The three main spiral processes are outlined with their subtasks which were developed as parallel spirals, as follows:

- Spiral 1.0 **Explore and learn capabilities of *DX***
 - 1.1 Learn *DX* visual program syntax
 - 1.2 Learn about *MERGE* (its critical states) and import *MERGE* data
 - 1.3 Show static visualization of *MERGE* output
 - 1.4 Determine what to show during the animation

- Spiral 2.0 **Develop visualization at the tracking level of visualization**
 - 2.1 Develop method to apply iterative changes
 - 2.2 Develop data inquiry & manipulation (*Pick & Probe*)
 - 2.3 Explore method to retain iterative changes (*Caching*)
 - 2.4 Develop desired interface with user

- Spiral 3.0 **Approach real-time communication and interactivity**
 - 3.1 Link to external modules written in other languages
 - 3.2 Inter-process communication
 - 3.3 Explore performance issues

At this point in the *MERGE* animation project, both the research and prescriptions of the three completed spirals suggest several extensions to the development of the animation. These are included in the appropriate spiral. In addition, two phases of future development extending in completely new directions are partitioned as follows:

- Spiral 4.0 **Approach dynamic feedback capabilities (*Steering*)**

- Spiral 5.0 **Approach inferential capabilities (*Constraint & By-Demonstration*)**

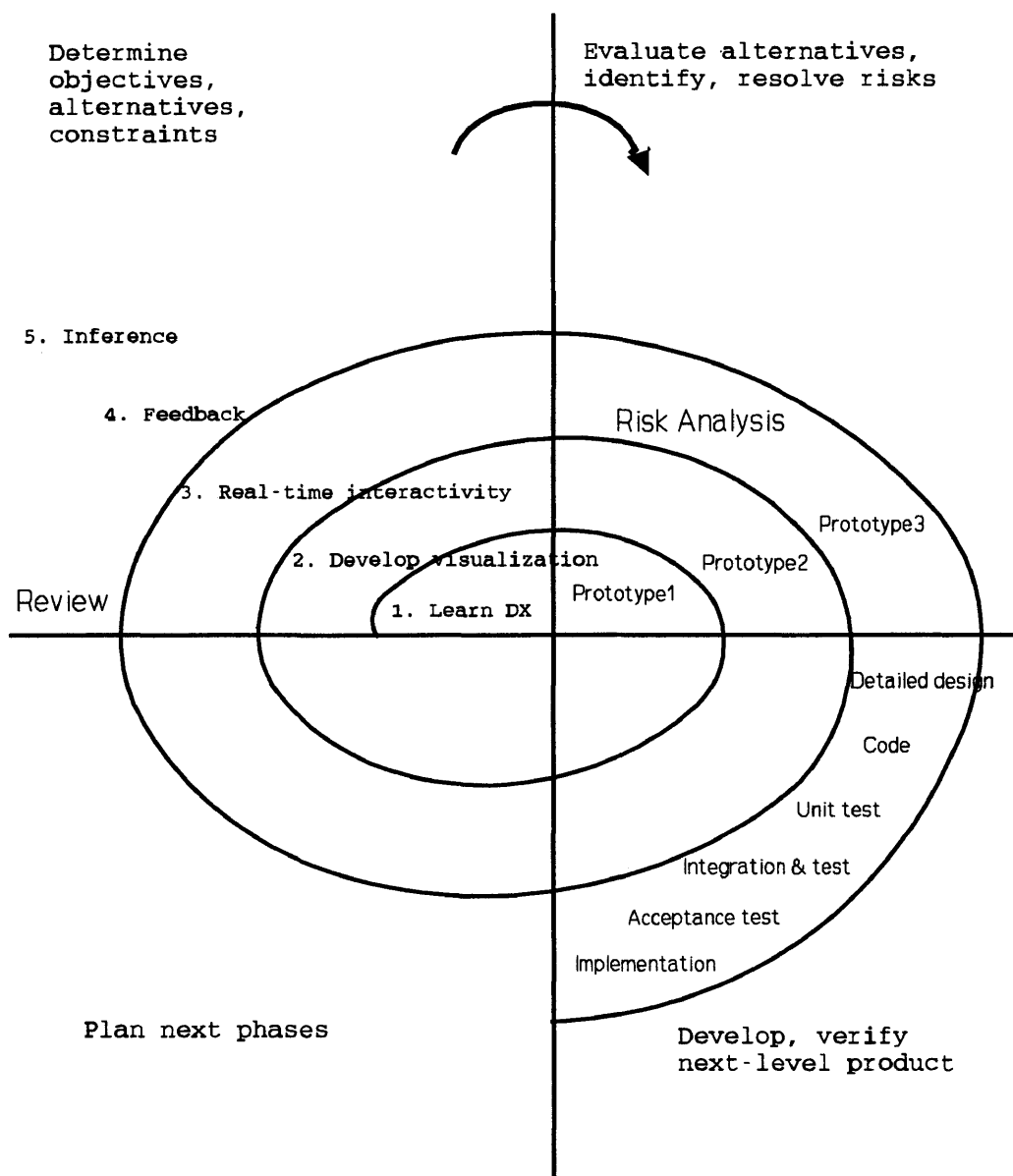


Figure 3.1: MERGE Algorithm Animation Main Spiral

3.3.1 Spiral 1.0: Primary goal: Feasibility

The goal of the first top-level spiral is to explore the feasibility of the project in several different ways. These are broken down into overlapping subtasks, as shown in Figure 3.1. The primary constraint of this main spiral is technology, in reference to the available system software. The first subtask is to learn the Data Explorer visualization system. In general, the learning of a complex system incorporates new paradigms and the requirement is on-going to meet the completion of each specific programming task. Spiral 1.1 continues until the end of the project or until a different software system is chosen. Learning a specific subset of *DX*, the data model and import format, is a requirement of the second subtask, which is to apply the syntax to be able to successfully import the *MERGE* data. This is the goal of Spiral 1.2 which is specifically formulated to meet the requirements of phase 2 of Spiral 1.3. The import process only needed to be understood at this point as far as the representation of static data arrays. Further knowledge from Spirals 1.1 and 1.2 helps guide the formulation of alternatives in Spiral 1.4. Thorough knowledge of the capabilities of the system are required to evaluate the different alternatives and risks of what and how to animate.

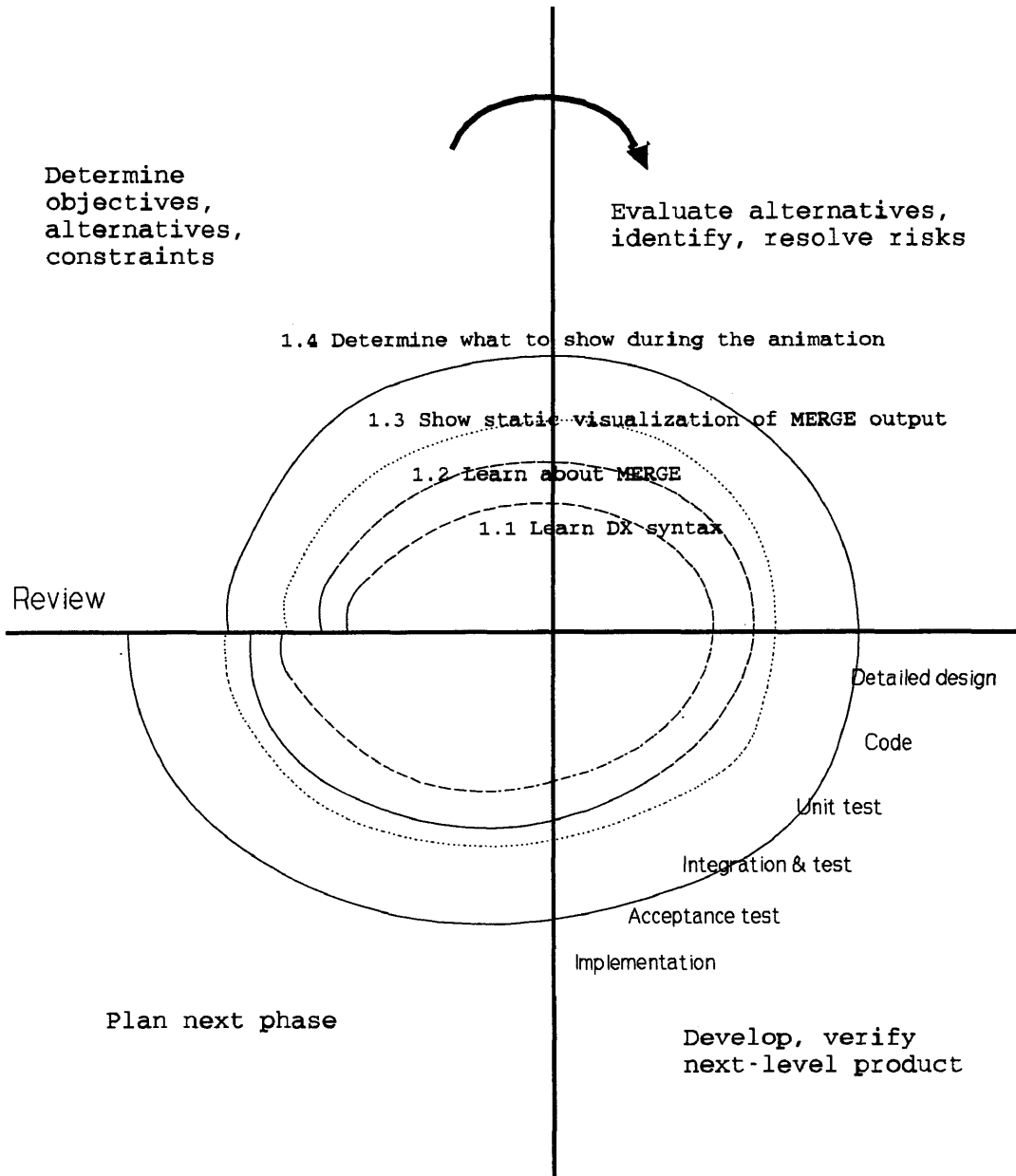


Figure 3.2: Spiral 1.0 Feasibility Subtasks

3.3.1.1 Spiral 1.1: Learn *DX* visual program syntax

Table 3.1. Spiral 1.1

Objectives	Learn <i>DX</i> visual program syntax
Constraints	<p>TIME: spring '94 semester deadline</p> <p>TECHNOLOGY: <i>DX</i> in beta-test mode limited access to manuals manuals often inaccurate <i>MERGE</i> program documentation minimal</p> <p>PERSONNEL: programming staff of 1 with sharp learning curve required</p> <p>FACILITIES: on-campus</p>
Alternatives	<p>"Stay home & bake cookies"</p> <p>Program in Fortran forever</p>
Risks	<i>DX</i> capabilities insufficient, full of bugs, or too complex
Risk resolution	<p>Audit status in CS495: Scientific Visualization w/ <i>DX</i> studied /usr/lpp/dx/sample programs</p> <p>Prototypes: solid geometry model, unix pipe to create import file</p>
Risk resolution results	<p>Beta-test version of <i>DX</i> is full of bugs & holes</p> <p>Release 1.0 1/94 vast improvement so far</p>
Plan for next phase	Focus on import formats aimed for efficiency and flexibility as input to Spiral 1.2
Commitment	On-going

Goal

My goals at this stage were to learn enough of the *DX* system capabilities to use as input in determining the objectives, alternatives, and constraints of subsequent design and programming tasks.

Discussion

I was able to attend an introductory class incorporating visualization with *DX* and learn from the examples given. My first prototype was a solid geometry model which uses the *DX* *faces, loops, and edges* internal model and an external program to dynamically compute the data positions and connections from user input and "pipe" the data descriptions to *DX*. Several capabilities of the *faces, loops, and edges* data model described in the manuals turn out to be not fully implemented, and incompatible with various display and render modules on a system-wide basis. Knowledge of these

unimplemented features comes only by trial and error. As an experienced programmer with a foundation in various assembler languages, Fortran, PL/I, Lisp, Ada, and C++, I found the visual programming interface and dataflow programming paradigm completely unfamiliar. It was easy and exciting to compose visualizations at the simple import, transform, and render level. Still, the graphical manipulation of modules and filling in of predefined parameters involved making a lot of mistakes before things "clicked". The syntax and semantics of *DX* modules do not parallel the programming constructs of most textual languages. Complex control mechanisms require thorough familiarity with the available tools. Later at a more advanced level of using *DX*, I continued to find indexing problems easy to "see" in the visualization but difficult to diagnose. When debugging textual programs a listing is of the utmost importance to track down elusive bugs. The *DX* message window receives trace output but there is nothing to print out to illuminate the structure of a visual program. A listing of the *.net* file is available but not particularly helpful in that there is additional unfamiliar syntax introduced which obscures the logic. Once a dataflow network gets large and complex, it overflows the maximum window size on the screen and is impossible to view it in its entirety. At this point it seems helpful to organize groups of modules into macros to encapsulate logic and unclutter the screen even if those macros are only used once.

Evaluation/Conclusions

The Data Explorer software involves a number of components, and the expenditure in the learning curve on startup time is considerable, even for an experienced programmer. Concepts from object oriented design and programming, the dataflow paradigm, geometric representations and 3D transformations are basic requirements for understanding the data model and visual program editor. Simple visualizations of complex datasets are easy to construct in *DX* especially from a good set of examples, yet manipulating modules to build a complex program requires a complete paradigm shift from textual to visual programming. Much practice is needed before the syntax and semantics of the module definitions are mastered and become natural to work with. Control and sequencing

mechanisms that are second nature to an experienced programmer used to working in a variety of textual languages ranging from machine code to procedural and object oriented languages must be worked out with a completely different set of tools. Non-programmers may actually have an advantage at the beginning and intermediate levels of visualization development.

This particular spiral is continuous throughout the extent of the project. Direct input to the next phase (Spiral 1.2) is knowledge of the import process. The on-going risk analysis of this task involves building prototypes to demonstrate both the understanding of *DX* and its operational status. Concurrent with this spiral is the incremental improvement of *DX* itself, through the release of new software versions and updated documentation. As *DX* stabilizes, the risk analysis of the system goes from high to low in terms of reliability. However, risks involving the complexity and extensibility of the system remain high. The examples and literature give warnings of how easy it is to construct visualizations that are misleading both in terms of scale and validity of transformations. Future plans regarding *DX* involve an on-going learning curve, paralleling continued development of system capabilities in new releases of the software.

3.3.1.2 Spiral 1.2: Learn about MERGE and import data

Table 3.2. Spiral 1.2

Objectives	Learn about <i>MERGE</i> critical states and import the <i>MERGE</i> data
Constraints	TIME: spring '94 semester deadline TECHNOLOGY: <i>DX</i> in beta-test mode with limited support for connections-based data
Alternatives	<i>DX</i> file format General Array Importer Browser program can scan files (capability added in Release 1.0) Size of arrays range from small to huge(3x3 ... 7500x7500)
Risks	Direct data representation used by <i>MERGE</i> may require a pre-processing filter potential bugs or limitations of connections-dependent data
Risk resolution	Prototyped <i>DX</i> file format, used Browser to scan the data files
Risk resolution results	<i>DX</i> file format works for full array import
Plan for next phase	Go on to import scattered data points as Sequencer input required in Spiral 1.4
Commitment	Continue with next format required; subject to rework

Goal

The goal at this stage is to demonstrate the ability of the *DX* import facility to successfully represent the *MERGE* data by developing a prototype that imports a small sample grid. Small data sets are useful to demonstrate actual details; large data sets confirm overall system performance and behavior [Brown-92].

Discussion

Results indicate that the data can be specified in 2D as an $n+1$ by $n+1$ grid of $n \times n$ integer data values defined as dependent upon the connections component. Connections-based data allocates a constant data value to the area defined by the intersecting grid lines, rather than interpolating a value based on values at neighboring grid intersection points. The *DX* General Array Import facility is used to construct a header definition that details how to read the actual *MERGE* input file. *MERGE* documentation indicates that the rectangular data grid is processed row by row from upper left to lower right. To provide visual consistency with the way that *MERGE* addresses

the data, the grid is to be accessed in row majority order defined at the origin with unit increments in the positive x direction and the negative y direction. The default storage order in *DX* is specified by *MAJORITY = row*, yet the *MAJORITY* keyword is only available in the General Array Import format and not the *DX* data format specification or the Applications Program Interface in this initial release of *DX*.

Evaluation/Conclusions

An evaluation of the import process is that it is sufficiently flexible to handle data based on regular positions and connections, such as *MERGE* input and output data arrays. Test files for *MERGE* input are available in both ASCII and binary (unsigned byte) format. Both are able to be read by changing the type and format statements of the header section of the General Array Import specification for the data grid defined below.

```
# Data Explorer import file for 10 x 10 image
file = Im10.in
grid = 11 x 11
dependency = connections
type = ubyte
structure = scalar
format = msb ieee
header = bytes 128
majority = column
positions = 0,1,0,-1
end
```

3.3.1.3 Spiral 1.3: Show static visualization of MERGE input

Table 3.3. Spiral 1.3

Objectives	Show static visualization of <i>MERGE</i> output
Constraints	TIME TECHNOLOGY
Alternatives	Color options: autocolour, colormap, direct color table Data layout & origin Grid specification Size most useful? 2D or 3D plan for compatibility with changing data
Risks	Incompatible with future requirements
Risk resolution	Show simple 2D view using Autocolour Experiment with different array sizes
Risk resolution results	Different amounts of data appropriate for different purposes Specific color mapping not too significant
Plan for next phase	Determine animation and how to modify data values before doing much more
Commitment	complete, subject to rework

Goal

The goal of this cycle is to produce a static visualization of *MERGE* data that satisfies several different criteria based on future plans for animation:

- Allow differentiation between raster polygon areas. This is satisfied by applying a continuous colormap to the discrete cell values to show similarly colored adjacent cells as large polygons.
- Designate areas with data values that are to be ignored by not coloring them at all.
- Lay out data compatible with *MERGE* addressing scheme, by row from top left to bottom right.
- Be able to illustrate the "merging" of cells. Color displays global patterns very effectively.
- Minimize bandwidth, i.e. the amount of information that is passed between processes.
- Highlight activity.
- Allow for control over the animation by storing sequences to be replayed.
- Allow graphical data query and manipulation.

Discussion

Implementing a static visualization of the *MERGE* data is relatively simple. The input is already defined as an $n \times n$ array of integer cell values which translates easily into a *DX* object via the various import facilities as discussed in Spiral 1.2. Text glyphs are useful on the smaller grid sizes to identify class values. Additional attributes of the data are communicated by the addition of color which has the potential of conveying a lot of information efficiently. The data values of the cells are colormapped to an eye-pleasing spectrum ranging from blue to red. At this point neither the absolute or relative color assignments are significant. Color is used in this static view primarily to encode the state of implied data structures by differentiating area polygons. Consideration is given to the five ways of implementing color for animation purposes for application to future dynamic views, specifically how color can be used to highlight activity [Brown-88].

Evaluation/Conclusions

In summary, the selection of input data strongly influences the message conveyed by a visualization. The amount of input data is significant to convey information for different purposes. Sizes ranging from 3×3 to 10×10 grids are good for experimenting with "cooked" data and to introduce the detail required to examine special cases. A larger 100×100 grid is useful as an overview of the process without incurring too much overhead. Textual annotation is useful to introduce small data cases and aid the users in relating these initial visualization displays to their previous understanding of the process and the application specifications. Once the user makes these connections, larger and more interesting data sets convey larger patterns and the same degree of annotation only adds unnecessary clutter to the picture.

It is foreseen that this spiral may be reworked after an animated prototype has been developed. Revisiting this spiral after developing the visualization of the similarity matrix in Spiral 2.4 has already yielded an idea for an alternative visualization that involves shading the polygons relative to the selection function value.

		0							10	
0	2	2	1	1	5	5	6	6	6	6
	2	2	1	1	1	5	6	6	6	6
	2	6	1	1	8	8	5	5	3	3
	2	6	1	1	8	8	5	5	3	3
	2	6	1	5	8	8	8	8	4	4
	2	6	6	5	8	8	8	8	8	2
	6	6	6	5	8	8	1	8	8	7
	6	6	6	8	8	1	1	1	2	8
	6	6	6	8	8	1	1	4	4	8
-10	3	3	3	3	3	3	3	5	5	4

Figure 3.3: Static Visualization of *MERGE* Data

3.3.1.4 Spiral 1.4: Determine What and How to Animate

Table 3.4. Spiral 1.4

Objectives	Determine what to show during the animation
Constraints	TIME; TECHNOLOGY
Alternatives	<p>1.Offline / Full Array: Prior to initiating <i>DX</i>, run <i>MERGE</i> to generate successive images of full array data. Read in to <i>DX</i> as series objects; currently working with 100x100 array of integer values on <i>DX</i> regular grid.</p> <p>2.Synchronous / Full Array: Run <i>MERGE</i> as a synchronized communicating process with <i>DX</i>. Read in series(s) of n x n gridded data values as generated.</p> <p>3.Offline / Scattered cells to update: Prior to initiating <i>DX</i>, run <i>MERGE</i> to generate marked sets of scattered data values to update. Pass i,j,data points: Bandwidth = #points * (sizeof(j) + sizeof(j) + sizeof(data))</p> <p>4.Synchronous / Scattered cells to update: Run <i>MERGE</i> as a <i>DX</i> synchronized communicating process, generating marked sets of scattered data values to update.</p>
Risks	<p><i>MERGE</i> logic unknown</p> <p>SEQUENCER: enough control over series objects</p> <p>FRAME RATE - THROTTLE: enough control?</p> <p>CONTROL PANEL: access to interactors: integer, scalar, selector</p>
Risk resolution	<p>Prototype 1: Implemented Alternative 1 with several variations:</p> <ul style="list-style-type: none"> a) 3x3 array, artificial values b) 10x10 array, real <i>MERGE</i> input c) 100x100 array, real <i>MERGE</i> input <p>Prototype 2. 3-D simulation of 5 series of 10x10 arrays generated off-line with different threshold values controlled by the sequencer</p>
Risk resolution results	<p>Requested modification to <i>MERGE</i> for intermediate results</p> <p>New control state information available from trace output</p>
Plan for next phase	Determine how to visualize data from this stage
Commitment	Refine during Spiral 2.4

Goal

The goal of this final subtask of the initial round of feasibility spirals is to determine what is possible to animate based on preliminary knowledge of the system and the application, as well as an understanding of the purpose and classification of algorithm animation projects. An algorithm animation should be designed to illuminate the logic, progress, and internal workings of the executing program. The *MERGE* program is an implementation of a set of aggregation rules as outlined in Section 1.5. The early goal

of the animation project was set to graphically illustrate the sequence of "merges" within the aggregation process to facilitate this collaboration. The application easily lends itself to a direct graphical interpretation. The data is laid out as a rectangular (usually square) grid of data cells. On a macro level, the application experts are interested in seeing the development of polygon areas using different threshold values. On a more detailed level, the individual rule applications are of interest on a case by case basis of properties and arrangements of neighboring cells.

Discussion

At this point in the project, the material that I had available to specifically detail algorithm animation was [Brown-88] and the accompanying video. The specific techniques of algorithm animation mentioned by Brown are concrete and well-specified yet the levels of abstraction of Brown's examples and the capabilities with respect to *DX* differ to the extreme. The level of abstraction of Zeus capabilities according to Roman, *et al*, is at a very high level of structure and synthesis. Animation of the *MERGE* algorithm at that level would require detailed knowledge of the implementation with respect to internal data structures as well as a meta-language built on top of *DX* to map program features to graphic primitives. [Guo-93] provides some documentation and a C++ version based on the actual Ada implementation. These sources provide insight into the complexity of the still-evolving nature of the data structures, yet the problem of determining what to animate at the high level of abstraction necessary to characterize algorithm animation with the capabilities of *DX* seem to involve a wide range of unavailable alternatives and a high degree of risk. Successful illustration of the *MERGE* process under the current set of constraints at a low level of abstraction can be classified as effective *process visualization*.

I decided to limit the alternatives to what appeared feasible on a first round of using *DX* to produce an animation. The first significant problem is the lack of time-dependent data. *MERGE* takes as input a grid of data cells, several parameters including a threshold value,

and runs to completion which culminates in the output of one final array where all polygon areas are larger than the threshold. Currently there are no intermediate results to indicate the succession of merges. I decided to focus on intermediate results later, and pursue an initial prototype of animating full images that were merged to different threshold values using the sequencer as a tool to control the animation. The full image prototype was successful yet showed some shortcomings of *DX* regarding speed control. *DX* has a *throttle* parameter denoting the number of seconds to allocate to each frame, but the actual animation speed is more a function of the complexity of the scene and whether it has been cached or not, and the computational power of the workstation on which the visualization is executing. Images can be shown in either forward or reverse and continuous or step mode. Again I found it helpful to annotate the series. Several different 3D views and levels of abstraction were experimented with in Prototype 2, none of which were extremely illuminating.

The real goal of animating the *MERGE* process is to graphically illustrate the implementation of the *MERGE* aggregation rules as outlined in Section 1.2. The intermediate results of each merge operation are required to demonstrate this. Considerations of bandwidth being the amount of data passed between two processes, affect alternatives 2, 3, and 4. Passing the full array as Alternative 2 was not seen as desirable, leading to overhead as follows:

Bandwidth Analysis: $O(N^2)$
100x100 grid -> 10,000K
1000x1000 -> 1 Mg
7500x7500 -> 50 Mg

Alternative 3 (Spiral 1.4) and Alternative 4 (future Spiral 4.0) were decided on after the prototype implementing Alternative 1 was successful. An alternative method of implementing the visualization within Alternatives 3 and 4 is to preprocess the data value changes for each cell into a vector of value for each cell. The vector field can be built

in 3-space, and sliced to reveal all values less than the current time stamp. This alternative was rejected as being inconsistent with future real-time goals but does feed into a suggested alternative in Spiral 3.3 which involves a version based on building the vectors dynamically. A specification for the desired import format of scattered data cells to update was agreed upon and tested before the actual modification was made to the *MERGE* code. The General Array Import file is listed below.

```
# Data Explorer import file for series of scattered points
file = merge.trace
points = 3
series = 17,separator = lines 1
field=locations, olddata, newdata
structure = 2-vector,scalar,scalar
type = int, int, int
format = ascii
header = marker "*** Series # 1\n"
interleaving = field
end
```

Prof. Ford modified a version of the program (*MERGE18*) to produce the necessary output to generate an animated trace of "merges". A succession of the desired image frames is illustrated in Figure 3.4.

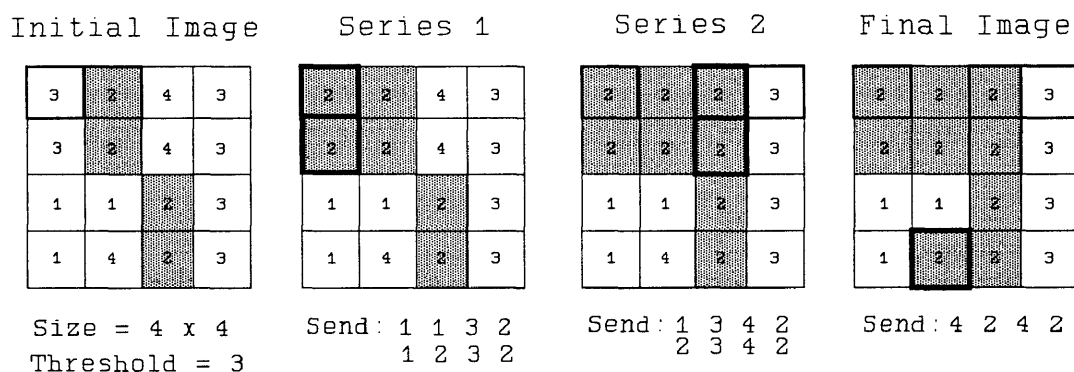


Figure 3.4: Series of Changed Cells

Examination of this level of output is illuminating. At first there appears to be no reason to pass over the old data value with the new data value. After all, the old data value is currently known at the receiving end. Having two data variables in one series is certainly possible, yet the resulting hierarchy of the imported data is surprising in that two separate series are actually created. The *DX* documentation implies that the data series share the same positions and connections components, and effectively they do, with separate pointers to the positions and connections components for each of the series objects. The number and sequence of modules to extract the desired information from this structure, however, is sufficiently complicated that it should be represented by an example and available as a standard macro.

The *merge trace* records are the output that is created to show which cells are to be changed with each "merge". A listing of the *merge trace* records occasionally shows old values equal to the new values. This first appeared (to me) to be a bug in the *MERGE* trace records but the explanation is that these values indicate a significant control state. These records are the "*cascaded effects*" [Ford-94], where adjacent polygons with the same cell value are being merged.

Evaluation/Conclusions

This cycle is significant in resolving one of the biggest outstanding risks - that the application has been up until now essentially a black box. A certain set of rules are implemented yet there is no evidence of the process except the final single output image. At this point, the form of the intermediate results is known, and the question of what to animate is clear; how to do it is the subject of Spiral 2.0. It is also apparent that once it is determined how to animate, it will be worthwhile to show different versions of the algorithm side by side (Spiral 2.4).

3.3.2 Spiral 2.0: Primary Goal: Development

The goal at this point is to develop a visualization based on the results of the initial round of feasibility studies. Three separate subtasks involving the technical development details dependent on *DX* module semantics are apparent during Spiral 1.4.

1. A technique is needed for mapping the individual points and new cell values to the existing grid (Spiral 2.1).
2. Graphical data inquiry and manipulation are necessities for interaction and are available through the facilities of the Pick and Probe modules which are implemented in Release 1.0. This is the topic of Spiral 2.2.
3. Once the individual cell values are applied to the grid object, a method is needed to retain changes for the next application (Spiral 2.3).

After these three technical difficulties are resolved, Spiral 2.4 can commence to incorporate them into a useful animation and user-friendly interface.

The geometry of the three concurrent spirals and the fourth sequential spiral is illustrated in Figure 3.5.

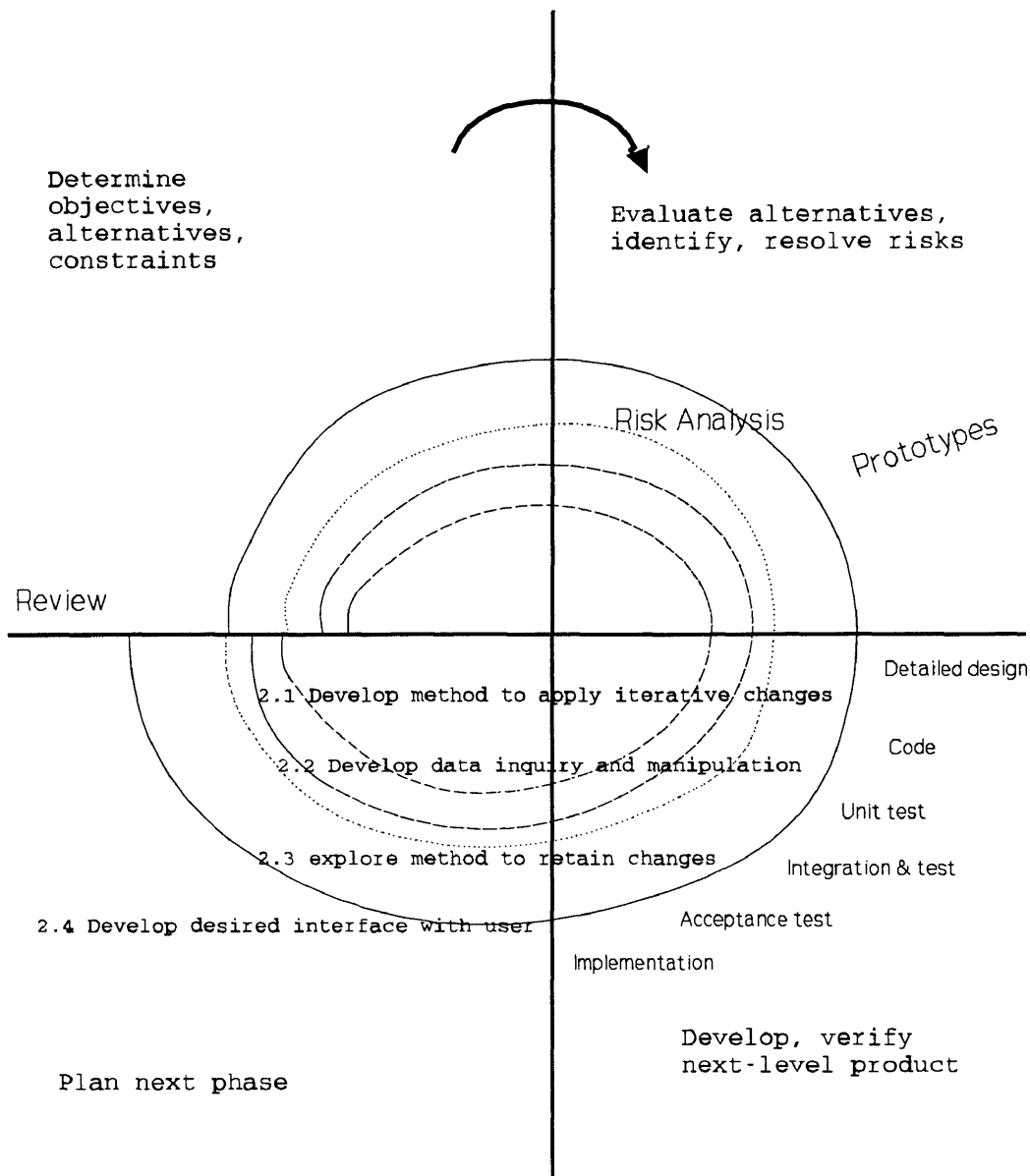


Figure 3.5: Spiral 2.0 Development Subtasks

3.3.2.1 Spiral 2.1: Develop method to apply individual changes

Table 3.5. Spiral 2.1

Objectives	Develop method to apply individual changes to cells
Constraints	TIME; TECHNOLOGY (specifically, support for connections based data and scattered points)
Alternatives	<ol style="list-style-type: none"> 1. Import series of scattered points. Map to grid. 2. Preprocess scattered points to generate invalid connections. 3. Write specialized routine to process cell by cell.
Risks	Modules Import, Map, Regrid, Rubbersheet won't do it; No support in <i>DX</i> for this requirement.
Risk resolution	Prototypes: <ol style="list-style-type: none"> 1. Import of scattered data points, using a) regrid b) map. 2. Reviewed support of invalid positions and connections. 3. Designed user-written module to read individual data points to index into the data arrays and change individual cell values.
Risk resolution results	<p>Prototype 1: failure. <i>DX</i> likes #data values = #connections. Haven't had success with using Map & Regrid on cell-centered data without interpolation. Conclusion: <i>DX</i> import format and data model does not have this capability in the form required.</p> <p>Prototype 2: not desirable. This would just be a delay of $O(N^2)$ processing. Warnings of invalid positions & connections components are not compatible with all modules.</p> <p>Prototype 3: success with user-written CHANGECELL module.</p>
Plan for next phase	Expand on flexibility of CHANGECELL module
Commitment	Subject to rework (see discussion under Spiral 3.3)

Goal

The goal is to apply the data for each individual "merge". This spiral exists because there does not appear to be an obvious way to implement this with the given set of *DX* modules.

Discussion

Given a set of positions and new data values, the problem is to change the corresponding data values in a *DX* object. Specifically, the plan is to import a set of scattered data

points and update the appropriate data element in a regular connections-based object *without* interpolation. The data is in the form of a *row number*, *column number*, *old data value*, and *new data value* for a set of individual data points, and needs to be mapped to the corresponding data cell in the original image. There are an indeterminate number of sets of points in the series, depending on how many "merges" are necessary to combine all polygon areas to the minimum threshold size. Each set may include from one up to the threshold number of points.

Three alternatives are considered. First, experimentation with *DX* modules for Import, Map, Regrid, Rubbersheet, etc., in Prototype 1 reveals inconsistencies in support for scattered points based on connections-based data. Data is interpolated when it should be constant for any point mapped to a cell. There also appears to be no way to modify an individual cell value. Discussions with IBM support personnel confirm this and also the approach suggested by Alternative 3. Meanwhile the second alternative is suggested by the *DX invalid positions and connections* approach. This designation of flagging certain elements of the positions and connections components as invalid is available to process only parts of an object. However, other student visualization results and warnings in the manuals indicate that this is not uniformly supported by all modules. Also, it appears that this option retains the $O(N^2)$ overhead.

Alternative 3 is implemented successfully. It involves a user-written module in *C* currently implemented as `CHANGECELL(object,index,datum)` with an cell-centered grid **object** to accept an integer as an **index** into the cells and output a new cell-centered grid with that cell modified to the integer input value **datum**. Writing a *DX* user module is a good introductory education into *DX* internals and inspired the attempt to diagram the class hierarchies implied by the data model included in Appendix A. The use of the *Builder* program to generate an outboard module is recommended and the class hierarchies aid in understanding the recursive code that is generated. Listings of the *mdf* file, *source code*, and *makefile* for the `CHANGECELL` module are listed in Appendix B.

Data Explorer can be configured to automatically load outboard module descriptions with the *-mdf filename* environment variable on the command line. The user module assigned categories then appear on the category menu. If the *mdf* file is not specified user modules are not found when network programs are opened, and are deleted from the VPE window along with all their connections which is a very efficient reminder but not very helpful.

Evaluation/Conclusions

The experience of writing a user module provides good insight into the workings of the *DX* dataflow model, but it is necessary to devote a lot of time to implementation details. The *DX* Repository at Cornell University is collecting and promoting the creation of user-written modules, but in this case, *CHANGECELL* seems to be necessary to correct a deficiency in *DX* that could be consistently handled with an extension to the current library of modules.

3.3.2.2 Spiral 2.2: Explore graphical data inquiry and update

Table 3.6. Spiral 2.2

Objectives	Explore graphical data inquiry and update
Constraints	TIME; TECHNOLOGY
Alternatives	PICK, PROBE modules, pick data structure
Risks	Above features not working or poorly documented
Risk resolution	Testing of Pick: a) graphical data inquiry b) graphical data update
Risk resolution results	PICK works, must be careful to map coordinates to "picked" object to get world coordinates and correct data values; bug in Pick output with autoaxes option on; Pick data structure procedures unclear.
Plan for next phase	Avoid using autoaxes with images in PICK mode Try accessing the pick structure
Commitment	Can delay trying pick structure until really needed

Goal

Interactivity depends on user input and appropriate system response. The basic tool for interactive visualization is the ability to graphically select objects and query on attribute values. The user should also be able to modify attributes as needed to influence future system behavior. The goal of this subspiral is to investigate the capabilities of *DX* with respect to these requirements.

Discussion

The PICK and PROBE modules are new additions to Release 1.0. PROBE returns any selected point in 3-space whereas PICK anchors a selection to the surface of an object or multiple objects. The PICK module also constructs a *pick structure* which is documented as being able to return the exact component of a complex object, even down to the individual positions element [IBM-94, PG p.7-1]. It is necessary to write a user module to access the pick structure but the documentation on how to traverse it is unclear. More sample programs would be useful. The risk of using this facility successfully appears too high to be attempted at this point.

PICK returns an [x y z] vector in screen coordinates. The Map module is required to convert to world coordinates. Misleading coordinates are currently returned when the *autoaxes* option is selected for the image window due to the axes generation "changing the geometry" of the objects in the rendered image. Currently the PICK module is used in the algorithm animation to select the source cell to inquire on. Data values are written to the message window in addition to this information being apparent from the display. Future development of the animation may find a more useful purpose for data inquiry.

Evaluation/Conclusions

Due to the simple geometry and low level of abstraction of the display, the data inquiry feature is not an essential part of the algorithm animation. The main result of this spiral is to demonstrate the implementation of this capability in *DX*. The mapping of screen to world coordinates does work with respect to connections-based data as it should, yet the output of the pick data structure appears to be a more useful implementation in terms of reliably connecting to the physically picked component. Currently an index into the data array that corresponds to the "picked" cell must be computed from world coordinates mapped from PICK. The pick data structure documentation implies that this index is available directly, but other problems with respect to the order of data storage have been inconsistent. Hence the decision to delay accessing the pick structure until really needed. The disadvantage of computing the index is having to know details of the import process and internal data organization.

3.3.2.3 Spiral 2.3: Implement iterative changes

Table 3.7. Spiral 2.3

Objectives	Be able to retain iterative changes to an object
Constraints	TIME; TECHNOLOGY
Alternatives	<ol style="list-style-type: none"> 1. CACHE 2. EXPORT & re-IMPORT the new image each time 3. script mode 4. User module read and write updated image implicitly
Risks	Nothing will work
Risk resolution	<ol style="list-style-type: none"> 1. Test the caching routines to accumulate iterative changes to data. 2. Try EXPORT & re-IMPORT - document performance impact 3. Read up on it. 4. Same effect as Alt #2.
Risk resolution results	<ol style="list-style-type: none"> 1. Cache routines are incompletely specified, and not working yet across module executions. CACHING: "DX not tailored to maintain state" 2. Using this option for now, need to turn module caching off for the IMPORT module that reads the last image. 3. WHILE and IF are marked for future implementation in script mode. 4. Currently used in MODFLOW visualization
Plan for next phase	Retry caching at later date.
Commitment	Medium

Goal

Individual data cells in the main *DX* object that represents the *MERGE* data grid are updated and rendered for display. Subsequent executions do not retain the data cell changes which are necessary to achieve the desired animation results. The goal of this spiral is to determine how to retain or accumulate the results of successive changes to a *DX* object.

Discussion

The *DX* dataflow model includes implicit support for reusing the results of previous computations. Options for caching each input are available at the module level. Explicit access to cached results is what is needed. If the output of *CHANGECELL* can be cached with a known key, it can be retrieved for use as input by subsequent executions. IBM support confirms the validity of this approach and suggests the addition of the

CacheIn and CacheOut modules. Current problems caching the results of outboard modules are also confirmed.

CacheIn retrieves the cached object from the previous execution. Output is possibly *null* and should be Inquired upon before depending on the results. CHANGECELL represents any set of operations to modify the object. The output is passed to CacheOut to store the object with a known key and a permanent caching designation (explicit deletes of cached items no longer used should be done but are not shown here). Any set of keys (such as one for each object in a series) can be constructed. A sample network diagram of how to use these facilities is shown in Figure 3.6.

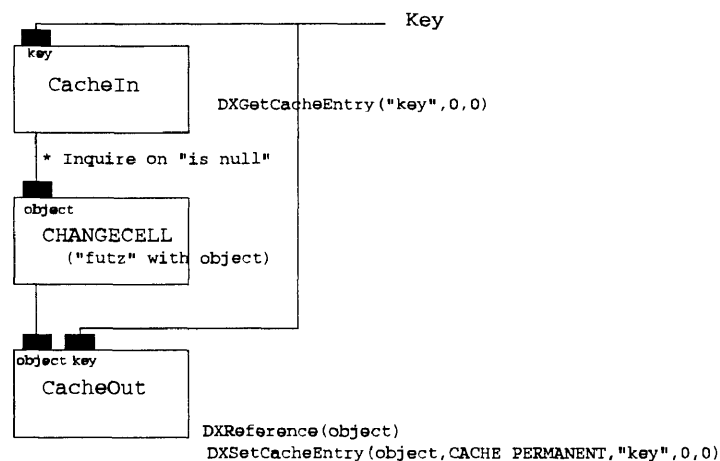


Figure 3.6: Sample Network of Caching Modules

Evaluation/Conclusions

"Because Data Explorer modules follow pure function semantics, the cache should **not** be used to store state that affects the output of the module. A module must always be able to recreate the object from the same set of inputs; the cache should only be used as an optimizing tool." [IBM-93,

Programmers' Reference Manual, p. A-70]

Thus, the dataflow network model places some restrictions on the behavior of the modules. A module's outputs must be based completely on the inputs rather than on some internal or external state derived from a previous execution. In this project implementation, the object is currently maintained on disk using the export and import facilities. This method of maintaining state has some impact on performance but has proven reliable.

3.3.2.4 Spiral 2.4: Develop desired user interface

Table 3.8. Spiral 2.4

Objectives	Develop desired user interface
Constraints	TIME; TECHNOLOGY; application goals
Alternatives	Initial, final image, animated image Single or multiple windows Sequencing control What do update with PICK Similarity function Different colormaps and labelling options
Risks	Can't do simple sequence of animated tasks
Risk resolution	Construct and demonstrate prototype
Risk resolution results	Most features implemented in complex dataflow network; Complex visual program hard to reprogram
Plan for next phase	Continue
Commitment	Medium/High

Goal

The desired user interface at this level of development is to provide the initial data grid, the derived data grid, and the similarity matrix with several control options to demonstrate the process and progress of the *MERGE* algorithm. The process to be demonstrated is the order of merges and the selection of the "best" neighbor with which to merge.

Discussion

The sequencer provides control over the series of "merges". The sequence can be played forward or backward, in single step or continuous mode. Applying changes to the updated image in reverse, however, yields illogical results. There is no way to prevent the user from doing this. Also, sequencer controls are not adequate enough. The user must click on the ellipses of the sequencer and reset the *next* value to zero. Instead there should be a way to have explicit control over restarting the sequence by clicking on a control panel option. *Next* should be an input tab on the Sequencer module. Another constraint is that only one sequencer is allowed per network. Another looping mechanism

is needed to cycle through the points in each series. This is programmed by passing the points as a generic array and looping within the CHANGECELL module, explicitly skipping values that denote dummy data points.

BUILDSIM, arbitrarily written in C++, dynamically builds the similarity matrix, shown in Figure 3.7, according to the maximum size of the data values of the imported grid. BUILDSIM runs as a Unix shell program initiated dynamically but not linked to DX. Parameters are dynamically determined and passed to the shell program, which is relatively easy to do and works well, using the Statistics module to pass the value to Format and generate the string. Once the target cell and its source and neighbor data values are known, the function value can be highlighted and used to color the data grid with the colormap of the similarity matrix object. The mapping to different data values is extremely tricky to program with the given modules. Logically, this is a very simple operation that merely exchanges data values in one array for another, and again should be available as a standard macro. Also, control facilities for managing the two options of picking or sequencing are cumbersome. This approach was abandoned due to the high level of confusion, the lack of ability to specify defaults for the control paths, and the lack of control over the necessary sequence of user events. After several iterations, output appears to be non-deterministic. Debugging different sequences giving the lack of control over asynchronous user options is difficult.

		Neighbor			
		1	2	3	4
Source	1	-	1.0	.5	.6
	2	1.0	-	.9	.8
	3	.5	.9	-	-.2
	4	.6	.8	-.2	-

Figure 3.7: Similarity Matrix

The sample similarity matrix is symmetric, but selection functions are not necessarily so, i.e. merging 1 to 2 may not be the same function as 2 to 1. Here a source data value of 2 has a higher value if merged to a 1 than a 3 or 4.

The derived image is labelled with the old and new data values as the cell changes are applied. A different message is put up if the merge is a cascaded effect since no new cell value or color change is indicated yet a significant state change is to be noted. The similarity matrix value of the merged-with neighbor is highlighted.

Evaluation/Conclusions

Many possibilities for extending the animation are seen at this point, but not undertaken due to the time constraint. A brief list follows:

- Animate two different versions of *MERGE*, side by side
- Automate the input generation with flexible parameters for handling different sizes of data grids.
- Revise the implementation with respect to efficiency; specifically, generate quads for new cell values, see discussion of Spiral 3.3.
- Partition large data sets across processors
- Build polygon lists in the small simulation

3.3.3 Spiral 3.0: Primary Goal: interactivity

Burnett, *et al*, emphasize the power of a visual program as a spectrum of capabilities with interactivity as the basic goal [Burnett-92]. The final implementation spiral in this project explores issues approaching real-time communication which is necessary for interactivity. Three subtasks are identified: Spiral 3.1 addresses the question of linking to external modules written in other languages. Spiral 3.2 is about inter-process communication. Finally, Spiral 3.3 raises some performance issues.

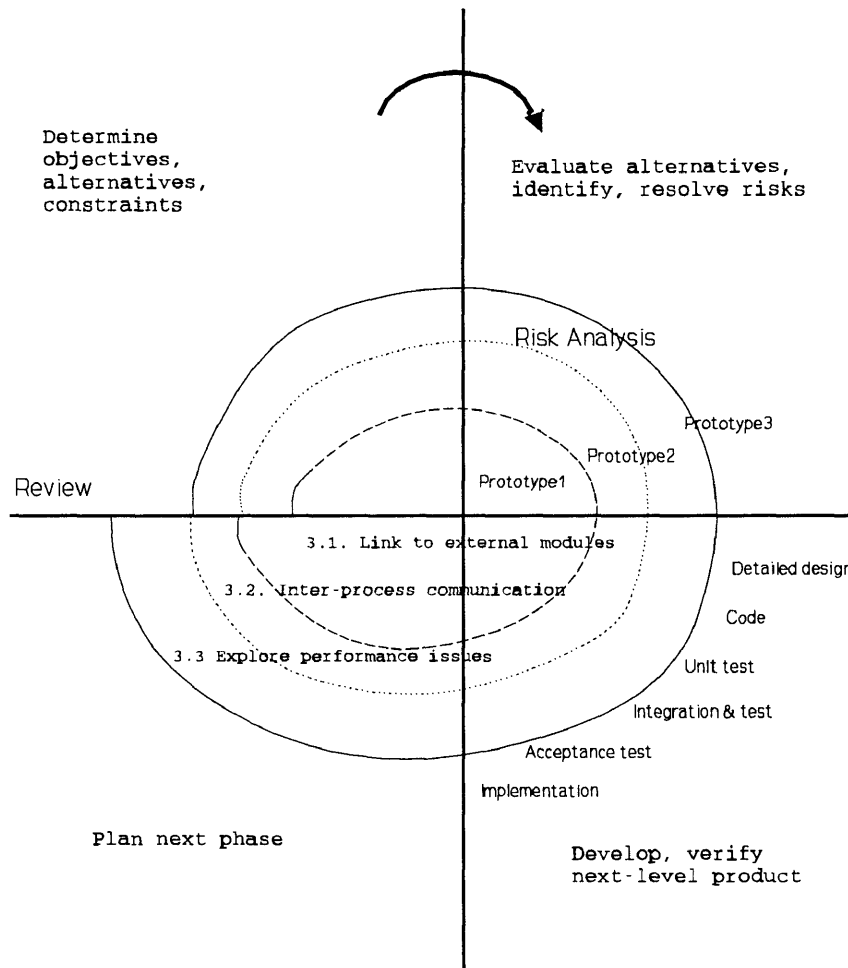


Figure 3.8: Spiral 3.0 Interactivity Subtasks

3.3.3.1 Spiral 3.1: Link to external modules

Table 3.9. Spiral 3.1

Objectives	Be able to link to <i>MERGE</i> as an external process
Constraints	TIME; TECHNOLOGY; C vs C++ vs Ada linkage
Alternatives	1. OUTBOARD modules 2. Modules written in other languages
Risks	Can't do it for several reasons, one being a reported bug in C compiler linking to C++ modules
Risk resolution	Prototype 1. Successful link & execution of OUTBOARD modules. Prototype 2. Successful compile & link to C++ subprogram
Risk resolution results	Bug in OUTBOARD specifications
Plan for next phase	Diagram processes in Spiral 3.2; Test link to Ada; Review inter-process communication plans with <i>DX</i> support
Commitment	On hold

Goal

The goal here is to address a particular problem with the RS/6000 C compiler, that C programs cannot call C++ modules. The risk is that this could impact the ability to communicate with the *MERGE* program from *DX*.

Discussion

Most development of *MERGE* has been in the Ada programming language. A version of *MERGE* is implemented in C++ [Guo-93]. Prototype 2 addresses the reported bug in the C compiler. The resolution is that a C++ subprogram must include the *EXTERN C++* directive and be compiled with the C++ compiler. The main program written in C is compiled with the C compiler. The final link must also be done with the C compiler. Successful results are observed except for the loss of any output directed to *stdout* with *cout <<* statements. Output in the C++ *MERGE9* implementation is written using *printf*, which functions as expected.

Evaluation/Conclusions

A specific risk involving a compiler bug was resolved. In reality, this is not an essential part of the animation. There is no direct need to link to the C++ version; the Ada version is more current, yet the prototype successfully addresses a risky area. With a successful resolution within a relatively small spiral effort (time investment = 1 day), the issue is clarified rather than avoided or obscured.

3.3.3.2 Spiral 3.2: Interprocess communication

Table 3.10. Spiral 3.2

Objectives	Enable <i>DX</i> to pass input to and receive output from another process.
Constraints	Time constraint
Alternatives	<ol style="list-style-type: none"> 1. IMPORT: !<u>pgm1</u> p1 p2 p3 p4 (execute shell <u>pgm1</u> and import what <u>pgm1</u> writes to stdout) 2. OUTBOARD, ASYNCHRONOUS - parameters for a user-defined module to facilitate asynchronous control over an external program 3. DXRegisterInputHandler 4. RPC code
Risks	exact implications of EXECUTE ON CHANGE, DXReadyToRun, and the MDF parameter re side effects are unclear
Risk resolution	Prototype 1: "!BUILDSIM %d < Sim.std" Prototype 2: <i>DX</i> sample program async.c, async.mdf, Makefile.ibm6000, outboard.c, outboard.mdf
Risk resolution results	Inconsistent results, same reported by Dick Thompson
Plan for next phase	Pursue synchronous approach (FileHandler)
Commitment	On hold; review with <i>DX</i> support

Goal

The specific goal is to be able to initiate and communicate with an application program running as a separate process. Communication should be able to go either way. Also, *DX* should be able to send data to the application and receive data directly from the application without writing to a file.

Discussion

The first alternative demonstrates one option for dynamically executing an external program from *DX*. Command line parameters can be formatted and passed to a shell program that generates a *DX* import file with each execution of the network. This method is used to build the similarity matrix yet does not appear flexible enough for communication with *MERGE*. Prototype 2 involves testing the *DX* facilities for asynchronous communication with an externally linked module. The sample module *async* functions as expected to reinitiate itself every *n* seconds under *DX* control. What

is really needed is the synchronous control suggested by the `DXRegisterInputHandler` facility, even though this requires file I/O. Figure 3.10 illustrates a sample dataflow network with `MERGE` represented as a user-written module able to receive input from user control panels and pass output on to the graphics modules. In effect, `MERGE` needs to run as a separate continuously cycling process. The following diagram (Figure 3.9) illustrates a possible configuration:

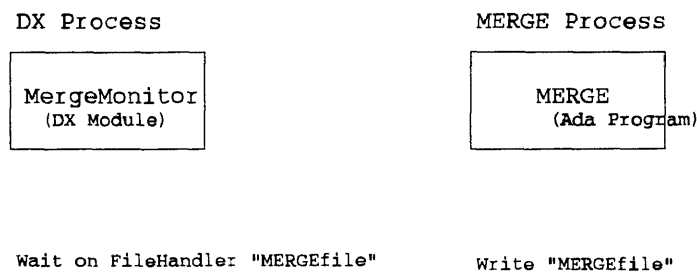


Figure 3.9: Interprocess Communication

Evaluation/Conclusions

It is not clear at this point what options are desired for real-time communication with `MERGE`. The fourth alternative may be the direction to go, once interactive options for `MERGE` are specified. Given that `DX` currently lacks synchronization of Unix file timestamps on successive "import" executions, and various reported bugs exist with `OUTBOARD` modules, this spiral is on hold. There is no software effort worth pursuing at this point.

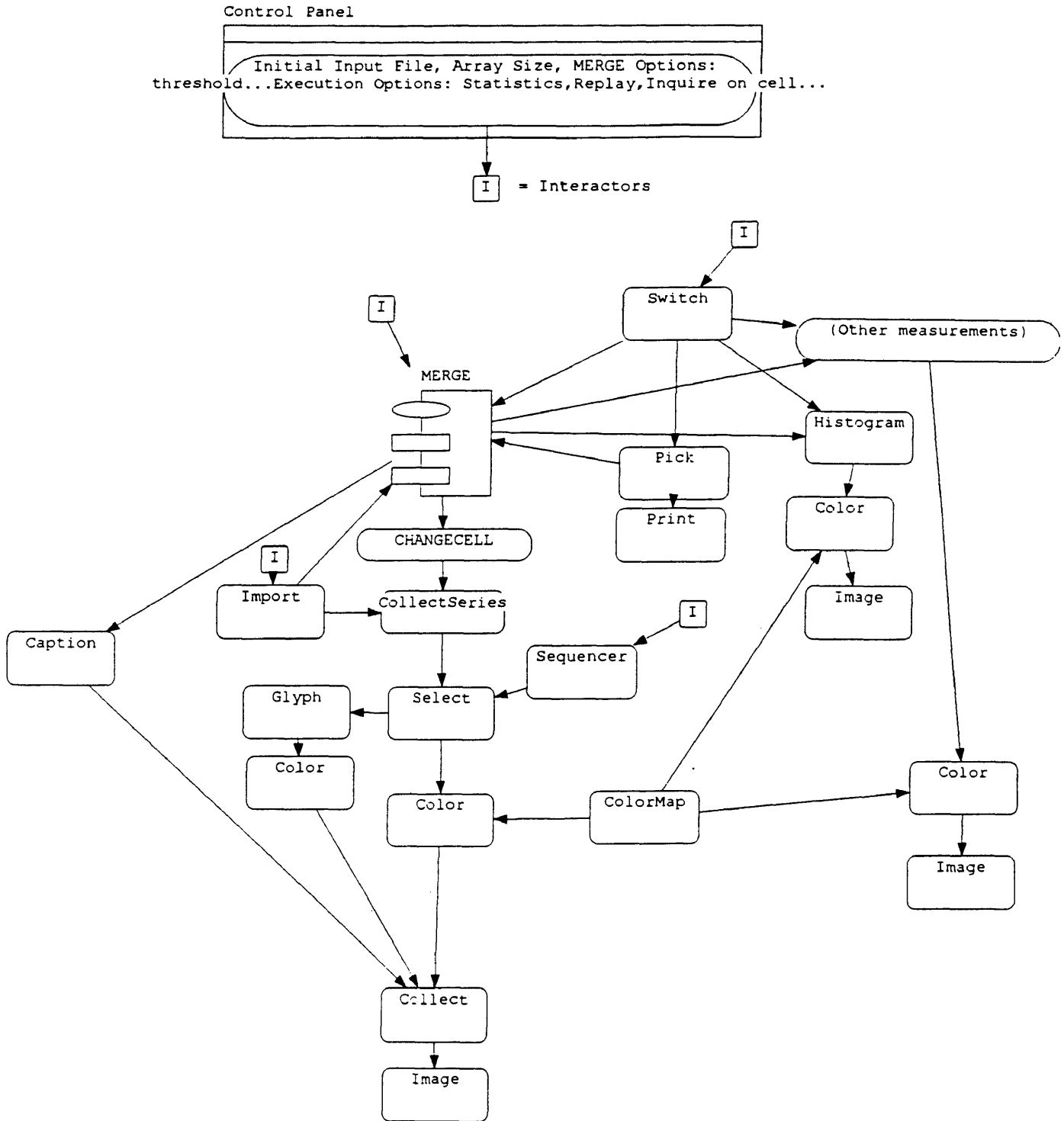


Figure 3.10: MERGE Dataflow Network

3.3.3.3 Spiral 3.3: Explore performance issues

Table 3.11. Spiral 3.3

Objectives	Explore performance issues
Constraints	TIME; TECHNOLOGY
Alternatives	Available <i>MERGE</i> versions: MERGE 9(9K), 17, 18(18K) ¹ Local memory: 32M, 64M, 128M, 256M, ... RS/6000 Processors: 220, 25T, ... Wall clock time vs Trace("time",1) and Usage("memory",1)
Risks	Insufficient knowledge of <i>DX</i> internals
Risk resolution	Benchmark wall clock time Guess at <i>DX</i> overhead
Risk resolution results	Summarized below
Plan for next phase	Determine feasibility based on results, rework dataflow design
Commitment	Plan further tests

¹ The K designations refer to versions modified to produce the merge.trace file for this project.

Goal

By exploring performance issues at this stage, the goal is to approach the whole issue of scaling by determining the feasibility of the current approach in handling different magnitudes of data.

Discussion

Rough measurements using wall clock time were deemed sufficient but *DX* has a timing facility that appeared to be a more repeatable method. Timing samples can be gathered by running the target networks remotely on each of several workstations using the following procedure:

```
login ( to workstation x)
xhost + workstation w
rlogin w
setenv DISPLAY x:0
dx -timing on -memory 64/256 -macros .
```

A module specifying TRACE("time","0") from the Debugging category is placed on the

canvas to cause execution timestamps to be written to the message window. Results were inconclusive and non-repeatable, with wide variations in module entry and exit times. Therefore, the results using elapsed wall clock time are summarized in Figure 3.11. The static rendering was generated with the *execute once* option. The full image series was timed using a set of 10 series objects, each being a full array of the specified size. The updated point series was generated with a threshold of 3 for each size. Ten sets of points, each representing an individual "merge", were applied to gather the timings in order to do a rough comparison with the same number of full image updates. These results do not include processor time for running *MERGE* to generate the images or the *merge trace* file.

Figure 3.11: Performance Timings

Workstation	Grid Size	(Capron)	(Eisgate)
RS/6000 Model		220	25T
SpecMarks(int/ffp)		17/29	62/72
Local Memory		64 MB	256 MB
Static Rendering			
	10 x 10	7	4
	100 x 100	8	5
	200 x 200	10	5
	400 x 400	17	7
	500 x 500	21	8
	800 x 800	34	13
	1000 x 1000	105	17
Full Image			
	100 x 100	41	16
	200 x 200	56	18
	400 x 400	130	32
	500 x 500	234	46
	800 x 800	406	98
	1000 x 1000	1320	216
Updated Point Series			
	10 x 10	108	45
	100 x 100	-	274

The effects of caching were not measured here since the exact overhead of caching outputs of each module is unknown. The results of caching, however, are very observable; once an image is rendered and cached, the speed of the animation is parameter bound rather than compute and storage bound.

Evaluation/Conclusions

A gross analysis of the *MERGE* algorithm overhead in [Ford-94] explores the feasibility of the computation for large datasets, summarized below.

Cost of Processing $N \times N$ Image, Threshold = T

	<i>TIME</i>	<i>SPACE</i>
<u>Algorithm Costs</u>		
N^2 process to store and process	N^2	$N^2 + K$
$K < N^2$ areas to order and process	$K * \log(K)$	-
$K < N^2$ areas to describe	K	-
<u>TOTAL costs</u>	$N^2 + K * \log K + K$	$N^2 + K$

A gross analysis of the complexity of processing by each module in the network regarding the $n \times n$ array is as follows.

Animation Costs

Initial Image size = N^2	N^2	N^2
L modules processing full image	$L * N^2$	
$M \leq L$ modules caching full image		$M * N^2$
J Full image updates from individual area merges ($J=f(T,K)$)	$J * N^2$	$J * N^2$
<u>TOTAL costs</u>	$(L * J + 1) * N^2$	$(M * J + 1) * N^2$
Final Analysis:	$O(N^2)$	$O(N^2)$

Lack of knowledge regarding *DX* internals and further behavior of the algorithm with respect to scale prevents further analysis, but the attempt has shed light on a weakness of the current animation scheme. The CHANGECELL module addresses the problem of how to modify one element of the $n \times n$ array by processing the entire array to make one data substitution. Instead the original grid should be retained and new elements added to it one by one, or T by T, where T is the *threshold* value. Each new cell value can be added to the visualization by constructing a quad and rendering it in 3-space in front of the original grid. By estimating the overhead of each *DX* module, an entirely new approach is suggested, similar to Alternative 4 of Spiral 1.4.

The goal is to redo the prototype and minimize the quantities J, L, and M in the above equations. The final quadrant of Spiral 3, where the next phases are planned, also points to a new spiral to analyze the theoretical implications of the dataflow networks with respect to synchronization and performance. The effect of passing large quantities of data ($O(N^2)$) from one module to the next is as significant as the concern of bandwidth between processes.

3.3.4 Future Spirals

Two future spirals are discussed in this section. Constraints for these spirals have yet to be determined by the continuation of previous spirals as mentioned. Risks and alternatives are not known at this point either. The templates are included here for completeness while the discussion serves to clarify goals and objectives.

3.3.4.1 Spiral 4: **Primary Goal: Dynamic feedback capabilities**

Table 3.12. Spiral 4.0

Objectives	Steering capabilities
Constraints	Depends on interactive capabilities, especially Spiral 3.2 for interprocess communication
Alternatives	Unknown
Risks	Will implement something not useful to developers
Risk resolution	Prototype & demo for the user
Risk resolution results	-
Plan for next phase	-
Commitment	Future

Goal

The goal of this next spiral is work toward *steering*. *Steering* means "the scientist can provide feedback to the computation affecting the visualization and the computation itself as it progresses" [Burnett-92]. Feedback is to be dynamic and bi-directional. The scientist should be able to make unanticipated changes to any aspect of the program as the calculations proceed, thereby aggressively debugging and refining the process to produce better answers.

Discussion

The following description depicts a potential implementation of the animated *MERGE*

process and how steering capabilities might be used with respect to current and future implementations of *DX*.

A full scene (~7500 x 7500 pixels) is the target image for MERGE. DX allows explicit data partitioning to take advantage of parallelism. The DX Partition module chooses the size and number of partitions dependent on the number of processors available, or the user can explicitly control these parameters. Complications of boundary conditions trying to join separately processed regions together can be avoided. Different thresholds and other parameter settings can be applied to individual data areas. The rule application process can be directed interactively and rule sets defined to handle different conditions in different areas of the target data. Visual processes can be fired up to manipulate individual sections of already processed data for specific purposes. Data values relevant to selection functions of different classification schemes can be chosen dynamically.

Evaluation/Conclusions

Only the developers and application experts are able to specify what options are useful for modifying aspects of *MERGE*. The interactive level of communication necessary to support the above scenario is implied but not yet available in the *DX* visualization software package.

3.3.4.2 Spiral 5: Primary Goal: Inferential capabilities

Table 3.13. Spiral 5.0

Objectives	Inferential capabilities
Constraints	Depends on interactive and dynamic feedback abilities
Alternatives	Simulate with cooked data Type of knowledge-based system, programmer-assisted? Reworking of <i>MERGE</i> rules and implementation
Risks	Compute-bound
Risk resolution	Propose simulation first
Risk resolution results	-
Plan for next phase	-
Commitment	Future

Goal

It is possible to imagine a visual interface to an intelligent, rule-based system. The goal of this spiral projecting future development is to include animated interactive expert system capabilities in the goal of completely flexible aggregation software.

Discussion

Inferential capabilities are suggested by the paradigms of constraint and demonstrational programming. "The central idea of the constraint-programming paradigm is to sufficiently constrain the solution value set such that only the desired solution or solutions are possible. Then, ideally, an intelligent system will employ some means to realize what the solution must be. Programmers can demonstrate solutions to specific instances of similar problems and let the system generalize an operational solution from these demonstrations" [Ambler-92].

Evaluation/Conclusions

Only a few references are available in the area of graphical user interface tools and icons to represent expert system constructs, however the goals and rule-based logic of this project suggest development of this ultimate capability. Any further analysis is beyond

the scope of this project, but scenarios of what would be useful to application and algorithm designers at this level could influence alternatives being considered at the level of implementing interactive capabilities in Spiral 3.

4 Conclusion

4.1 What was accomplished

The accomplishments are organized into two sections, product and process. A summary of the product of the algorithm animation project is of interest to the application experts and perhaps to future developers of visual algorithm animations using *DX*. The discussion of the process results of using the spiral software development model is subjective, yet potentially of interest to other users of process models of software development.

Product

This project refines the category of algorithm animation by implementing the early stages of a complicated "process visualization". Current visualization results effectively simulate the merge process with a "specification demonstration" at a low level of abstraction, yet high level of direct information. Specification details are revealed and implementation details are hidden in accordance with the principles of object oriented design.

The final product of this project consists of three separate visualizations, detailed in Appendix D. First, the macro view (Spiral 1.3) demonstrates the effects of merging the same image to different thresholds. Although it is relatively unsophisticated in the level of abstraction or detail as a non-interactive simulation using inputs generated off-line, it is of interest to the application developers as an overview of the entire process, and is easily extended to demonstrate multiple algorithms side-by-side. Second, the micro view (Spiral 2.4) is designed to integrate as much information as is available with a limited bandwidth regarding the *MERGE* algorithm rule application process. It needs to be expanded to flexibly handle different input sizes, yet is successful as a simulation to illustrate what is unavailable regarding internal data structures (specifically neighbor lists)

to the animation process at this point of still using purely off-line data generation for tracking purposes. Third, the interactive view of the selection function demonstrates more potential for animation of the rule application process.

Process

This project had the production of this paper under a time limit as its primary constraint. The spiral process model is very appropriate for this software development situation. The initial directive to show what's possible within the framework of current *DX* capabilities was satisfied just within the time allotted. However, the built-in flexibility of the model can be taken advantage of. In retrospect and rereading the exact steps for each phase, the model was not strictly adhered to in regard to the following points:

- prioritize risks;
- fully evaluate alternatives (would have presented more alternatives to caching);
- complete each cycle with a review or walkthrough; and
- partition product into increments outlined for successive development and documentation before cycle is completed.

Only after two cycles were complete did it seem possible to break the tasks into separate spirals. Once the templates were created, it was easy to fill them out from a sequential log of computerized notes. A strong recommendation would be to do this as each task is recognized and determine the groupings of the tasks later. Admittedly this takes practice and discipline.

It should be realized that a meta-spiral of documenting the process is defined outside of all of the development spirals described. Documentation is the most effective (and time-consuming) influence on the final outcome of the development. Only by writing about, not actually performing, the timing tasks at the same time as other tasks, are the connections made that lead to further development. It does appear that without a time constraint the revisiting of each stage could involve an infinite spiral of discovery.

4.2 Future Enhancements

The outlook for extensions of this project is optimistic from both the developer and application area point of view. Animated demonstrations are useful to generate support for future projects that involve detailed specifications of hard-to-picture results. Truly, *a picture is worth a thousand words* in future discussions between the algorithm and application developers. Future enhancements to the IBM Data Explorer platform will provide an appropriate environment for continuation of this effort. The *MERGE* algorithm appears to be an ideal application for algorithm animation, being significantly compute and memory bound, yet conceptually simple to visualize.

4.3 Lessons Learned

I gained valuable experience in practical application of the spiral model of project development and new paradigms for programming in a visual environment. I would choose to use elements of the spiral model in even more general (and non-software development) decision-making situations. Even though the content of this paper will technologically be out of date very soon, hopefully the structure and practice of documenting the spiral process will be applicable to real-world projects with real-world constraints. I was able to match a personal interest in the areas of math and art with "hands-on" knowledge of the field of visualization. I find interactive design of three-dimensional data forms very exciting, and plan to continue development of software to aid the design of 3D art forms (specifically new wildflower patterns for Tiffany-style stained glass lamps). The current status of *DX* rekindles fond memories of debugging systems software and working with IBM's full level of support, as well as a tendency as systems programmer to play with software package design problems longer than warranted.

Object-oriented principles were brought to life with the application of the *DX* data model, yet were most convincingly portrayed by observing the method teaching of Prof. Ray Ford. He consistently personifies the beneficial characteristics of encapsulation, message-passing, and information hiding both in the classroom and as a project leader. Finally, I gained insight into the real-world requirements of scientific laboratories for tangible research tools.

4.4 Summary

This project explores the intersection of scientific data visualization, visual programming, and algorithm animation, with interactivity as the ultimate goal. The state of the art of interactivity, where the scientist affects the computation as it progresses, is still cumbersome. A flexible, easy-to-use, experimentation environment for the researcher is still to be realized, yet initial prototypes have been implemented for animation of this specific application that incorporate sophisticated graphical techniques without having to deal with the lower-level nitty-gritty details of a textual graphical programming language such as XWindows. The visualizations incorporate static data display, data and process animation, and a user interface for graphical data query and manipulation at the *tracking* level of visual programming capabilities. The application goal, "to create a general purpose, user-friendly aggregation package that provides predictable and repeatable results, and which is usable in a wide variety of applications" [Ford-94], will be enhanced by the continuation of the *MERGE* algorithm animation project.

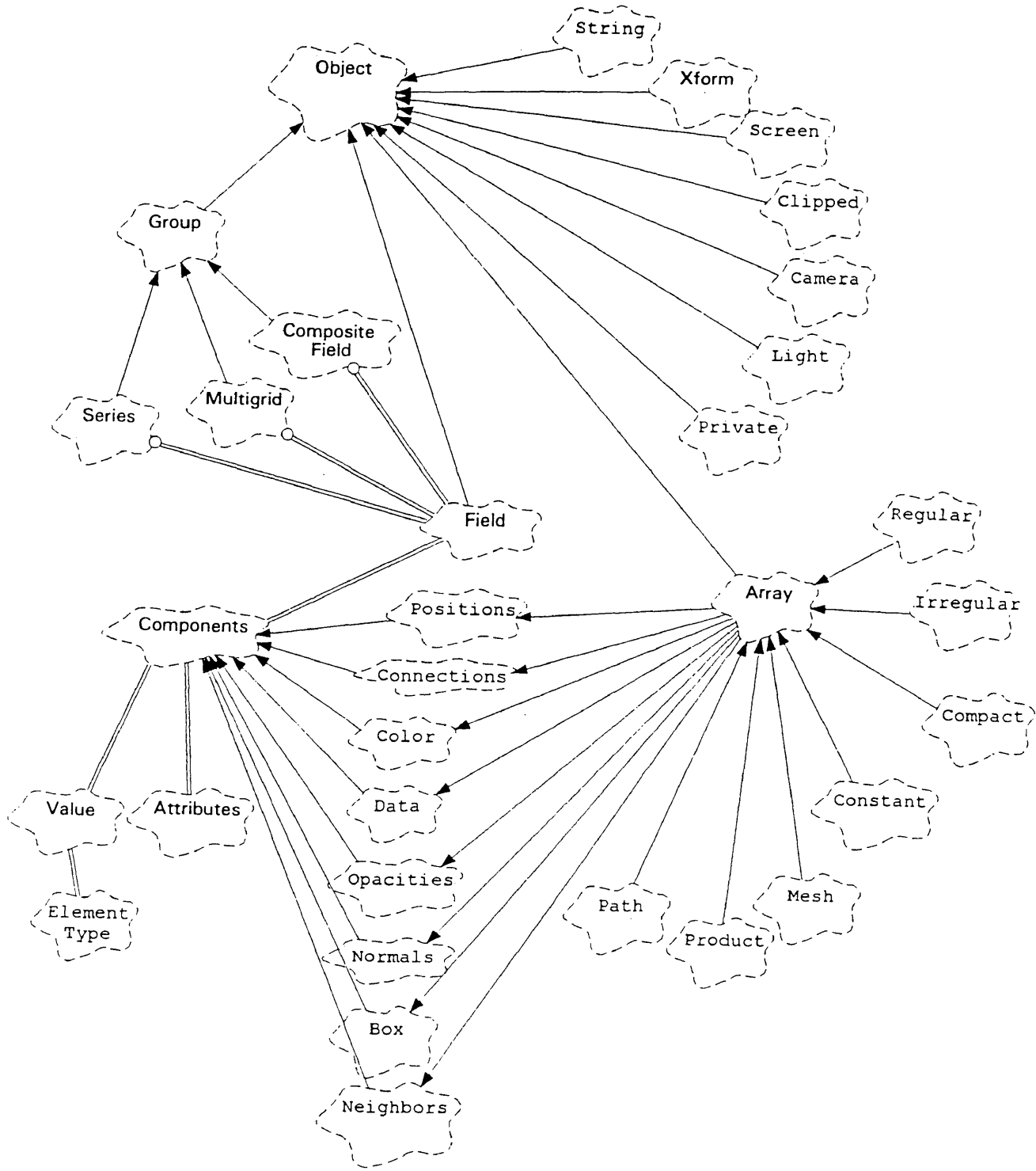
The state of the art of visual languages for visualization is demonstrated by the current implementation of *DX*, and its successes and failures. The semantics of the visual language constructs, execution model, plus restrictions on the dataflow model in this implementation raise questions of *computability*. Current constraints categorize this project on the low end of abstraction levels possible for algorithm animation, yet effective

as "process visualization". Highly volatile algorithm implementation details are hidden while the fulfillment of the specifications are available for visual confirmation. Most significant in a practical sense, this project outlines manageable goals for iterative enhancements of the *MERGE* algorithm animation project. Credit for this is due to the descriptive and prescriptive properties of the spiral model. Finally, programming paradigms that define the conceptual patterns underneath these emergent fields are described. They control how we think about solutions to new problems, and whether we are able to formulate them at all.

Bibliography

- [Ambler-92] A. Ambler, M. Burnett, E. Zimmerman, "Operational Versus Definitional: A Perspective on Programming Paradigms", *Computer*, (September 1992), pp.28-42.
- [Baecker-86] Baecker, Ronald M., "An Application Overview of Program Visualization", *Computer Graphics*, 20, 4 (July 1986), p.325.
- [Boehm-88] Barry W. Boehm. "A Spiral Model of Software Development and Enhancement", *Computer*, May 1988, pp. 61-72.
- [Booch-91] Grady Booch. *Object-Oriented Design*. Benjamin/Cummings Publishing Co., 1991.
- [Brown-85] M. Brown, R. Sedgewick, "Techniques for Algorithm Animation", *IEEE Software*, Vol. 21, No. 1 (January 1985), pp.28-39.
- [Brown-88] M. Brown, *Algorithm Animation: ACM Distinguished Dissertations; 1987*, The MIT Press, 1988.
- [Brown-92] Marc H. Brown and John Hershberger, "Color and Sound in Computer Animation", *Computer*, May 1988, pp. 61-72.
- [Burnett-92] M. Burnett, R. Hossli, T. Pulliam, B. VanVoorst, X. Yang, "Toward Visual Programming Languages for Steering in Scientific Visualization: a Taxonomy", Computer Science Department Technical Report CS-TR-92-12, Michigan Technological University, December 1992.
- [Clark-92] Jim Clark, "Roots and Branches of 3-D", *Byte*, May 1992, pp. 153-164.
- [Earnshaw-92] Rae A. Earnshaw and Norman Wiseman, *An Introductory Guide to Scientific Visualization*. Springer-Verlag, 1992.
- [Ford-93] Ray Ford, Roland Redmond and Zhenkui Ma. "Aggregation of Image Classification Units for Mapping Large Area", lecture script on 1993 Northwest Arc/Info Users Conference.
- [Ford-94] Ray Ford, Alden Wright, "*A Proposal Submitted in Response to NRA-94-MTPE-02 Information System Technology Applicable to EOSDIS*", Computer Science Department, University of Montana, 1994.

- [Guo-93] Jin Guo. "An Object Oriented Model with Efficient Algorithms for Identifying and Merging Raster Polygons", Masters' Thesis, Department of Computer Science, University of Montana, 1993.
- [IBM-93] IBM Visual Data Explorer *Programmer's Reference*, 4th Ed., SC38-0497-03, *User's Reference*, 1st Ed., SC38-0486-03, *User's Guide*, 4th Ed., SC38-0496-03, IBM Corporation, Fourth Edition, October 1993, SC38-0497-03.
- [Keller-93] Keller and Keller, *Visual Cues: Practical Data Visualization*. IEEE Press, 1993.
- [Lucas-92] B. Lucas, G. Abram, N. Collins, D. Epstein, D. Gresh, K. McAuliffe, "An Architecture for a Scientific Visualization System", *Proceedings of Visualization '92*, Boston, October 1992.
- [Ma-93] Zhenkui Ma and Roland L. Redmond. "Using LANDSAT TM Data and a GIS to Classify and Map Existing Vegetation", paper for Second International Conference/Workshop on Integrating Geographic Information Systems and Environmental Modeling, Breckenridge, Colorado, USA, 1993.
- [Roman-92] Gruia-Catalin Roman *et al*, "Pavane: A System for Declarative Visualization of Concurrent Computations," *J. Visual Languages and Computing*, Vol. 3, No. 2, June 1992, 161-193.
- [Roman-93] Gruia-Catalin Roman and Kenneth C. Cox, "A Taxonomy of Program Visualization Systems", *Computer*, December 1993, pp. 11-24.
- [Shu-88] N.C. Shu, *Visual Programming*, Van Nostrand Reinhold, New York, 1988.
- [Simone-92] Luisa Simone. "The Motion is the Message", *PC Magazine*, August, 1992, pp. 435-467.
- [Stasko-90] J.T. Stasko, "Tango: A Framework and System for Algorithm Animation," *Computer*, Vol. 23, No. 9, Sept. 1990, pp.27-39.
- [Stasko/Patterson-92] J.T. Stasko and C Patterson, "Understanding and Characterizing Software Visualization Systems," *Proc. IEEE Workshop Visual Languages*, IEEE CS Press, Los Alamitos, CA, Order No. 3090, 1992, pp.3-10.
- [Weinstock-86] Neal Weinstock, *Computer Animation*, Addison-Wesley Publishing Co., 1986.

Appendix A: DX Data Model Class Diagram

Appendix B: User-written *DX* Modules

Buildsim.c:

```

#include <iostream.h>
#include <stdlib.h>
int main ( int argc, char* argv[] )
{
    int row, col, num;
    int npline = 10;
    int grid;
    float matrix[256][256];
    int size = atoi(argv[1]);
    if (size < 1 || size > 256) return 1;
    for (row = 0; row < size; row ++)
    {
        for (col = 0; col <= row; col++)
            cin >> matrix[row][col];
    }
    for (row = 0; row < size-1; row++)
    {
        for (col = row+1; col < size; col++)
            matrix[row][col] = matrix[col][row];
    }
    grid = size + 1;
    cout << "# buildsim.dx: size = " << size << endl;
    cout << "object 1 class array type float rank 0 items " << size*size << " data follows" << endl;
    /* put data here " can't use forward offsets in this file in this mode */
    if (size < npline) npline = size;
    for (row = 0; row < size; row ++)
    {
        for (col = 0; col < size; col++)
        {
            num = size*row + col;
            cout << "\t" << matrix[row][col];
            // npline numbers per line
            if ( (num+1) % npline == 0) cout << endl;
        }
    }
    if (num % 10 != 0) cout << endl;

    cout << "attribute \"dep\" string \"connections\"" << endl;
    cout << "object 2 class gridpositions counts " << grid << " " << grid << endl;
    cout << " origin      0      0" << endl;
    cout << " delta        1      0" << endl;
    cout << " delta        0     -1" << endl;
    cout << "attribute \"dep\" string \"positions\"" << endl;
    cout << "object 3 class gridconnections counts " << grid << " " << grid << endl;
    cout << "attribute \"element type\" string \"quads\"" << endl;
    cout << "attribute \"ref\" string \"positions\"" << endl;
    cout << "#" << endl;
    cout << "object \"sim\" class field" << endl;
    cout << "component \"data\" value 1" << endl;
    cout << "component \"positions\" value 2" << endl;
    cout << "component \"connections\" value 3" << endl;
    cout << "attribute \"name\" string \"sim\"" << endl;
    cout << "end" << endl;
}

```

Makefile for CHANGECELL:

```

FILES_CHANGECELL = CHANGECELL.o
BASE = /usr/lpp/dx
CFLAGS = -Dibm6000 -O -I$(BASE)/include
LDFLAGS = \
    -bl:$(BASE)/lib_ibm6000/dxexec.imp \
    -bE:$(BASE)/lib_ibm6000/dxexec.exp \
    -L$(BASE)/lib_ibm6000
LIBS = -IDX -ly -lI -lX11 -lm
OLIBS = -IDXlite -lm
CHANGECELL: $(FILES_CHANGECELL) outboard.o
    $(CC) $(LDFLAGS) $(FILES_CHANGECELL) outboard.o $(OLIBS) -o CHANGECELL
# how to make the outboard main routine
outboard.o: $(BASE)/lib/outboard.c
    $(CC) $(CFLAGS) -DUSERMODULE=m_CHANGECELL -c $(BASE)/lib/outboard.c
# make the user files
userCHANGECELL.c: CHANGECELL.mdf
    mdf-c CHANGECELL.mdf > userCHANGECELL.c

```

CHANGECELL.mdf

```

MODULE CHANGECELL
CATEGORY USER
DESCRIPTION Change the value of a cell
OUTBOARD CHANGECELL;
INPUT data; field; (no default); array of data values: either unsigned bytes or integers
INPUT index; integer list; 0; index into array
INPUT cellvalue; integer list; 0; value to change cell to
OUTPUT result; field; array of changed values

```

CHANGECELL.c: (comments denote sections of code deleted for clarity)

```

/*
 * Automatically generated on "/tmp/CHANGECELL.mb" by DX Module Builder
 */
#include "dx/dx.h"
static Error traverse(Object *, Object *);
static Error doLeaf(Object *, Object *);
/*
 * Declare the interface routine.
 */
int CHANGECELL_worker(
    int, int, int *, float *, float *, int, int, int *,
    int, ubyte *, int, int *, int, int *, int, ubyte *);

Error m_CHANGECELL(Object *in, Object *out)
{ int i;

    /* * Initialize all outputs to NULL */
    out[0] = NULL;

    /* * Error checks: required inputs are verified. */

    /* Parameter "data" is required. */
    if (in[0] == NULL)
    {
        DXSetError(ERROR_MISSING_DATA, "\"data\" must be specified");
        return ERROR;
    }

    /*
     * Parameter "index" not required, but default object could
     * have its default (0) set up here.
     */

    /*
     * Parameter "cellvalue" not required, but default object could
     * have its default (0) set up here.
     */

    /*
     * Since output "result" is structure Field/Group, it initially
     * is a copy of input "data".
     */
    out[0] = DXCopy(in[0], COPY_STRUCTURE);
    if (! out[0])
        goto error;

    /*
     * If in[0] was an array, then no copy is actually made - Copy
     * returns a pointer to the input object. Since this can't be written to
     * we postpone explicitly copying it until the leaf level, when we'll need
     * to be creating writable arrays anyway.
     */
    if (out[0] == in[0])
        out[0] = NULL;

    /*
     * Call the hierarchical object traversal routine
     */
    if (!traverse(in, out))
        goto error;
}

```



```

return OK;

error:
/*
 * On error, any successfully-created outputs are deleted.
 */
for (i = 0; i < 1; i++)
{
    if (in[i] != out[i])
        DXDelete(out[i]);
    out[i] = NULL;
}
return ERROR;
}

static Error
traverse(Object *in, Object *out)
{ switch(DXGetObjectClass(in[0]))
  {
    case CLASS_FIELD:
    case CLASS_ARRAY:
        /* * If we have made it to the leaf level, call the leaf handler. */
        if (! doLeaf(in, out))
            return ERROR;

        return OK;

    case CLASS_GROUP:
        {
            /* ..... recursive group stuff ..... */

            default:
                {
                    DXSetError(ERROR_BAD_CLASS, "encountered in object traversal");
                    return ERROR;
                }
        }
    }
}

static int
doLeaf(Object *in, Object *out)
{
    int i, result;
    Array array;
    Field field;
    Pointer *in_data[3], *out_data[1];
    int in_knt[3], out_knt[1];
    Type type;
    Category category;
    int rank, shape;
    Object attr, src_dependency_attr = NULL;
    char *src_dependency = NULL;
    Object element_type_attr;
    char *element_type;
    /*
     * Regular positions info
     */
    int p_knt = -1, p_dim, *p_counts = NULL;
    float *p_origin = NULL, *p_deltas = NULL;
    /*
     * Regular connections info
     */
}

```

```

int c_knt, c_nv, c_dim, *c_counts = NULL;

/*
 * positions and/or connections are required, so the first must
 * be a field.
 */
if (DXGetObjectClass(in[0]) != CLASS_FIELD)
{
    DXSetError(ERROR_INVALID_DATA,
        "positions and/or connections unavailable in array object");
    goto error;
}
else
{
    field = (Field)in[0];

    if (DXEmptyField(field))
        return OK;

    /*
     * Determine the dependency of the source object's data
     * component.
     */
    src_dependency_attr = DXGetComponentAttribute(field, "data", "dep");
    if (! src_dependency_attr)
    {
        DXSetError(ERROR_MISSING_DATA, "\"data\" data component is missing a dependency attribute");
        goto error;
    }

    /*
     * NO! - this error check commented out ! KK 3/94
     * get the dependency of the data component
     attr = DXGetAttribute((Object)array, "dep");
     if (! attr)
     {
         DXSetError(ERROR_MISSING_DATA, "data component of \"data\" has no dependency");
         goto error;
     }
     */

    /* ..... 25 more error checks ..... */

int CHANGECELL_worker(
    int p_knt, int p_dim, int *p_counts, float *p_origin, float *p_deltas,
    int c_knt, int c_nv, int *c_counts,
    int data_knt, ubyte *data_data,
    int index_knt, int *index_data,
    int cellvalue_knt, int *cellvalue_data,
    int result_knt, ubyte *result_data)
{
    /*
     * The arguments to this routine are:
     *
     * p_knt:        total count of input positions
     * p_dim:        dimensionality of input positions
     * p_counts:     count along each axis of regular positions grid
     * p_origin:     origin of regular positions grid
     * p_deltas:     regular positions delta vectors
     * c_knt:        total count of input connections elements

```

```

* c_nv:          number of vertices per element
* c_counts:     vertex count along each axis of regular positions grid
*
* The following are inputs and therefore are read-only. The default
* values are given and should be used if the knt is 0.
*
* data_knt, data_data: count and pointer for input "data"
*                    no default value given.
* index_knt, index_data: count and pointer for input "index"
*                    non-descriptive default value is "0"
* cellvalue_knt, cellvalue_data: count and pointer for input "cellvalue"
*                    non-descriptive default value is "0"
*
* The following are outputs and therefore are writable.
*
* result_knt, result_data: count and pointer for output "result"
*/

/* * User's code goes here - FINALLY ! */
    int i;
    if (index_knt != cellvalue_knt) {
        DXSetError(ERROR_INVALID_DATA,
            "mismatch in size of index and cellvalue arrays");
        return ERROR;
    }
/* set most of the data */
    for (i = 0; i < c_knt; i++)
        result_data[i] = data_data[i];

/* now set the changed data */
    for (i = 0; i < index_knt; i++) {

        /* check valid index */
        if (index_data[i] >= c_knt) {
            DXSetError(ERROR_INVALID_DATA,
                "index greater than #cells in object");
            return ERROR;
        }

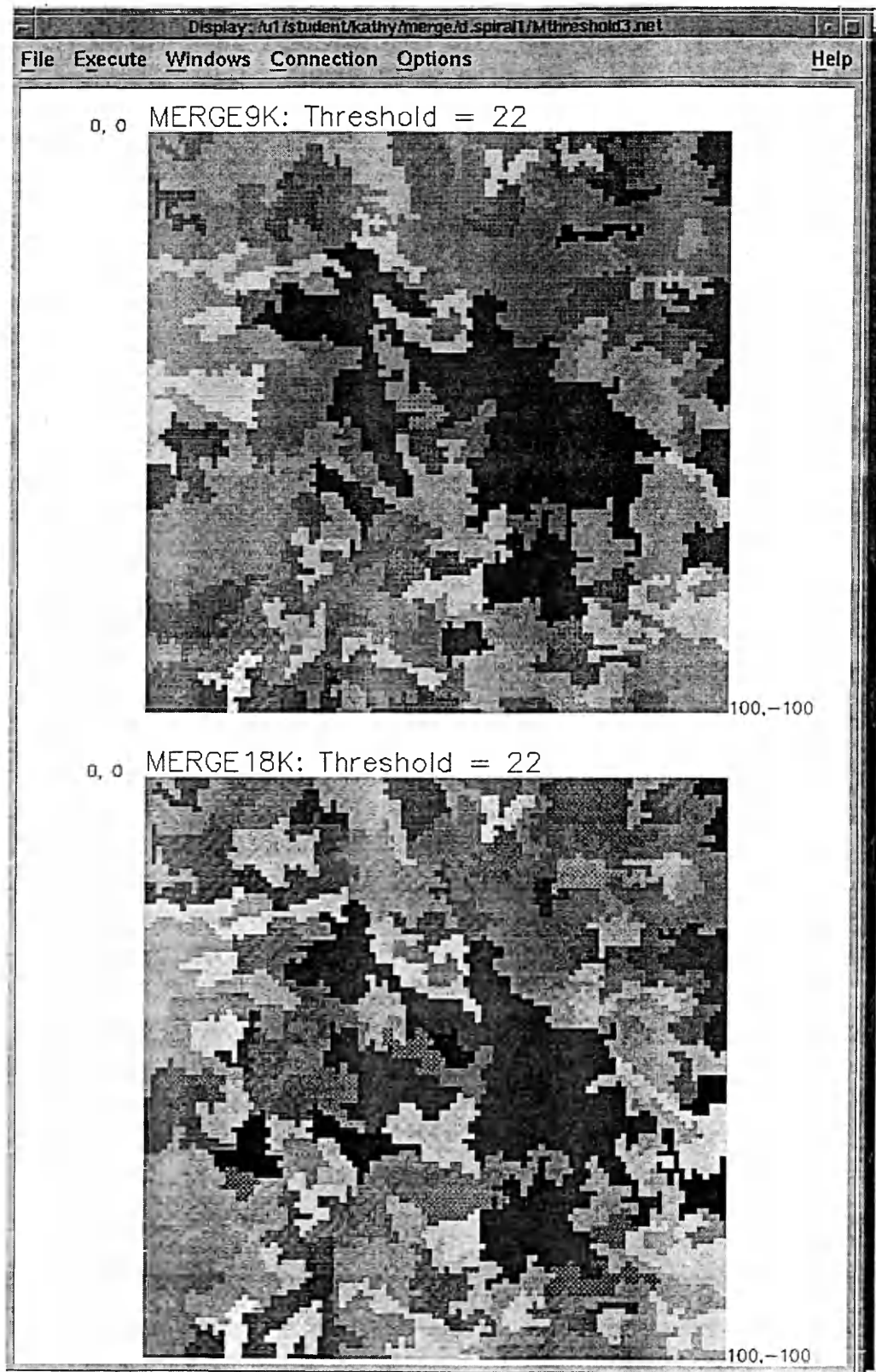
        /* set new value but skip negative indices */
        if (index_data[i] >= 0)
            result_data[index_data[i]] = (ubyte)cellvalue_data[i];
    }

    result_knt = data_knt;

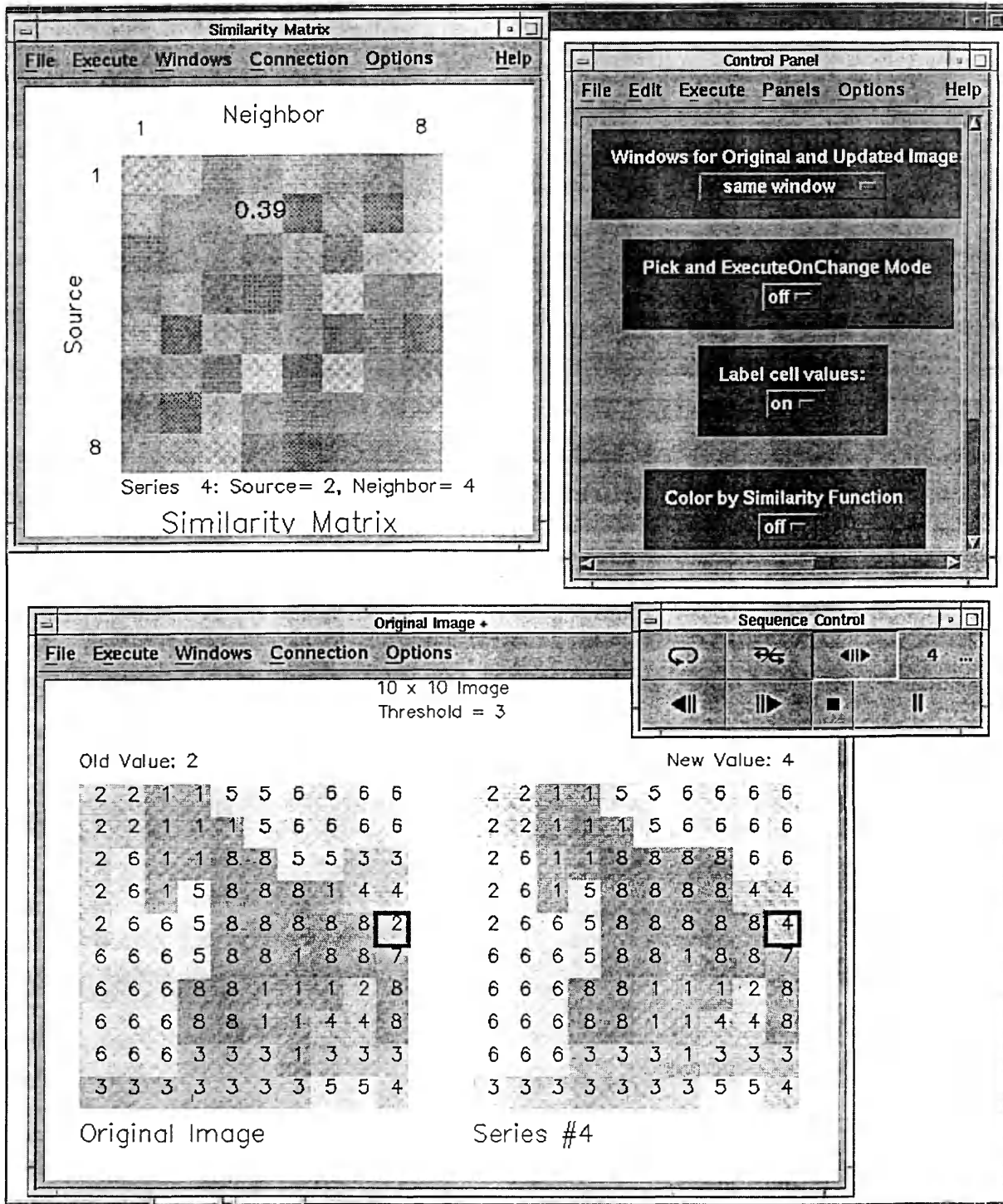
```

Appendix C: Hardcopy Output of Animation Sequences

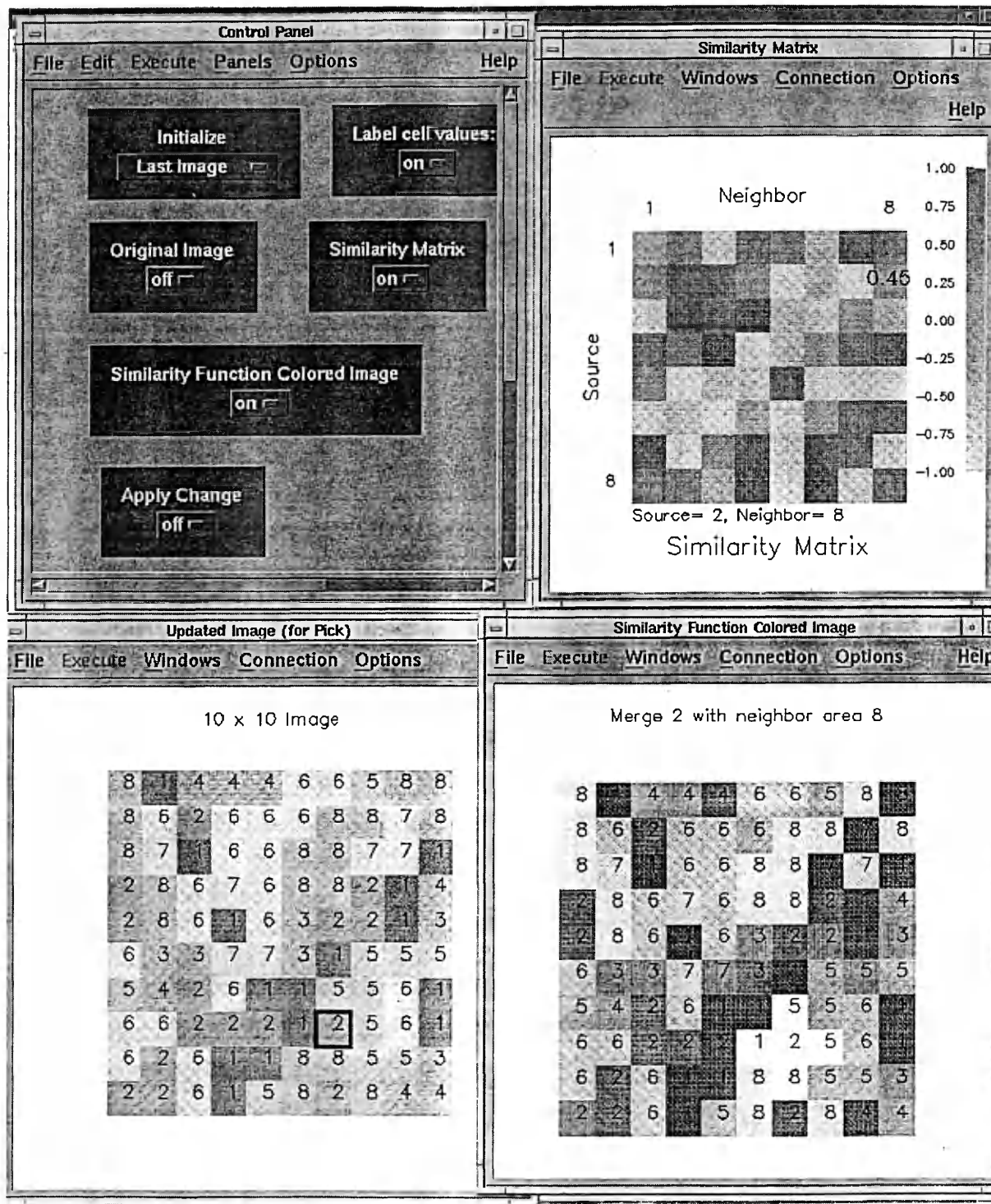
Mthreshold.net - animated series of MERGE output grids merged to different thresholds



Mseries.net - animated series of individual "merges" from merge trace output



Msim.net - interactive simulation of MERGE process



Appendix D: DX Network Descriptions and Instructions

Three dataflow programs are implemented to demonstrate the *MERGE* process. More detailed instructions are available in the online help facility under comments for each visual program. These instructions should be stable with respect to *DX* versions but are to be updated with changes to the visual programs.

DX is initialized with the following command line:

```
dx -mdf ALL.mdf -directory /u1/student/kathy/d.demos -macro .
```

Mthreshold.net - *animated series of MERGE output grids merged to different thresholds*
(dynamic, passive macro view showing different versions side-by-side in single synchronized window)

Start the program by selecting the **sequencer** from the **execute** menu, click on the **ellipses**, and set next to 0. Close the **ellipses** box and select the forward play button of the **sequencer**.

Mseries.net - *animated series of individual "merges" from merge trace output*
(dynamic, passive visualization with highlighted activity, demonstrates processing order & "cascaded effects" with multiple windows and minimum bandwidth communication with off-line process)

Start the program with **all control panels** open and select the **sequencer**.

Msim.net - *interactive simulation of MERGE process*
(dynamic, highly interactive simulation of source and neighbor cell selection, multiple windows, demonstrates missing attributes of polygon and neighbor lists)

Start the program with **all control panels** open. To apply the changes consistently, *Initialize* must be set to *Original Image* and *Apply Changes* to *Off*. Select **Execute Once** and enter <control-i> with the cursor on the updated image window. Then set *Initialize* to *Last Image* and *Apply Changes* to *On*, and select **Execute on change**.

The user picks a source or target cell to be merged. It is highlighted in white and its neighbors are colored according to the similarity function. The target cell is merged with the neighbor polygon with the highest similarity value.