

A Real-Time Graphics Conferencing System

by

David Joseph Mauro

M.Sc., University of British Columbia, 1979

B.Sc., University of British Columbia, 1976

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the School

of

Computing Science

© David Joseph Mauro 1988

SIMON FRASER UNIVERSITY

April, 1988

**All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without permission of the author.**

APPROVAL

Name : David J. Mauro

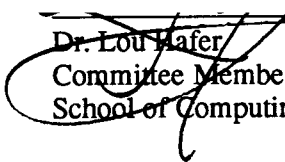
Degree : Master of Science

Title of Thesis : A Real-Time Graphics Conferencing System

Examining Committee :

Chairman : Dr. B. K. Bhattacharya

Dr. Tiko Kameda
Senior Supervisor
School of Computing Science, SFU

Dr. Lou Mafer
Committee Member
School of Computing Science, SFU

Dr. Stella Atkins
External Examiner
School of Computing Science, SFU

Date Approved : April 20, 1988

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

A Real-Time Graphics Conferencing System.

Author:

[Signature]
(signature)

DAVID MAURO
(name)

Apr 21/88
(date)

Abstract

The exchange of ideas among a group of people traditionally takes place in an environment where face-to-face interaction is possible - that is, in the same place at the same time. With electronic conferencing systems the reliance on spatial simultaneity for meetings disappears. We study electronic conferencing in the context of real-time multiple-user interactive environments which use graphic images as the medium of information exchange.

We introduce two basic principles that are used to develop a model conferencing system. The first principle states that during an electronic meeting all conference participants should see the same graphics image at the same time. The second principle states that the performance, functionality and interface of a single-user system should not be compromised when that system is integrated into a conferencing system with multiple participants.

With these principles in mind we develop a model for a two-user, real-time graphics conferencing system. We focus our development on efficient graphics storage structures and concurrency control methods. We test the feasibility and applicability of our model by implementing a prototype. The application and analysis of the prototype implementation point out the strengths and weaknesses of our model, and indicate the trade-offs and relaxations of our basic principles which are necessary in order to achieve satisfactory performance.

*To James,
who made my life as a graduate student so interesting by being born
on the first day of the first semester of my graduate program.*

Acknowledgements

I would like to thank my thesis supervisor, Tiko Kameda, for his insight, his incisiveness and his patience. I would also like to thank my other committee member, Lou Hafer, for his suggestions and for his support of the project, and the external examiner, Stella Atkins, for her comments and her suggestions. Finally, I would like to thank my wife, Carol, for her support and her energy throughout graduate school and especially for the duration of the thesis work.

Table of Contents

Abstract	iii
Dedication	iv
Acknowledgements	v
Table of Contents	vi
List of Figures	ix
CHAPTER 1. Background and Concepts	1
1.1 Basic Conferencing Systems	2
1.1.1 Foundational Abstractions	3
1.1.2 The CIS Model	5
1.1.3 Process Organization	6
1.2 The Common Information Space	8
1.3 Protocols	10
1.3.1 Image Data Communication	11
1.3.2 Image Synchronization	12
1.4 The User Interface	13
CHAPTER 2. Modelling the Graphics Common Information Space	15
2.1 Image Representation	15
2.2 The Single-User Basic Action Cycle	17
2.3 Communication Channels	19
2.3.1 Channel 1 Communications and the IDP	20
2.3.2 Channel 2 Communications and the ISP	20
2.4 The Multiple-User Basic Action Cycle	21
2.4.1 The Completion Phase	24
2.4.2 Deadlock Prevention	26

2.5 Local Division of Labour	27
2.5.1 Local Event Flow	30
2.6 Summary of the CIS Model	30
CHAPTER 3. Implementing the Concurrent Imaging System	32
3.1 CIS Architecture	33
3.2 CIS Data Structures	35
3.2.1 The Display Structure	35
3.2.2 The Itree Algorithms	40
3.2.3 Object Selection and Locking	45
3.2.4 Data Structure Management	48
3.3 CIS Communications Protocols	49
3.3.1 Image Data Protocol	50
3.3.1.1 Distributed State Information	51
3.3.2 Image Synchronization Protocol	53
3.3.2.1 Examples	53
3.3.3 CIS Connection Protocols	60
3.3.3.1 Maintaining the Connection	61
3.4 CIS User Interface	63
3.4.1 Graphical Functionality	63
3.4.1.1 Objects and Operations	63
3.4.1.2 Attributes	66
3.4.1.3 Functionality and Distribution	66
3.4.2 File Maintenance	68
3.4.2.1 File Types	68
CHAPTER 4. Observations, Evaluation and Further Research	71
4.1 Itree Motivation, Rationalization and Analysis	71

4.1.1 Linked Lists and the Itree Structure	72
4.1.2 Itree Pathologies	75
4.1.3 Improving Itree Operations	79
4.1.4 Other Display Structures	81
4.1.5 Itree Consistency	84
4.2 Multiple Images	87
4.3 Non-Blocking Completion	89
4.4 Distributing IDP State Information	90
4.4.1 Maintaining Remote State	91
4.5 Connection and the Distributed Name Server	92
4.5.1 The Advantages of a Distributed Name Server	93
4.6 Long Haul Connections	94
4.7 Multiple Participant Conferences	95
CHAPTER 5. Conclusions	101
6. References	105

List of Figures

Figure 1. The Common Information Space Abstraction	1
Figure 2. What You See Is What I See	4
Figure 3. Shared Program Design	6
Figure 4. Multi-User Application	7
Figure 5. Distributed Control Model	8
Figure 6. CIS as a Single-User Abstraction	9
Figure 7. The Single-User Basic Action Cycle	18
Figure 8. Two_User Basic Action Cycle - Client	21
Figure 9. Two_User Basic Action Cycle - Server	22
Figure 10. Basic Action Cycle with Two Users	23
Figure 11. Insertion Commit Action Subphase	26
Figure 12. Combined Action and Sync. Request Queues	27
Figure 13. The Local and Remote BAC	28
Figure 14. Local Division of Labour	29
Figure 15. Modelling the CIS	31
Figure 16. Single-User Architecture	33
Figure 17. Two-User Architecture	34
Figure 18. The Itree Structure	37
Figure 19. Example Itrees	38
Figure 20. The Insertion Algorithm	41
Figure 21. The Deletion Algorithm	43
Figure 22. The Hide Operation	45
Figure 23. Communication Channels Between Subwindows	50
Figure 24. Key to Message Passing Diagrams	54
Figure 25. Message Passing During the Selection BAC	55

Figure 26. The Selection Operation MPD	56
Figure 27. The Insertion Operation MPD	58
Figure 28. MPD for the Image Synchronization Action	60
Figure 29. Maintaining the Connection	62
Figure 30. The CIS Display	64
Figure 31. An Impossible Image to Draw?	75
Figure 32. Images Which Exhibit Linked-List Behaviour	76
Figure 33. Inconsistencies in an Enhanced Itree	86

1

Background and Concepts

A graphics window shared among two or more users is an example of a shared or common information space. Other such spaces include the common auditory space of a telephone connection; the (semi) common textual space of the UNIX system program 'talk'; conventional computer conferencing systems [BaN81] or the shared knowledge space of AI-oriented black-board systems. We distinguish a common information space from other types of shared data spaces (i.e., databases, bulletin boards, etc.) by including only real-time multiple-user interactive graphics environments. Such a system will be referred to as a *Common Information Space* [Tho84] (CIS). The pictorial abstraction of a CIS is shown in Figure 1.

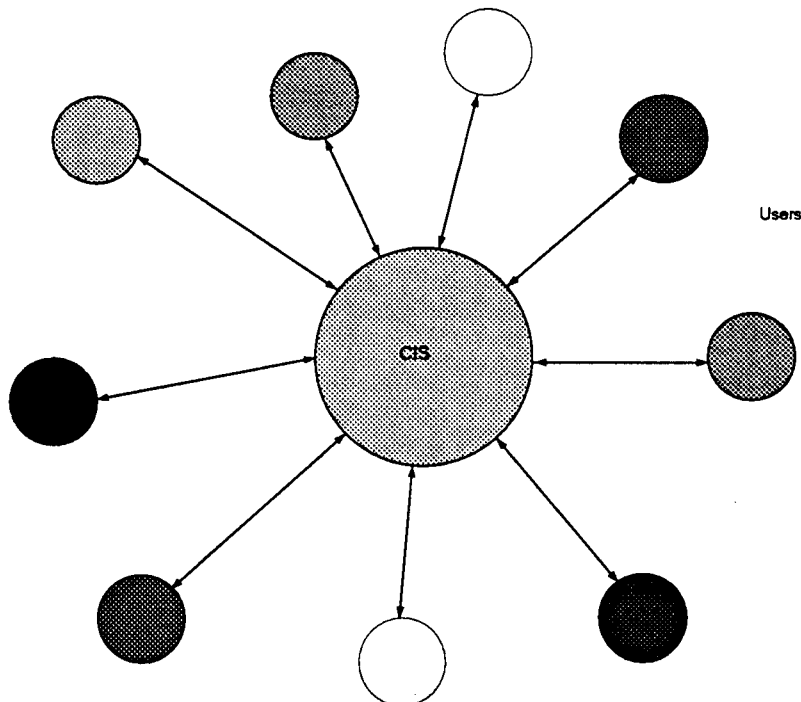


Figure 1. *The Common Information Space Abstraction*

When the label 'Common Information Space' is too generic, we will use 'CIS' to refer to the more application-specific label *Concurrent Imaging System*. We will primarily be concerned with the concept of a CIS in the context of two-user, real-time conferencing with a graphics interface, over a Local Area Network (LAN). As an extension to this we will discuss the additional problems encountered with multiple users possibly conducting sessions over a long haul network.

1.1 Basic Conferencing Systems

The principal use for real-time electronic conferencing systems is to allow multiple contributors to a project concurrent access to the project representation or model, with the idea that many heads are better than one. Conferencing systems have been designed that are meant to be used by people sitting in the same room at the same time [SFB87], or by people separated by any distance and time [SaG85]. In conventional conferencing systems participants communicate with each other by manipulating a common textual data space. The text files are usually stored in a central location and access to them is mutually exclusive. One of the major benefits of such a system is the temporal freedom it allows its participants; users may access the files whenever it is convenient. An added benefit is that conference control and data consistency measures are relatively simple to design and implement.

Real-time interactive systems, on the other hand, require the on-line presence of all participants in order to initiate and conduct a conference. This enables immediate feedback from other users to a particular user's input. However, the conference control and data consistency problems become more complex. Some systems rely on face-to-face meetings or audio connections in order to synchronize activity within the CIS; [SFB87] talk about using a "vocal lock", while [SaG85] consider minimizing conflicts by having participants "negotiate externally by voice".

While the immediacy of real-time conferencing enhances the brainstorming process, the

loss of temporal freedom may impose unwieldy scheduling restrictions upon groups with many participants. While we will point out the strong and weak points of our design, the point of this thesis is not to argue electronic conferencing vis-a-vis real-time vs. conventional, interactive vs. batch, etc. The technological and social advantages and disadvantages of such systems are discussed at length in *The Network Nation, Human Communication via Computer* [HiT78] and *Electronic Meetings: Technical Alternatives and Social Choices* [JVS79]. We will consider some of the basic principles laid out in the literature and use these principles to formulate goals upon which we will base our model of the CIS. Using the specificities of a real-time graphics-oriented CIS we will refine the model. Finally, we will lay out a set of design guidelines which will be applied to the model in order to arrive at an implementation.

1.1.1 Foundational Abstractions

To aid in the characterization of conferencing systems we need to lay out some basic goals which we would like to apply to these systems. We will summarize the concepts we have adopted to apply to the design of the CIS. In [SBF86] the concept of *WYSIWIS* (What You See Is What I See - pronounced "whizzy whiz") is introduced. *WYSIWIS* is a "foundational abstraction for multi-user interfaces that expresses many of the characteristics of a chalkboard in face-to-face meetings." This concept implies that everyone present in a meeting will see the same thing at all times, and will see where everyone else is pointing. It was developed as a starting point for the design of computer-supported collaborative tools that manifested themselves in an experimental meeting room called *Colab*, set up at the Intelligent Systems Laboratory in Xerox PARC. There are four dimensions to *WYSIWIS*: display space, time of display, subgroup population and congruence of view. The above article discusses various meeting tools and the relaxations on the strict sense of *WYSIWIS*, w.r.t. these dimensions, that were necessary in order to realize the potential of the designs and to maximize the functionality of the implementations. Figure 2 illus-

trates the WYSIWIS abstraction in the context of the CIS.

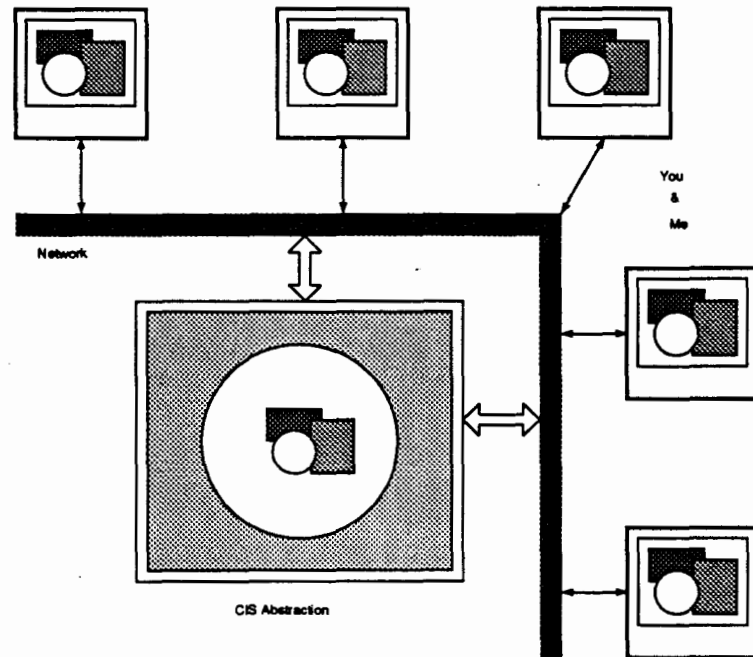


Figure 2. *What You See Is What I See*

Another basic design tenet is that *multi-user functionality* for a group of participants should appear the same as single-user functionality to the individual. That is, using the graphics CIS as a conferencing tool as opposed to a stand-alone tool, should not degrade its functionality. Furthermore, CIS actions should be the same as, or at least resemble, the actions which have become accepted on conventional tools of a similar nature. Stefik, et al., argue that "...computer media must accommodate the needs of a group as well as the needs of an individual". One of the principal problems that we expect will arise as this design idea is implemented is that individual response time will be degraded as group activity increases. One of our goals will be to determine ways in which group needs may be accommodated without affecting the creative input of the individual.

The goals of WYSIWIS and multi-user functionality stated above lead us to observe that any communications design should not make it possible for one user to block indefinitely while

waiting for another user's input. Otherwise WYSIWIS would be violated because much could be happening to the CIS while a user is blocked. And multi-user functionality would clearly be violated because we never block in a single-user system. We will call this the *Primary Communication Principle (PCP)*.

1.1.2 The CIS Model

With the foundational abstractions of WYSIWIS and multi-user functionality in mind we will consider the framework which we will use in the design of our model. In [Sar84] it is assumed that in a real-time conferencing system participants will be interested in manipulating a collection of logical *objects*, and that there will be an editor available for displaying and modifying the *state* of an object. These basic concepts, when thought of in the context of a real-time multi-user interactive environment, generate the following issues in computer-supported conference system design:

- (1) Shared vs. individual views
- (2) Concurrency control
- (3) User interface
- (4) Constraints on real-time conferencing design
- (5) Data manipulation (file access control)
- (6) Access control (access rights to shared objects)

When considering the CIS design we will concentrate on issues 1, 2, and 3. These issues are discussed in more detail in sections 1.2, 1.3 and 1.4 respectively. We will also be concerned with issue 4, but in the context of WYSIWIS as mentioned above. We will not discuss issues 5 and 6. These issues are concerned with conference management - who may join a conference, the object/display page/file access rights of a participant, file locations and so on. These problems need not affect the design and implementation of the CIS. To include them in the thesis would greatly increase its size without shedding much light on the way in which our basic goals are utilized.

1.1.3 Process Organization

Sarin and Greif [SaG85] consider two basic models for process organization: sharing existing programs and multi-user applications. The first approach operates on the assumption that most interactive applications communicate with the user via input/output character streams. See Figure 3.

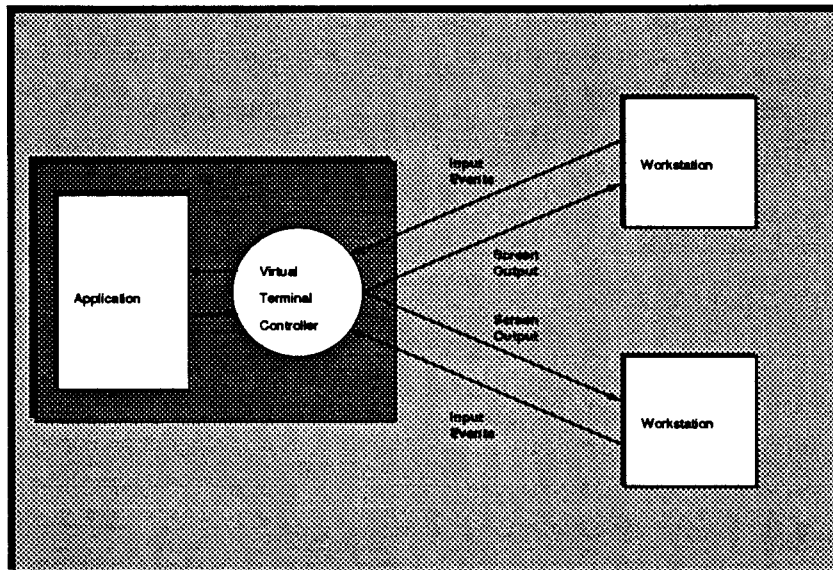


Figure 3. *Shared Program Design*

Then existing applications may be shared without modification by introducing a *virtual terminal controller*. This controller intercepts all user inputs and, based on which user currently has control of the shared space ("has the floor"), sends that user's input to the application for processing. Application output is returned to the controller for distribution to all the users. This approach has been favoured by [AGN87]. A variant of this approach has each user running an identical instance of the program and sending its input to each of the other users to be processed in parallel.

We can see from the description of shared program design that what the controller does is provide mostly transport level support between users. This model does not take into account the identity of different participants in a conference. A minimal amount of session support is pro-

vided via the controller in the form of mutually exclusive access to the data by whatever user has the floor. The "floor control" policy has also been adopted by [For85].

[Lan86] has generalized this concept in order to "demonstrate the feasibility of implementing conferencing facilities with no (or few) modifications to the existing software environment". This system employs a *conference manager* to provide floor control and "other necessary synchronization functions". However, the paper does not specify any other necessary synchronization functions.

In the multi-user application approach a new application program is written for each new need (See Figure 4).

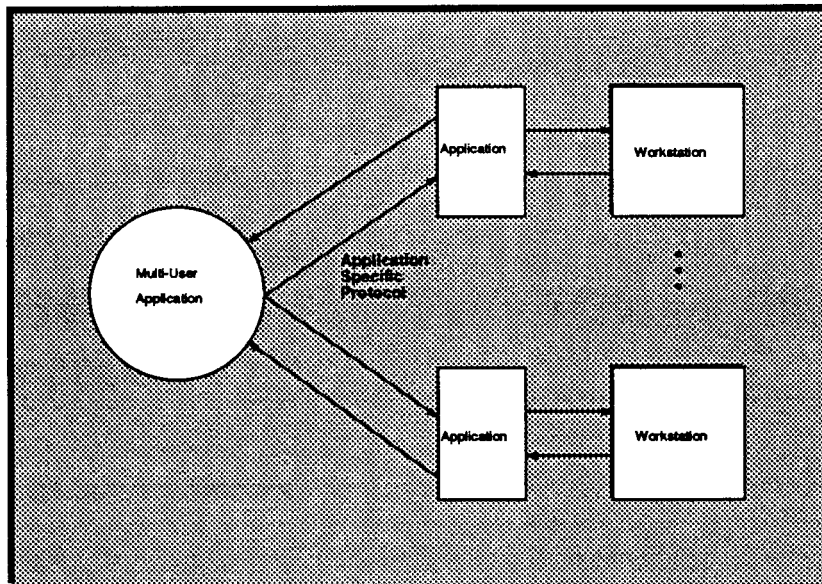


Figure 4. *Multi-User Application*

This model is closest to the approach we will take. Sarin and Greif [SaG85], however, still allude to the presence of a central controller to manage conference control and application-level protocols. We feel that a decentralized approach will lead to a more flexible conferencing system.

In our model the control will be distributed, with no single user being distinguished in any manner (See Figure 5). We will assume a fully replicated architecture with copies of the data at each site. Events generated by one user will be distributed to all other users 'simultaneously'.

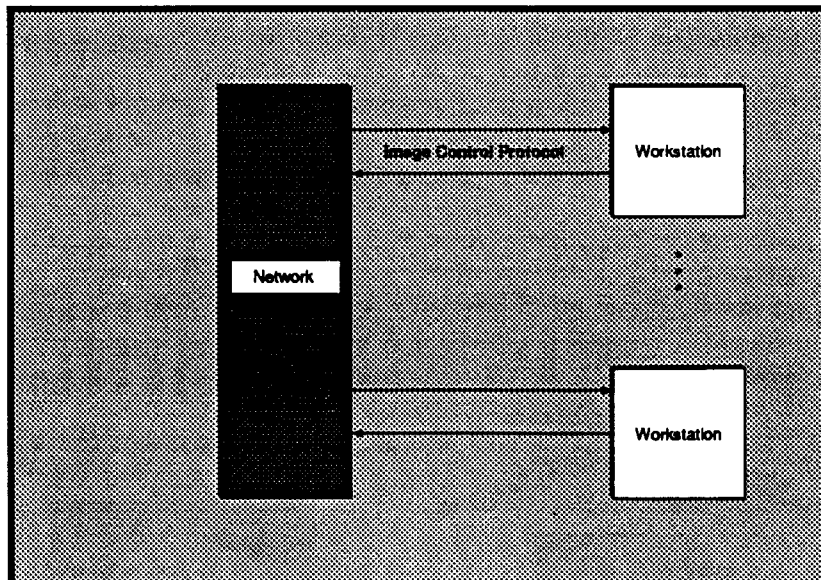


Figure 5. Distributed Control Model

Object contention, deadlock and other concurrency issues will be dealt with using distributed algorithms as opposed to centralized ones. Some of the algorithms will be based on conventional distributed algorithms, while others will be developed to deal with some of the special real-time graphics problems that arise.

1.2 The Common Information Space

We imagine the Common Information Space to be a single logical entity shared among two or more users. The manifestation of this entity is a graphical image appearing simultaneously on each user's output device. (See Figure 2.) An image is a collection of objects which form a graphical data base. This is a distributed, replicated data base. The distributed collection of graphical data objects is called the *distributed image state*. Each user has the ability to access this image; to add to, delete from or modify the objects of this data base. We may use the model of a blackboard (not an AI blackboard system), one or more pieces of chalk and many people participating in the manipulation of board contents, to describe the CIS.

A less concrete model would be to consider the CIS as an abstract data type combining

the distributed image state with the network state. A primitive operation on this abstraction would be a combination of primitive data manipulation operations and primitive network commands. For example, a primitive CIS operation might be a primitive object-insertion operation together with a send/receive pair. A conference would consist of the concurrent, interactive, distributed manipulation of the CIS abstraction using the image manipulation operations and network send/receive pairs. This model allows us to view the CIS in the context of a single-user imaging system as seen in Figure 6.

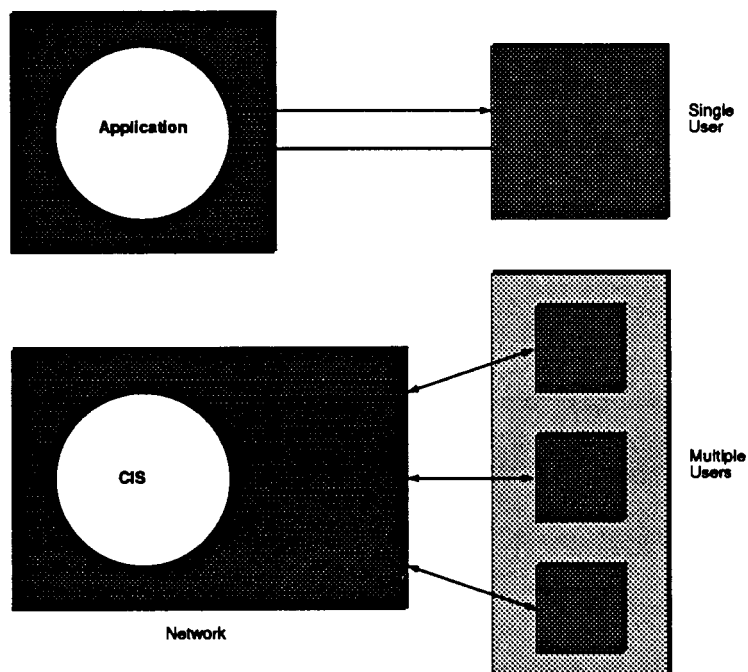


Figure 6. *CIS as a Single-User Abstraction*

In the blackboard model of the CIS each participant should be aware of what each of the other participants is doing at any time. The granularity of this awareness is one of the areas which we have studied. That is, while a participant must be aware of the end result of another participant's actions, it is not clear how much intermediate action should be shared in order to adhere to the goal of WYSIWIS. At one end of the scale every action of every participant would appear at every site. This would give everyone a completely up-to-date idea of what everyone

else was doing. On the other extreme, only the end result of object insertion, deletion or modification would be reflected at every site. In the former case, duplication of participant effort would be avoided and the present area of interest and activity of each participant would be known to others. The cost would be high communications overhead and perhaps a somewhat distracting display. Whether or not the extra communication would degrade system response, and the effect on the user of a multiple concurrent display access are questions that have been studied. These costs would not be incurred in the latter case, but their absence may lead to misunderstanding about the state of the CIS. Thus, like many distributed computing problems, the optimal amount of state exchange among sites is also of considerable interest in the CIS. Furthermore, because of the real-time, interactive nature of the CIS, the question of precisely *when* to exchange state information is also important.

1.3 Protocols

The problem of state exchange involves not only the extent of the state information, or state event information, but how it is represented internally, how it is to be represented on the network and how it is processed globally. We will find that the semantics of control and communication in the CIS will be complicated because of the real-time interactive nature of the system. The state information exchanged we will call **Image Data Communication**. The distributed processing of this information we will call **Image Synchronization**.

In order to reduce the amount of message processing that will need to be done, we will set up multiple communications channels. Messages arriving on a particular channel will have a particular interpretation. The protocols will be specialized to minimize the interpretation needed to derive the message context thereby, hopefully, reducing the overall response time. This is in contrast to the graphics protocols of [SpT74] and [LaN84] which are designed to be general-purpose virtual graphics terminal protocols. Such protocols are often implemented within the operating system kernel instead of on top of it.

State events may occur at many levels: hardware input events, image control events and image synchronization events. One of the problems we will be addressing is that of attempting to classify the event types, propose a general set of events for each type and then show how these events may be 'shared' among the participants, in the context of a real-time conferencing system, in order to present optimal solutions to the above questions.

1.3.1 Image Data Communication

Like most data base systems, update access to data objects must be exclusive and the data base must be kept consistent. To this end a protocol for image sharing, remote image manipulation and recovery from site failure must be devised.

Using the blackboard model, we see that each participant (standing in front of the blackboard) is aware at all times of who is currently accessing the image and how the image is being manipulated. We would like this to be the case in the CIS also (WYSIWIS goal). Thus, the image communication protocol must induce the actions of each participant on the screens of the other participants. That is, the *effect* of the input stream generated by every participant should be simulated on the screens of the others in such a way that the intermediate and end results would be as if each participant had generated the image locally.

The question of *what* information is to be distributed will depend upon *when* in the image manipulation cycle we decide to initiate the transfer. We may decide to transfer state at the beginning of, during and at the end of an action. In this case, we have to transfer input events at all times, and possibly other information at the end of an action. (Note that we must delimit what we mean by 'action'.) If we transfer only at the end of the cycle, for example, then we would have to transfer system output (image internal representations).

Another factor that will affect our protocol is multiple images or pages. If we allow multiple display pages to exist concurrently within the system then we must accept that different participants may be viewing different pages at any one time. Note that while this violates the con-

cept of WYSIWIS, it extends the functionality of the CIS. This is dealt with in [SFB87] by introducing stampsheets. A stampsheet is a miniature version of a window that permits some feedback when that window (page) is being accessed by a group or an individual. If two participants are in fact viewing different pages then either the type of information transferred or the interpretation put on that information will be different from that in the case where the same page is being viewed by both participants. Put another way, a participant does not need to send any screen update information to a participant that is viewing a different page. Only the final result of the operation would need to be sent. However, this means that each participant would have to be aware of which page every other participant was currently viewing. This in turn would require (i) state exchange every time a different page was selected for viewing, (ii) every participant maintain state about the current view of every other participant. Alternatively, we could ignore the problem of page state exchange and maintenance, and let the recipient of another participant's input events decide how to process the information. These problems fall into the category of 'further research'.

1.3.2 Image Synchronization

We would like the CIS to be as flexible as possible. Independent (i.e., local only) as well as concurrent image manipulation, image overlaying operations (and late joining of the session in conferences with more than two participants) should be supported. With these kinds of options available it is not hard to see that images at different locations may often enter an inconsistent state. Our aims are

- (1) to determine to what degree or for how long the system will tolerate inconsistent image states
- (2) to minimize the amount of time the CIS is in such a state.

Therefore, we will need a protocol for dynamically synchronizing images or portions of images. This protocol will be tied into the internal representation, or display structure (DS), of the screen image. We will develop one or more hierarchical data structures in which to store and

manipulate images created interactively in the CIS.

In a CIS with two or more users the control mechanism would also have to deal with the problems of starvation and fairness: how do we ensure that everyone who desires access to particular objects eventually gets it, and gets it before others who have made later requests? Another question is that of image synchronization granularity. That is, assuming we have a locking mechanism, how much of the image may be locked by one lock? How many different locks may be held? What is the effect of subtree locking? What happens when a lock is released? These questions must be answered in the context of real-time conferencing. Other questions that must be addressed include the problems of deadlock and the semantics of failure during image synchronization. The latter falls under the banner of future research.

1.4 The User Interface

The user interface to the CIS also presents some interesting problems. In addition to the actual use of the graphics tool, there are the questions of connection and termination protocols, file access management and conference control.

The CIS concept realized by effective implementation of the communication protocols discussed above must be presented to the user in an intuitive interface. The blackboard model should comfortably overlay the logical system. The abstract model should effectively underlie the physical implementation. Combining interactive graphics, distributed processing and data base management should appear as simple, to the user, as using a favourite text-processing or picture-drawing package. In fact, the nature of the graphics manipulation commands should be identical when implemented in a real-time distributed conferencing system, as when implemented in a stand-alone single-user system (goal of single-user vs. multi-user functionality). To this end we will often consider the user interface in the context of the abstract model of the CIS discussed above. Additionally, the commands for conference control should reflect a natural progression for conference initiation, instantiation and termination. The underlying connection and commun-

ication protocols should be transparent to the user.

We note that the user interface subsumes the problems of communication protocols and imaging capabilities in the sense that these problems are only the physical manifestations of the underlying model, which in turn is the realization of the abstraction that defines the basic system. Thus, the user interface should provide a smooth transition from CIS invocation, to image definition and manipulation to image synchronization to session termination.

In Chapter 2 we will define the elements which comprise our model of the Common Information Space. The CIS model will be developed 'vertically' from the single-user point of view, and 'horizontally' from the two-user point of view. We will take the model of the CIS and generate an implementation of the Concurrent Imaging System in Chapter 3. In the implementation we will demonstrate how we maintain the basic goals outlined above. Chapter 4 will provide a summary of the observations, results and evaluations derived from the development and operation of the working prototype. We will discuss the conclusions we arrived at with respect to the goals set out in Chapter 1. We will include in this chapter a discussion of directions for future research. In Chapter 5 we provide a summary of the research.

2

Modelling the Graphics Common Information Space

A model of the CIS in the context of real-time interactive graphics must reflect the goals laid out in the previous chapter; that is, WYSIWIS and multi-user functionality. As we have seen we may regard the distributed CIS as a single-user system by viewing the network combined with the image state as a single data abstraction. With this in mind we will first consider single-user interfaces and then see how they may be adapted to two-user distributed interfaces. (We will generalize the model in later chapters.) This will be done by breaking the functional aspects of our system down into the following general categories:

- local image representation
- local creation and manipulation control
- communications channels
- distributed imaging

2.1 Image Representation

There are two facets to the representation of an image: what appears on the terminal output device, the *visual representation* and how the image is stored by the application, the *internal representation*. We will also refer to these as the *outer image* or *state* and the *inner image* or *state*, respectively. The terms image and state will be used interchangeably. We will also call the outer image the *display*. The visual representation in a single-user system is a bit map that appears on the terminal screen. This image is stored and refreshed by the hardware in a device dependent manner. The refresh rate and device dependent storage do not concern us. The internal representation, called the Display Structure (DS), is a collection of *objects*, which may be used to update the hardware dependent internal storage, eventually causing the outer image to be

refreshed. We will say no more about the DS here except that each of its objects are uniquely identifiable and are of arbitrary size. (See section 3.2.)

While, in the best case, the visual representation should reflect the internal representation, and no more, disparities between the two may arise. The fundamental reason for this disparity is that due to the fact that display refreshing actions are often user-controlled, the inner image may be updated with no changes reflected immediately on the outer image. On the other hand, the display may go through temporary changes while the inner image remains constant. In the single-user system these disparities are due to outer image drawing or undrawing operations during the creation of a new object or the modification of an existing object that result in unwanted pixels that remain, or wanted pixels that disappear. For example, the deletion of an object may leave a 'hole' in the image. This is easily remedied by refreshing the display from the internal representation. Both of these cases lead to inconsistencies between visual and internal representations. One of our design decisions will concern *how* and *when* the inner and outer images are synchronized. While this may seem like a purely graphical-interface related problem we will see that graphical functionality, in general, has to be re-evaluated in a distributed environment.

In the multiple-user case, disparities may also arise among multiple visual representations and among multiple internal representations. Furthermore, the disparities between visual and internal representations on a single machine may be more significant in the multiple-user system than in the single-user system, because of multiple input sources at each of the inner and outer state levels. The goal of WYSIWIS demands that visual representations be consistent. Since visual representations may be refreshed from internal representations, WYSIWIS indirectly implies consistent inner states as well. *How* and *when* to synchronize representations across multiple users will have a great effect on the CIS model. These problems are addressed via the Image Synchronization Protocol.

We stated earlier that one of our goals was to determine to what extent the system will tolerate inconsistent image states. That is, to what extent or under what conditions can

WYSIWIS be relaxed? We will start from the position that we want WYSIWIS to be strictly observed. From the relation between the inner and outer images we can see that in order for WYSIWIS to be preserved the inner images must remain consistent among users. Thus, we make the initial assumption that inner state inconsistencies in a steady state will not be tolerated.

In our model we will set up a communication channel between inner images and another between outer images. We can view the two-user visual representation as the combined outer state of the individual users plus the contents of the communication channel. Similarly for the internal representation.

2.2 The Single-User Basic Action Cycle

The graphical functionality of the CIS is extremely important if the conferencing system is to be useful. From the single-user point of view the CIS is a graphical editing environment in which we would like to be able to create and modify a wide range of graphical objects. Associated with each object are a set of display coordinates which define the object's size, attitude and position on the display, and a set of *attributes*, such as line thickness, fill type, dashed line pattern, etc., which further refine the definition of the object. Modifying or 'editing' the DS is a matter of selecting an object, changing its coordinates and/or attributes and then regenerating the object in terms of those changes.

Generating or manipulating an object in an interactive graphics environment involves choosing a set of attributes for that object and then determining its size, attitude and position in the display. The *events* which are generated by the interactive interface are divided into two basic types. The first type is the *context defining event* (CDE). Attribute selection and object-type (line, rectangle, etc.) selection are context defining events. CDE's do not have an immediate effect on the display. Events which subsequently generate or manipulate the object (or the entire DS) are called *action events* (AE's). Low-level device input creates action events, which have no meaning outside of the context defined by the CDE's. These events are processed in the context

of the graphics environment set up by the CDE's.

An *action* or *operation* is defined to be the creation or destruction of an object, or the modification of an existing object. We will use the terms action and operation interchangeably. We will see that the precise modelling of an action takes on a deeper meaning in a distributed environment.

Carrying out an action consists of generating action events which change the state of the object. These events are divided into three phases which constitute the Basic Action Cycle (BAC). As shown in Figure 7, the three phases of the BAC in a single-user system are called *initiation*, *articulation* and *completion*.

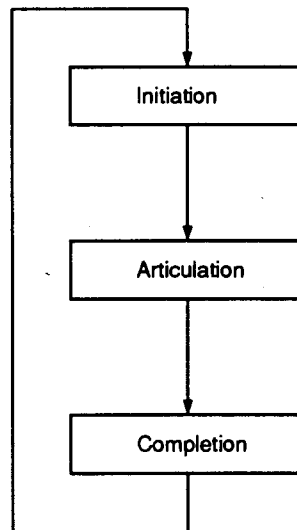


Figure 7. *The Single-User Basic Action Cycle*

Events generated during the initiation and articulation phases are called *initiation* and *articulation events*, respectively. Initiation events determine the *origin* or *anchor* coordinates of the object. By origin we mean some reference point upon which other points which define the object depend. For example, the origin of a rectangle may be the coordinates of the upper left hand corner of the rectangle. Articulation events determine the object's *terminal* coordinates. In the case of a rectangle, if we have defined one corner then articulation events would generate a sequence of coordinates, over time, which represent the opposite corner. The last coordinates in

the sequence would become the terminal coordinates. Until now, all events have affected only the visual representation. Once the object's anchor and terminal coordinates are known the completion phase of the cycle is invoked. During this phase the object may take on the relevant attributes, be internally realized, inserted into the DS and visually represented with all its attributes. The completion phase always affects the internal representation and may possibly affect the visual representation as well.

We may consider the initiation and articulation phases together with the visual representation as a data abstraction. Similarly we may consider the completion phase together with the internal representation as another data abstraction. This view of the BAC will be useful when we consider the local nature of the model.

As mentioned above, modifying an object is a matter of generating new context and then regenerating the object in terms of that context. Modification will still follow the BAC. In some cases (i.e., object transformations) actual initiation and articulation events will be necessary, while in others (i.e., changing fill type) initiation and articulation will be implied. In both cases the completion phase will be necessary; but, as we shall see, the former type of transformation will be much more difficult to deal with in the distributed conferencing system.

In terms of our protocols, initiation and articulation will be dealt with via the Image Data Protocol (IDP) (cf. sec. 1.3.1) and completion will be dealt with via the Image Synchronization Protocol (ISP) (cf. sec. 1.3.2). We will define two logical communications channels: *channel 1* to handle the IDP and *channel 2* to handle the ISP.

2.3 Communication Channels

We make the following assumptions about channel 1 and channel 2 communications:

- that both channels will be implemented on a relatively fast (Ethernet-like) local area network
- the communication medium is reliable
- messages arrive in the order sent

2.3.1 Channel 1 Communications and the IDP

Channel 1 is used to carry user events that affect the outer image. These are the primitive input events which include keyboard and mouse I/O. Context defining events are also exchanged on channel 1. In order to preserve the goal of WYSIWIS, these events must be distributed among users with minimum delay. The primary communication principle implies that users should not be blocked, waiting for these events. Also, since the communication medium is reliable we do not require explicit acknowledgements and no other reply is necessary. Thus, communication on channel 1 is asynchronous. Users send input events as they are generated and before they are processed locally. Received input events are queued and processed according to an algorithm which will be discussed later. Events that are generated remotely are processed using the same algorithms as events that are generated locally. Hence, the effect on the outer image of input events generated by any user is the same for both users.

2.3.2 Channel 2 Communications and the ISP

Channel 2 carries events that are intended to maintain the consistency of the inner image across all the users. This means treating the DS as a replicated, distributed data base and utilizing some of the techniques available for such systems in order to preserve its integrity. Thus, channel 2 carries events that will facilitate concurrency and deadlock control and inner image synchronization. Channel 2 activity has asynchronous as well as synchronous components. The asynchronous activity is directed toward setting up a synchronous channel using a two-way handshake. Once the synchronous channel has been set up all communication is blocking, server/client oriented exchanges. Note that this type of communication does not violate the primary communication principle (cf. sec. 1.1.1). For any time a receiver blocks, it will be because it is expecting a synchronous message from the sender. During the synchronous exchange the sender will not block for user I/O. All such synchronous exchanges will be strictly controlled and sequential.

We are now ready to put together the first simple model of the CIS.

2.4 The Multiple-User Basic Action Cycle

We define the Multiple-User Basic Action Cycle (MUBAC) in terms of the BAC and the channels for the IDP and ISP. As we have seen, events generated during the initiation and articulation phases of the BAC are distributed on channel 1 and the completion phase is distributed on channel 2. In effect, the user who begins the cycle would like to update the distributed data base of inner images.¹ He will be acting in this capacity as a *client*, making a request. The other user, who is maintaining a copy of the data, will take part in the update in the capacity of a *server*, responding to a request. Figure 8 shows the MUBAC from the point of view of the client.

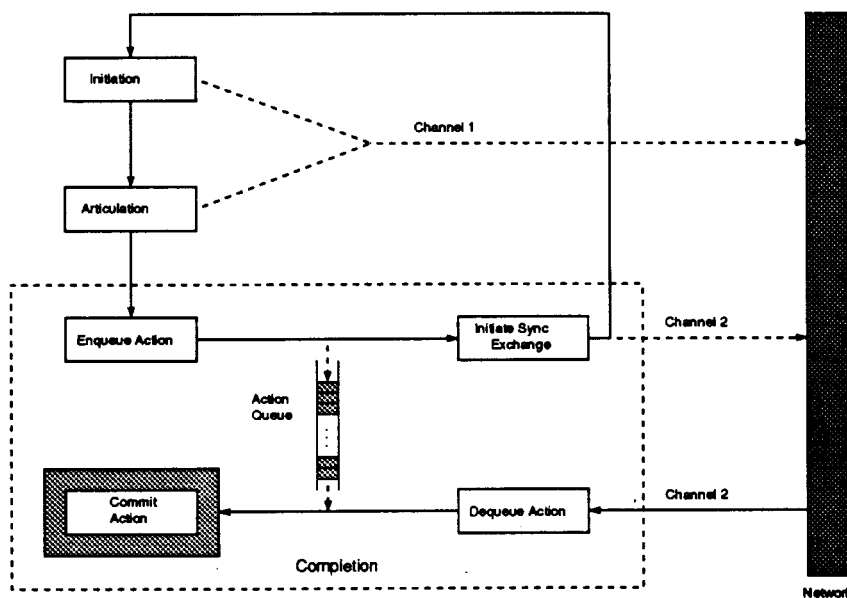


Figure 8. Two-User Basic Action Cycle - Client

(In our figures dashed and solid arrows correspond to asynchronous and synchronous communication, respectively.) After initiating and articulating an object and sending the corresponding events along channel 1, we enter the completion phase. In this phase the action is queued and a request for synchronous communication is sent out on channel 2. When the request has been granted and the reply returned along channel 2, the action is dequeued and the Commit Action subphase is entered. *At this point both users are in Commit Action.* (This might correspond to the

¹ Note that in the CIS the distributed data base is not file-based. Replicated data is stored in main memory at each site.

commit phase in a 2-phase commit protocol.) Commit Action may require many more messages to be exchanged. This will be done synchronously on channel 2. The number and content of these messages will depend upon the action being performed. In Chapter 3 we will supply a basic set of algorithms that will allow us to implement most of the desired actions.

Figure 9 shows the MUBAC from the point of view of the server.

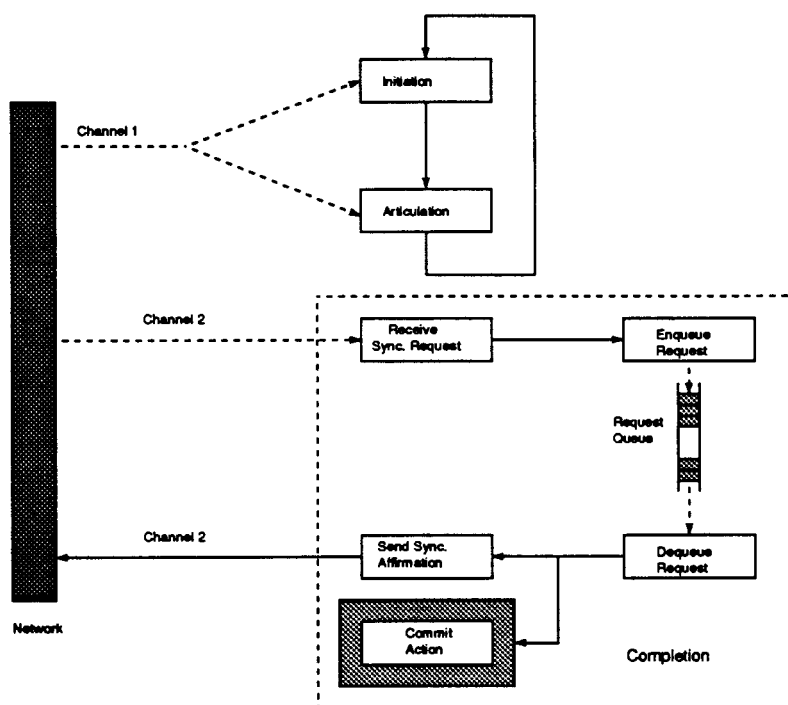


Figure 9. *Two_User Basic Action Cycle - Server*

Note that both initiation and articulation events are sent down channel 1. It is up to the server to maintain enough state information in order to know in which phase to evaluate the events. The context in which we evaluate these events is a function of the *event type* and the *action mode*. The event type is part of the low-level terminal input events, and the action mode is determined by the context-defining events. For most practical purposes the action mode is just a type of drawing operation. Thus, the proper application of an AE event received on channel 1 is a function of two finite variables.

Processing events received on channel 1 during the initiation and articulation phases

causes changes to the visual representation only. When the server gets a synchronization request on channel 2 it is queued until it is able to be serviced. Servicing a request involves dequeuing it, sending off a synchronization affirmation message and entering the Commit Action subphase. As mentioned above, the Commit Action subphase involves an action-specific synchronous exchange of messages to carry out the distributed update.

Figure 10 is a high-level Petri-net modelling the client/server relationship in a two-user model.

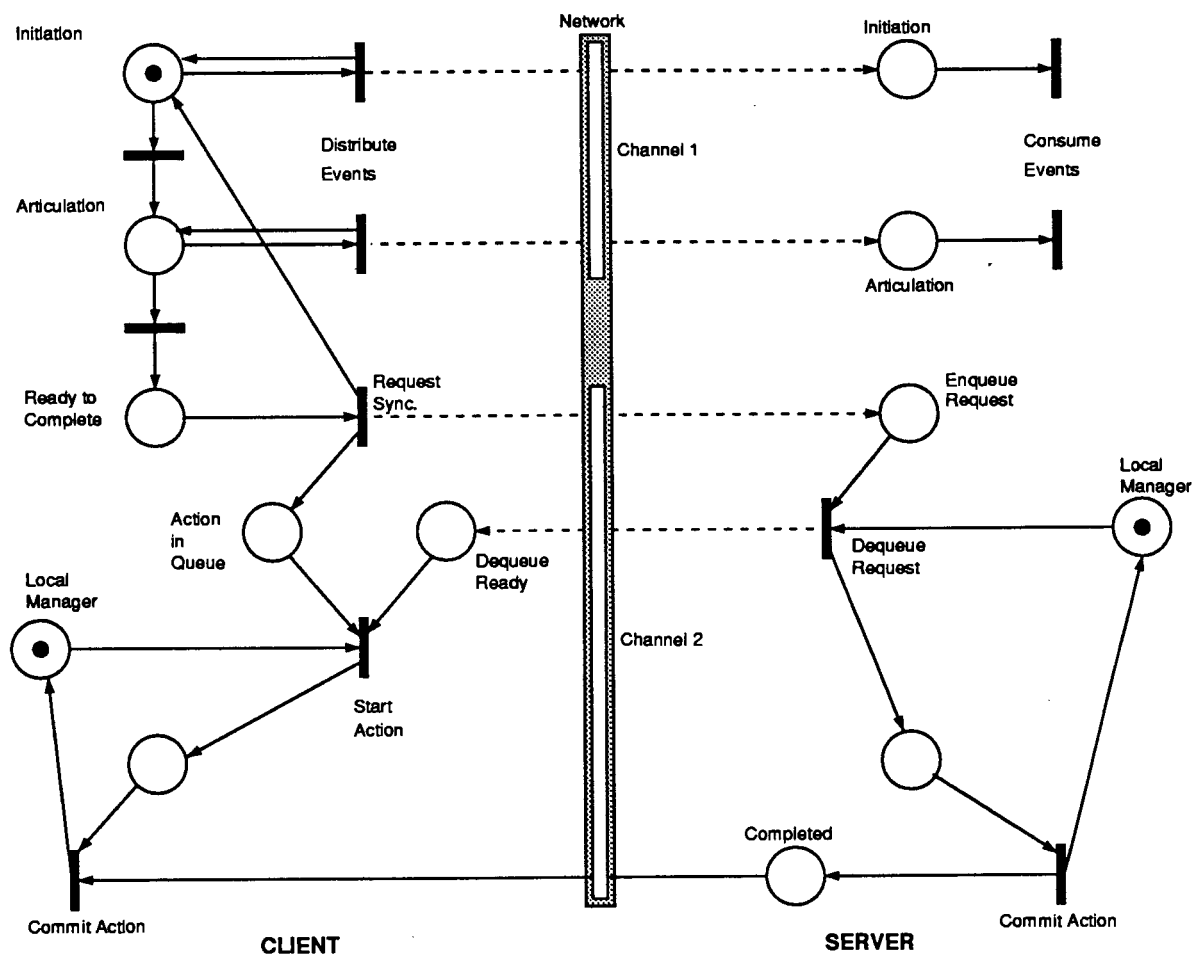


Figure 10. Basic Action Cycle with Two Users

Before the token moves from Initiation to Articulation or from Articulation to Begin Completion, many events may be generated. Thus, a Distribute Events transition may fire several times, sending the locally generated events to the other user, before moving down to the next place. Token

movement from Initiation to Articulation to the Completion Phase phase is user-determined. However, once the completion phase has begun the client is returned to the initiation phase. Since the completion phase is mostly synchronous, it is essential that this phase of the action not be subject to user interaction. The Local Manager abstraction represents the local decision-making process. It is here that synchronization request scheduling is carried out, leading to the invocation of the next Commit Action subphase. Once the scheduling decision is made the appropriate request is dequeued and a message is sent to the client. What is not shown in Figure 10 is the synchronous exchange between client and server during Commit Action nor the process by which the Local Manager determines when to dequeue the next request. Note, however, that the server's Local Manager does not dequeue another request until it receives confirmation that the previous request has terminated. These processes will be modelled fully later.

Note that in the model the client may continue generating actions once the synchronization request has been sent. This is in keeping with the goal of preserving single-user functionality in a multi-user system. The client may continue in this fashion as long as the Action in Queue place does not fill up. Both the client and server visual representations will be consistent up to network and local processing delays. However, if there are tokens in the Action in Queue place then there will be inconsistencies between the the inner and outer state. (But the inconsistencies will be consistent between client and server.) Entry to the Commit Action subphase ensures exclusive access to the inner image and the client dictates the changes that will be made. Thus, after an action has been concurrently executed in the Commit Action subphase the internal representations will be consistent. Therefore the model preserves WYSIWIS.

2.4.1 The Completion Phase

The completion phase begins (in general, not in all cases) with the client sending an asynchronous request message, **Request_Synchronization**, to set up synchronous communications on channel 2. At the same time the current action is enqueued. At this point the client

may restart the MUBAC. The server receives and enqueues the synchronization request, eventually dequeues the request and sends an **Affirm_Synchronization** message back to the client saying that it is ready to begin the synchronous exchange. The server then blocks waiting for the first synchronous message from the client. When the client receives the **Affirm_Synchronization** message it enters the Commit Action subphase by sending a message to the server describing the action mode along with action-specific data. Action-specific data may be, for example, an object (to be inserted) or the name of an object (to be modified). Once the Commit Action subphase has been entered the exchange between client and server becomes even more action-specific. It is clear that there can be no user interaction during the completion phase.

As an example of the action-specific exchange Figure 11 shows a Petri-net model for the Commit Action subphase of the insertion action. Starting with the Local Manager on the server side we see that this subphase begins with the dequeuing of a request. What follows is synchronized up to network and local processing delays. The synchronous exchange ends when the server sends its completion status message and returns to the Local Manager, and the client receives this message and does likewise. We will describe in more detail what happens during the Commit Action subphase of the MUBAC, in Chapter 3.

Remark

In a two-user system the locking activity indicated by the 'Lock DS' transitions is implicit. The server's Local Manager dequeues a request and sends it to the client only when the previous Commit Action subphase has completed. Thus, entry into this phase implies that no other DS activity is in progress and access to the DS is exclusive. In a multi-user system entry into Commit Action may be modelled by the second phase of a two- or three-phase commit protocol. We would require that the distributed data base of queues be consistent among users, thereby enforcing a primitive serialization on user actions. We describe a simple method for doing this in

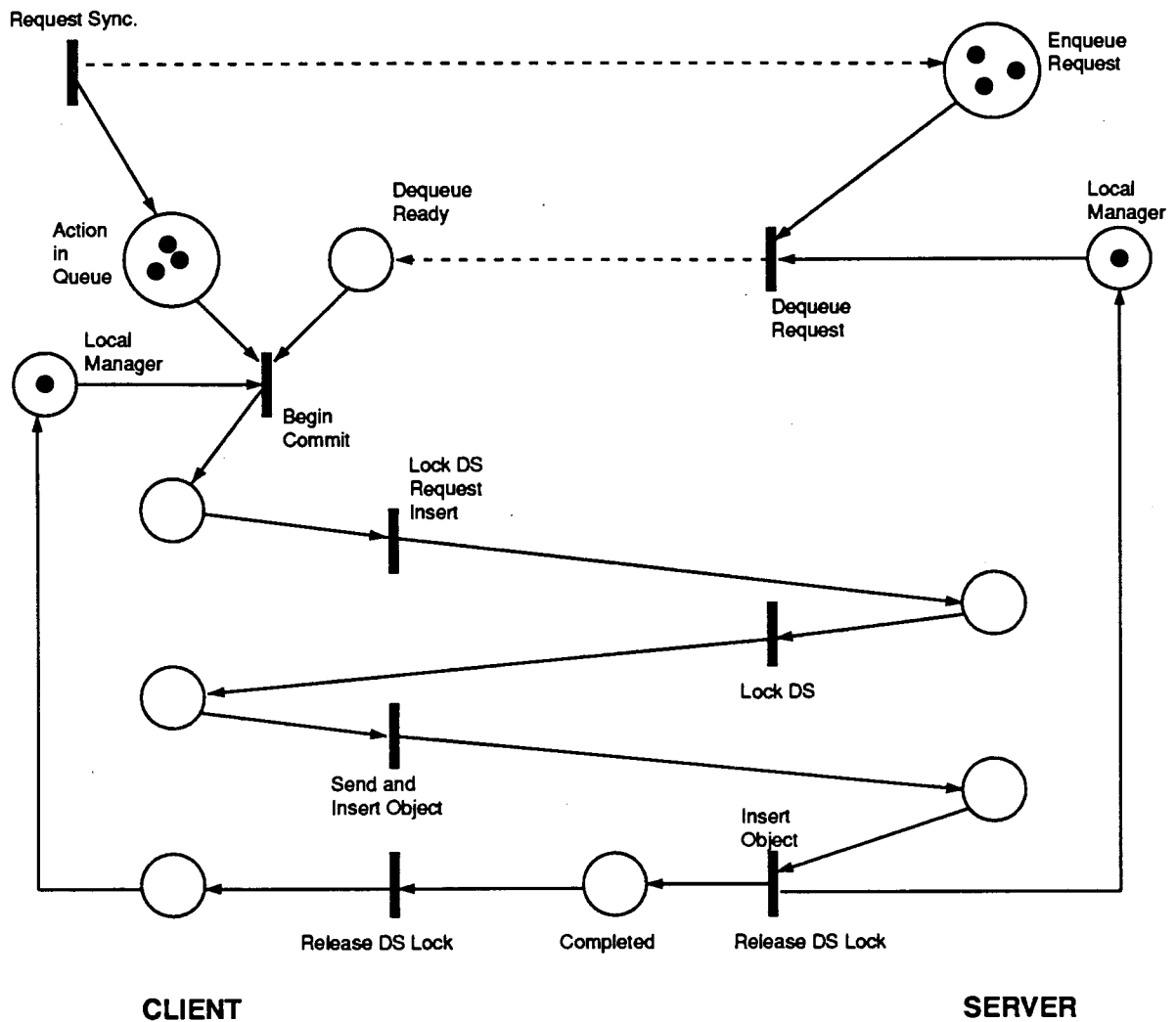


Figure 11. Insertion Commit Action Subphase

the next section.

2.4.2 Deadlock Prevention

The completion phase as described above contains a serious flaw unless we make further assumptions about the way in which this phase operates. Remembering that the client can also be a server when the other user initiates an action, we see that if both Local Managers decide to dequeue a request at the same time, then deadlock will occur because both users will block. We may avoid this by logically combining the Action Queue and the Request Queue for both users.

Figure 12 shows the result of this logical combination.

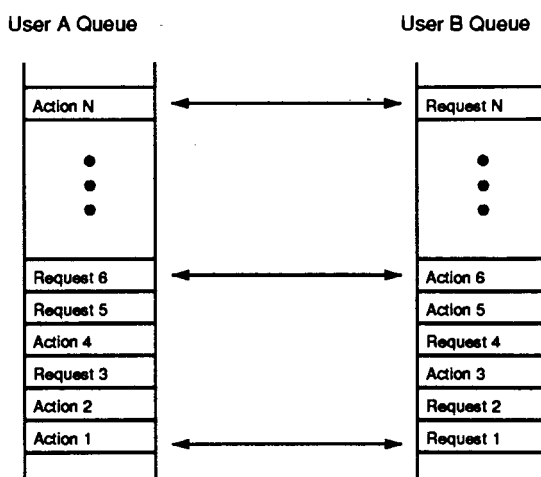


Figure 12. *Combined Action and Sync. Request Queues*

For this to work the queues must be consistent with respect to matching Action/Request pairs. Then the Local Manager would send an **Affirm_Synchronization** message only if it had a 'Synchronization Request' at the head of its queue. Since each Request is matched by a corresponding Action in the other queue only one user at a time will send out an **Affirm_Synchronization** and only one user will block. If **Request_Synchronization** messages are sent simultaneously by both users then we will use event ordering as in [Lam78] to resolve the conflict. If we carry out all dequeuing manipulations within the completion phase then this will ensure that the logical queue remains consistent.

2.5 Local Division of Labour

We have seen that in carrying out a distributed action one user takes on the role of client and the other of server. While the client is executing some phase of the MUBAC the server is executing the *same* phase, on behalf of the client, at its site (with some network delay). But for the CIS to be effective each user must be able to assume client and server roles concurrently. So, while the server is executing some phase of the MUBAC on behalf of the client, it may also be executing some phase of the MUBAC on behalf of itself; that is, assuming the role of client. Of

course, if the server may concurrently be a client then the client must also concurrently be a server. We may model this situation locally by a pair of BAC's as seen in Figure 13.

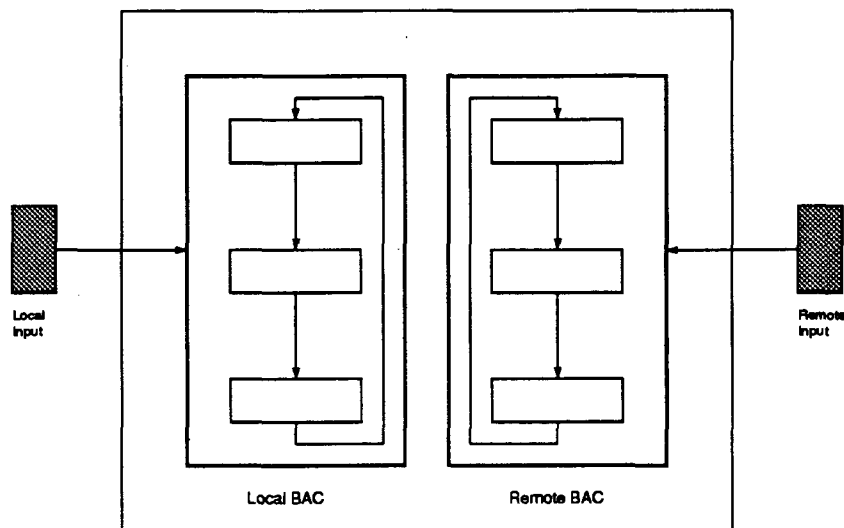


Figure 13. *The Local and Remote BAC*

One BAC, the *local BAC*, receives its input from the local input devices while the other BAC, the *remote BAC*, receives its input from the network. We will also call these the *client BAC* and the *server BAC* respectively.

From the point of view of the user, we would like the local and remote BAC's to exhibit identical behaviour. So remotely generated events should have the same effect on the inner and outer states as locally generated events. The net effect on the CIS should be as if a user had created the image locally on a single-user system. Since initiation and articulation events affect the outer state, the part of the model that affects the outer state should be held in common by both BAC's. Similarly for the completion phase and the inner state. If we now combine this idea with the view of the BAC as a data abstraction, as discussed earlier, we may model the local division of labour as shown in Figure 14. In Figure 14 the initiation and articulation phases of the MUBAC are handled by the *Image Data Manager (IDM)*. The IDM maintains state for the client and server BAC's via the *State Manager (SM)* and manages the outer image via the *Visual Representation Agent (VRA)*. The VRA does not distinguish between locally and remotely

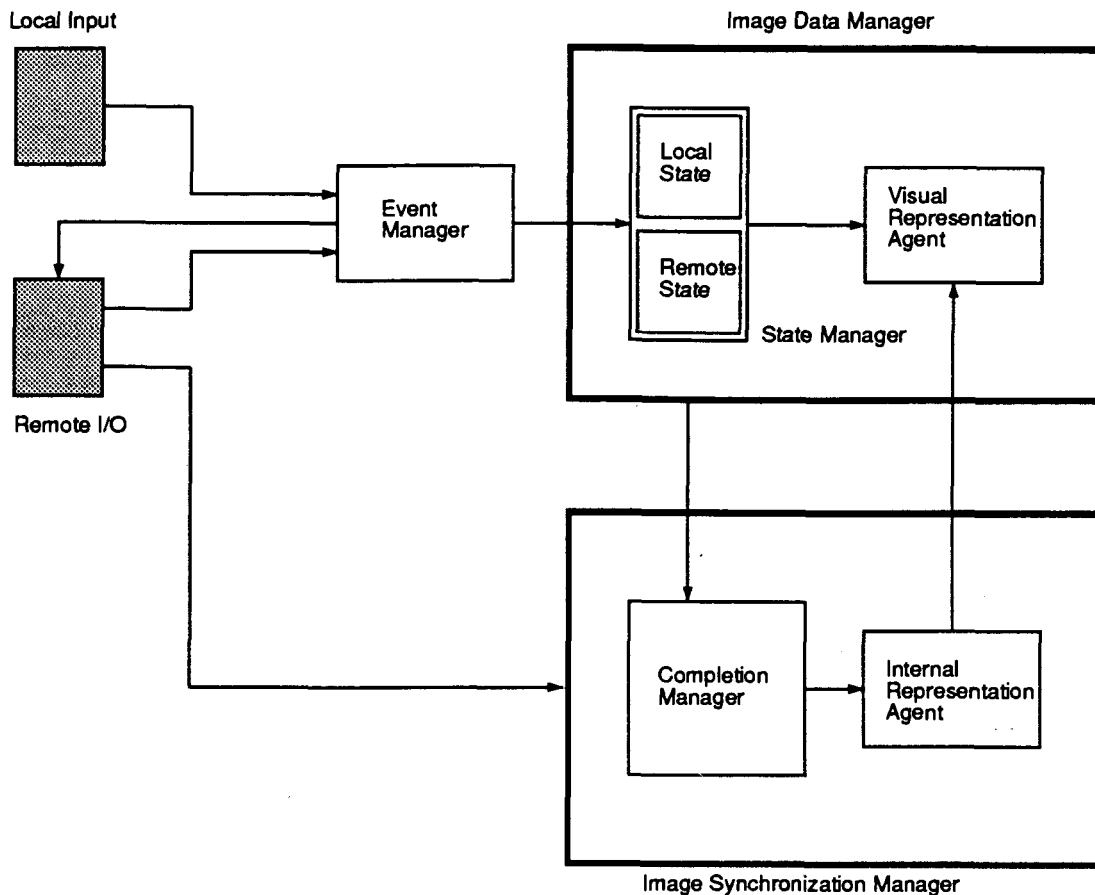


Figure 14. Local Division of Labour

generated events. The *Image Synchronization Manager* (ISM) is responsible for handling the completion phase of the MUBAC. One of the ISM assistants is the Completion Manager (CM) which is responsible for carrying out the Commit Action subphase. The ISM also manages the inner image via the *Internal Representation Agent* (IRA). The *Event Manager* (EM) has the responsibility of taking input from local and remote sources and passing it to the IDM for processing. It is also responsible for distributing local events to the remote EM.

It is important to note that Figure 14 models only the local behaviour of the CIS. It does not model the relationship between local and remote Synchronization Control Managers with respect to the completion phase.

2.5.1 Local Event Flow

Channel 1 Events

When an event enters the CIS the EM first checks whether it is locally or remotely generated and then checks the phase of the local or remote BAC respectively. If the event is remotely generated and the remote BAC is entering the completion phase then the remote user is assuming the role of client and the event is discarded. The reason for this is that once the completion phase is entered it is up to the client to initiate image synchronization, and subsequent events within this cycle of the BAC will be exchanged on channel 2. Certain locally generated events will signify the end of the articulation phase. These events may still affect the outer image so they must be passed to the VRA. Also, the CM must be informed by the IDM of the phase change so that the action can be synchronized.

Locally and remotely generated events at different points in their respective BAC's may be interleaved. The State Manager will check the origin of the event, set the appropriate state environment and send it to the VRA, where it will be applied to the visual representation. As mentioned above, the VRA does not distinguish between locally and remotely generated events; it operates within the context set up by the SM.

Channel 2 Events

Of course, channel 2 events are remotely generated completion events and would automatically be sent to the ISM. As mentioned above, exchanges on channel 2 are handled by the Completion Manager. The CM interprets the input according to the type of the synchronization action and then sends it to the IRA for application to the internal representation.

2.6 Summary of the CIS Model

We began by describing the local representation of images as having two components: the visual and internal representations. Then we described the Basic Action Cycle and its effect on the image representations. These concepts were then extended to the CIS. We have taken the

CIS and represented it in many different ways in order to build the logical components of our model. There are two basic perspectives upon which we based the representations. The horizontal perspective is defined in terms of the communications between corresponding layers at each site. The vertical perspective is defined in terms of the division of labour within a site. Figure 15 shows a conceptualized version of these perspectives.

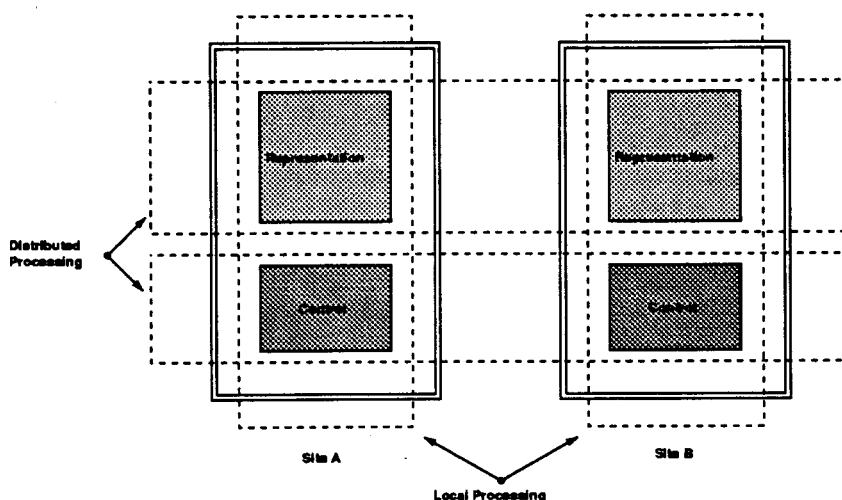


Figure 15. *Modelling the CIS*

We have seen that the requirements for information exchange in different layers affects the nature of the communication in those layers. The local view of the distributed model also indicated that there would need to be some global state retention at each site during some phases of the BAC.

3

Implementing the Concurrent Imaging System

The CIS is implemented in the C programming language running under the Sun UNIX 4.2 operating system (Release 2.3). The graphics interface is implemented using Sun's Suntools windowing system. The hardware is older version Sun 2 diskless workstations running 4Mb of main memory with a 1024×800 pixel raster display and a mouse. Workstations are connected together by a 10Mbit Ethernet with Sun's Network File System providing disk-access support.

The CIS has not been fully implemented. Some operations have not been completed (i.e., the scale operation has many unfinished aspects). Other operations such as 'Overlay' (which would allow 'pieces' of DS's to be taken from other sources and added to the image) and 'Rotate' have not been implemented at all. Many object types which have been included in the data structures do not have supporting creation and manipulation routines (i.e., there are no splines or arcs). Object-attribute support routines have not been fully implemented for all object types (i.e., circle fill routines have not been developed). Not all operations have synchronization code. In these cases the operations are performed in parallel using the same set of input events. Furthermore, some actions are implemented w.r.t. the internal representation (so that image print-outs have the desired effect) but not w.r.t. the visual representation (so the image is not WYSIWYG). Most of the gaps in the implementation could simply be filled in with code very similar to existing code in order to complete the implementation (ie. completing the routines for ellipse management would require almost identical to those for circle management²). For clarity and continuity, we will assume that the implementation of the CIS is complete except where our discussion of

² In fact a circle is just a special case of an ellipse so most of the routines would be identical.

unimplemented functionality goes beyond currently implemented functionality. In this case we will speculate about future implementations.

3.1 CIS Architecture

Up until now our discussion of the CIS model has been conceptual; we have indicated the flow of *ideas* in and around the CIS. We have demonstrated the structure of the communication between two users who wish to access a shared space with a graphics interface. We have shown how each user handles CIS events locally. Now we would like to support the concepts with a more concrete representation that turns the conceptual flow among logical components into the flow of events and messages among modules. It is this version of the representation that will be 'compiled' into the implementation of the CIS.

In an interactive single-user system a simple decomposition of the system components is shown in Figure 16.

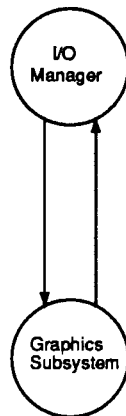


Figure 16. *Single-User Architecture*

The I/O Manager passes input events to the Graphics Subsystem, where they are processed, applied to the DS and passed back to the I/O Manager as output to be displayed. The Graphics Subsystem consists of the DS data abstraction, input-processing modules, output-generating modules and housekeeping routines. Extending this system into a CIS requires the introduction of communications agents. These agents interact with the single-user system by 'splicing' into

the BAC as described in detail above. The modularization of this splicing process for two users results in the architecture shown in Figure 17.

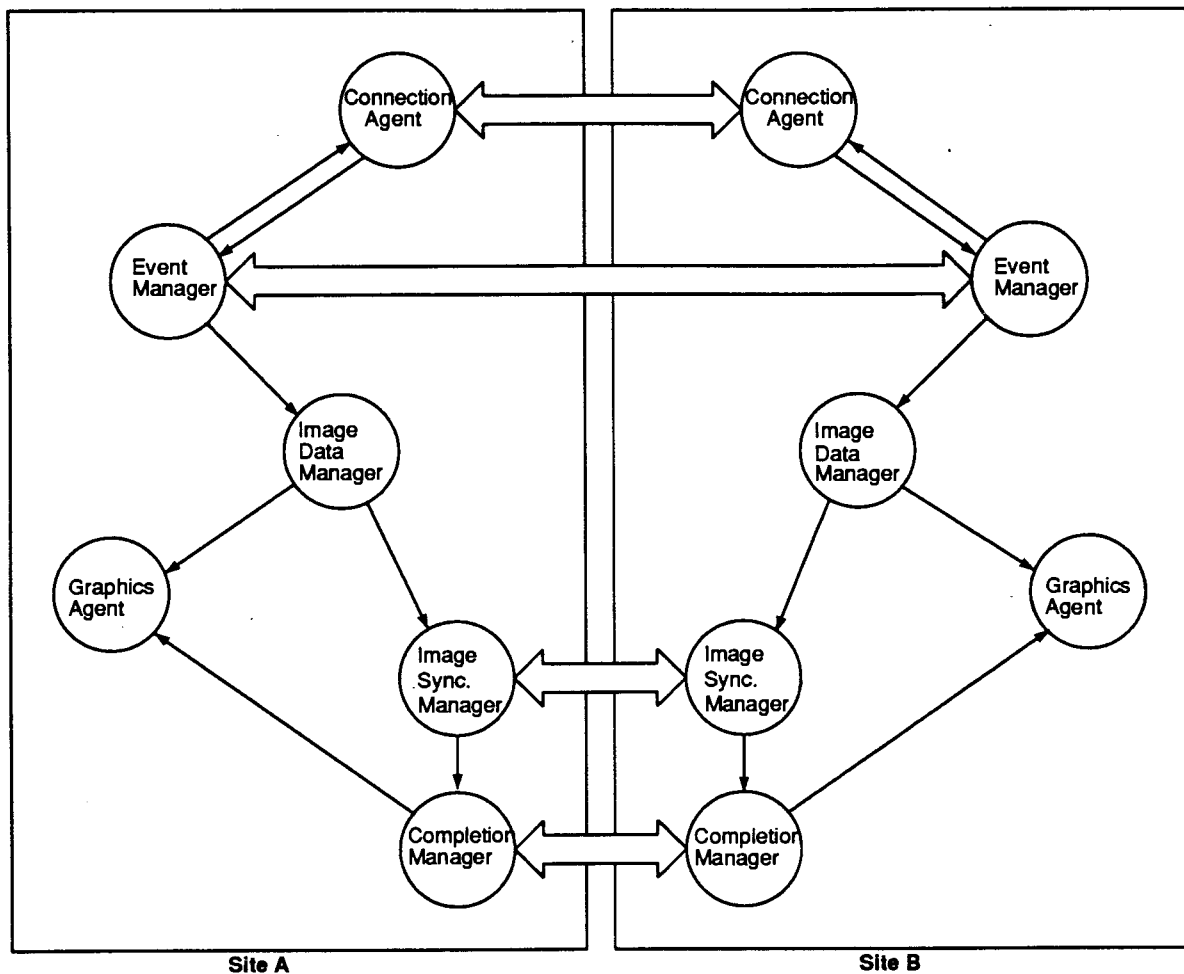


Figure 17. *Two-User Architecture*

The Graphics Agent is the logical combination of the Visual Representation Agent and Internal Representation Agent discussed in Chapter 2. The Event Manager (EM) distributes input to the remote Event Manager and to the Image Data Manager (IDM). The interaction between the IDM and the Image Synchronization Manager (ISM) and Graphics Agents was discussed in Chapter 2. The ISM sets up a synchronization channel with the remote ISM and alerts the Completion Manager (CM) about the forthcoming exchange. After the synchronous exchange the CM causes the representations to be updated.

The Connection Agent is responsible for setting up and maintaining contact with the remote user. If the Connection Agent detects the absence of the other site then it informs the EM. The EM will stop distributing its input and it will inform the other Managers. This action essentially 'short circuits' the IDM, ISM and CM turning the remaining site into a single-user system.

3.2 CIS Data Structures

The internal representation of the graphic image has many functions. It is used to refresh the local visual representation; it is used to update remote copies of itself; it provides feedback when the visual representation is being manipulated; it may be used in concurrency control and deadlock prevention routines; ultimately, it is stored on secondary storage to provide a permanent record of the image. The efficacy of the distributed manipulation of the internal representation determines the extent to which distributed graphical functionality may be implemented. Full graphical functionality in the distributed system is necessary to maintain the goal of WYSIWIS. The expediency with which this functionality is carried out affects the goal of multi-user functionality. Thus, the implementation of the internal representation affects the foundational abstractions upon which the CIS has been designed. We will discuss the main CIS data structure in terms of the operations defined on it and compare the algorithmic complexity to that of a linked list. We will show how the local representation fits into the distributed system with respect to concurrency control and deadlock prevention.

3.2.1 The Display Structure

The internal representation for the graphic image is called the *Display Structure* (DS). The DS is a collection of graphical *objects*. Each object contains information about the *object type*, *object size and location*, *object minimal bounding rectangle* and object-specific *object attributes*. When these objects become part of the DS further information is added to form a *DS node*. This new information includes a unique *object name*, *object-visibility* and *object-lock flags* and a

pointer to a block of *object annotation text*.

Most local actions require that a **search** be carried out in the DS. When the required location within the DS is identified some operation is performed. These include in-place operations on an object, such as attribute changes, and DS-topology-altering operations such as **insertion** and **deletion**. Most of the other topology-altering operations (such as **duplication** and **move**) are combinations of these two basic operations. Thus, we will motivate our selection of a DS by considering the complexity of the **search**, **insertion** and **deletion** operations.

In order to search for an object we must have a key that identifies that object. There are two keys that we will use when searching for an object: the first is by the *object name* and the second is by *rectangular region*. The first key is the unique object identifier mentioned above. Searches using this key simply terminate and return the object with object name that matches the key. When a rectangular region 'key' is used in the search, all objects whose minimal bounding rectangle (MBR) intersects the rectangular region key are returned. Region searches provide user feedback when an object has been found and terminates when either the user chooses an object or the entire DS has been searched.

For object name searches the choice of DS will not matter if the object name bears no relationship to the position of the object within the DS. We would need to search the DS sequentially until a match was made. As we shall see, most searching will be done on the basis of rectangular region or a single point and thus, there is little motivation for relating the object's name to its position in the DS. Therefore, we will attempt to optimize our DS to accommodate rectangular region searches. Searches will be expedited if we can quickly move to the general area within the DS which contains the desired object. Thus, we would like to impose a structure on DS objects which reflects the relative relationships of the objects on the display terminal. Then, if we identify an object which is 'close' to the search key, we should be able to determine in which 'direction' the search should proceed. We define two types of object relationships. An *implicit* relationship is a predetermined rule which is invoked by the system in order to relate the relative

positions of any two objects. An *explicit* relationship is one that is defined dynamically by the user to group or connect non-implicitly related objects.

Our display structure structure is called an *inclusion tree* or *itree*. It is a binary tree where each node contains the information discussed above. The root of the itree is always a special 'meta-object' called the *Display_Screen*. An 'empty' itree will still contain the *Display_Screen*. The left subtree is called the *family subtree* and the right subtree is called the *neighbour subtree*. We say that one object is *contained* or *included* in another if the MBR of the first is contained in the MBR of the second. Figure 18 shows the relationship between a parent node and its family and neighbour subtrees.

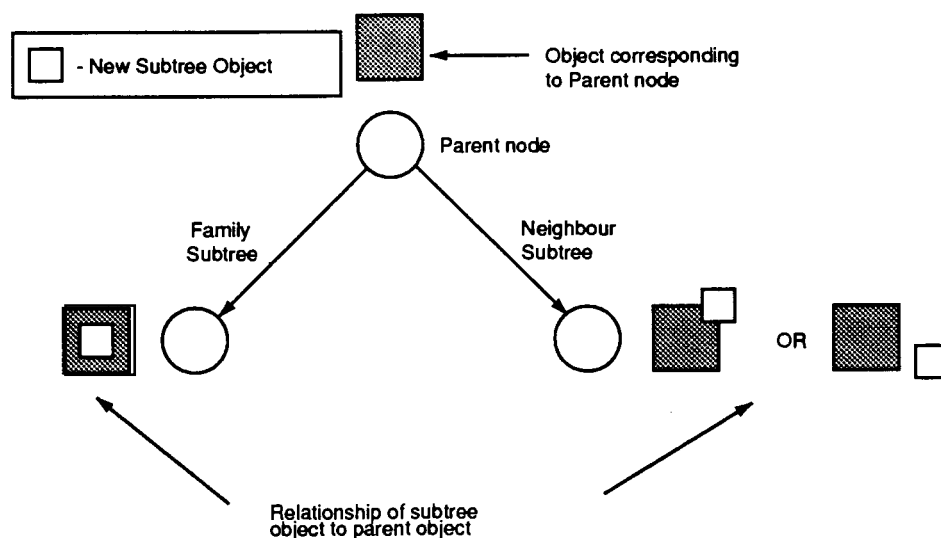


Figure 18. The Itree Structure

The family subtree contains objects which are contained in the parent object. The neighbour subtree contains objects which are not included in the parent. Thus, the neighbour subtree contains objects which may intersect the parent or may be disjoint. We call objects which intersect the parent but which are not included in the parent (and which do not include the parent), *relative* objects. Figure 19(b) gives an example of an itree corresponding to the image of objects with MBR's shown in Figure 19(a). Node numbers in (b) correspond to MBR numbers in (a). Note that the tree structure is dependent upon the order in which objects are inserted. For example, if

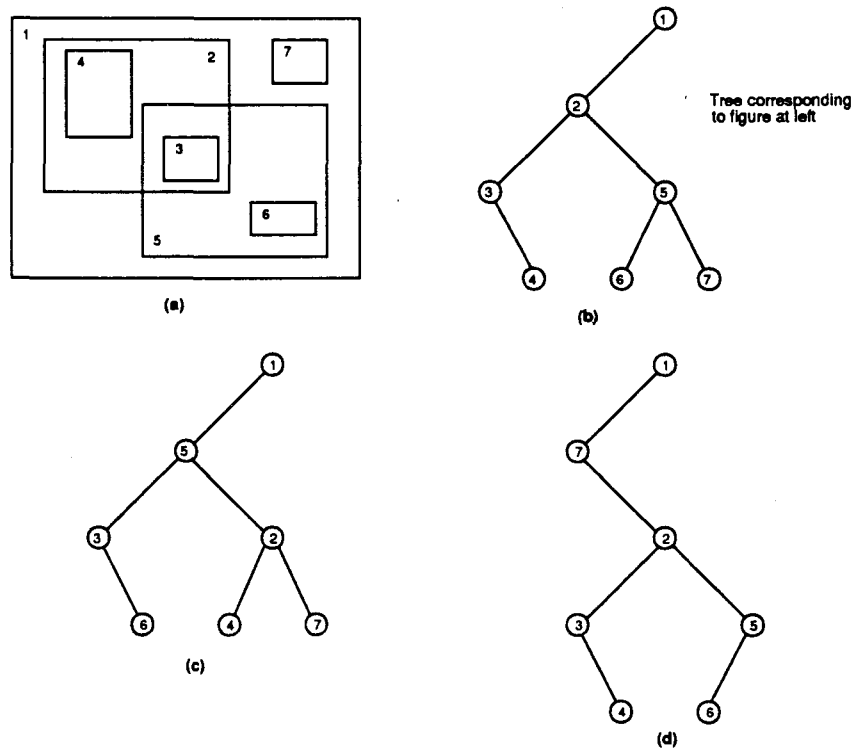


Figure 19. Example Itrees

we reverse the insertion order of objects two and five, then the resulting itree is shown in Figure 19(c). Coincidentally, in this case, the tree topology is isomorphic to (b). If we insert object seven after object one and insert the other objects in order then the resulting itree is shown in Figure 19(d).

The main reasons for using the itree are as follows. In order to **select** an object from the display the user will define a rectangular region around or upon the object. The DS will be traversed to see which objects have MBR's that intersect the defined region. The algorithms which carry out the intersection checks reduce to determining whether or not some point is not contained within a rectangular region. This point could either be (i) a corner of a rectangular region or (ii) a degenerate rectangular region consisting of a single point. Thus, if it is found that a point is not contained in a region then it will not be necessary to check any of the nodes in that node's family subtree. This is because inclusion is transitive and the MBR's of all nodes in a

family subtree are contained in the the MBR of the parent. The inclusive nature of the itree also implies that in order to visit all objects contained in the object represented by node i , we need only traverse the itree in inorder until node i is encountered.

Another reason for using the itree structure is that it provides more flexibility than a linked list, say, in the way in which an image is redrawn. The itree structure collects together into family subtrees, objects that are contained in other objects. At times the containment relationship also has meaning within the context of the image being generated. For example, a box with a text label inside usually indicates a relationship between the label and the box within which it is contained. If the box has a fill pattern then it is important that the label be drawn **after** the box so that it is not obscured by the fill. In general, when redrawing the image, family subtrees must be drawn pre-order. That is, as the family subtree is traversed, each object encountered must be drawn immediately. This way more deeply nested objects will be drawn after less deeply nested objects and will not be obscured. If there is no implied relationship between the nested objects then no harm is done. Thus, the drawing order in a family subtree ensures that all objects within the subtree will be seen. Even if the box with the fill is created after the label, redrawing the image will cause the label to be drawn after the box because it will have become a node in the family subtree of the box.

The itree structure may also provide improved performance when computing the minimum bounding rectangle of the entire image. This quantity is useful when computing image-scaling factors for print outs and for providing feedback during image-shifting operations. In order to compute the image MBR we must check the MBR's of the objects. With the itree structure it is necessary only to check the MBR's of the 'outermost' neighbour subtrees. That is, starting at the itree root, we need only check neighbour subtrees; we never have to check family subtrees because family MBR's are contained in the MBR's of their parents.

3.2.2 The Itree Algorithms

We will cover the **insertion** algorithm in some detail. Many of the other algorithms have similar complexity and, as mentioned above, the insertion operation is one of the building blocks for some of the more complicated operations. Because object locking is tied very closely to selection we will discuss the **selection** algorithm in the next section.

The Insertion Algorithm

The **insertion** algorithm requires a partial pre-order traversal of the itree followed by some itree manipulation. During the traversal at most two comparisons are made between the new node and the currently-visited node. The first comparison checks whether or not the new node is included in the visited node and the second comparison does the reverse. The only relatively expensive case occurs when it is found that the new node contains a visited node. In this case the new node is inserted into the tree in the place of the visited node and the visited node subtree becomes the family subtree of the new node. All that remains to be done is to determine the relationship between the visited node's neighbours and the new node; which of the visited node's neighbours will become neighbours of the new node and which will not? That is, which of the visited node's neighbours are included in the new node and which are not? Initially, the visited node's neighbour subtree becomes the neighbour subtree of the new node. Then each neighbour node in turn is checked to see whether or not it is contained in the new node. If not, it remains where it is; if so, it is put back into the neighbour subtree of the visited node. As we shall see, this situation may lead to a pathology wherein every node in the tree is reinserted.

In the following algorithm the Insert routine is initially passed the node to be inserted and the root of the itree.

```

PROCEDURE Insert ( new_node, node )

1      IF new_node  $\subseteq$  node THEN
2          IF node.family  $\neq$  NULL THEN
3              Insert ( new_node, node.family )
4          ELSE

```

```

5           new_node joins family of node
6   ELSEIF node  $\subseteq$  new_node THEN
7           new_node replaces node in itree
8           node joins family of new_node
9           FOREACH neighbour of new_node DO
10              IF neighbour  $\subseteq$  new_node THEN
11                 neighbour becomes neighbour of node
12   ELSE
13       IF node.neighbour  $\neq$  NULL THEN
14           Insert ( new_node, node.neighbour )
15       ELSE
16           new_node becomes neighbour of node

```

Figure 20. *The Insertion Algorithm*

The correctness of this recursive algorithm is shown inductively. The base case is when the itree is empty. That is, it consists of the Display_Screen meta-object only; no objects have been created. In this case any new_node is contained in node (Display_Screen) and node.family is NULL. Then new_node joins family of node.

Now suppose we have a non-empty DS which has correct parent-family-neighbour relationships according to the definition of the itree. That is, for each node j in the itree, all nodes in the left subtree (family) are contained node j and all nodes in the right subtree (neighbours) are either relatives or are disjoint. Given a *new node* to be inserted into the tree we must show that the algorithm does this in such a way that the itree structure is preserved. We say that a subtree that preserves itree structure is *itree correct*. There are two cases to consider: 1) the new node becomes a leaf in the itree and 2) the new node becomes an internal node. Case 1 may be broken down into two cases: a) the new node is family to its parent and b) the new node is a neighbour to its parent. For case a) the insertion must have taken place at line 5 of the algorithm. Since the parent (and its ancestors) have been inserted correctly, according to the induction hypothesis, and since the new node is contained in its parent, it is clear that the resulting tree is itree correct. A similar argument holds for case b).

For case 2 the insertion must have taken place at line 7 of the algorithm where it replaced

node j in the itree. By arguments similar to case 1 we see that the tree structure is itree correct up to (down to?) the new node. We must show that the new node's subtrees are itree correct. The new node starts out with all of node j 's neighbours in its right subtree. It starts out with node j in its left. Note that since node j is contained in new node and node j 's family preserves itree structure by induction, new node's left subtree is itree correct. The right subtree may not be itree correct, however, because some of the nodes in that subtree may be included in the new node. It is clear that if we remove any included nodes from the right subtree the resulting subtree will be itree correct. Now the included nodes must be inserted into the left subtree (as family). Since these nodes were neighbours of node j and since node j is the first node in the new node's left subtree, these nodes will still be neighbours of node j and the subtree with node j at its root will be itree correct. Therefore, both subtrees of the new node will be itree correct and we are done.

The Deletion Algorithm

The **selection** operation identifies an object called the `Current_Selection`. It is the `Current_Selection` that is operated on by the **deletion** operation. Therefore, with **deletion**, no searching is explicitly necessary because the searching is done during the **selection** operation. As we might expect, the **deletion** operation is just the inverse of the **insertion** operation, complete with pathologies. That is, under certain conditions this operation may require a large number of manipulations. Initially, the idea of the algorithm was to remove the deleted node from the tree and replace it with its first neighbour, as shown in Figure 21(a) and Figure 21(b). Then the deleted node's family subtree and all the neighbours of the family subtree would be reinserted into the tree with the search starting at the parent of the deleted node. Figures 21(c), (d) and (e) show three possible resulting itree configurations. We can see that under certain configurations (i.e., the deleted object contains all the other objects in a linked list of neighbours) this may require all remaining nodes to be reinserted leading to $O(n)$ manipulations.

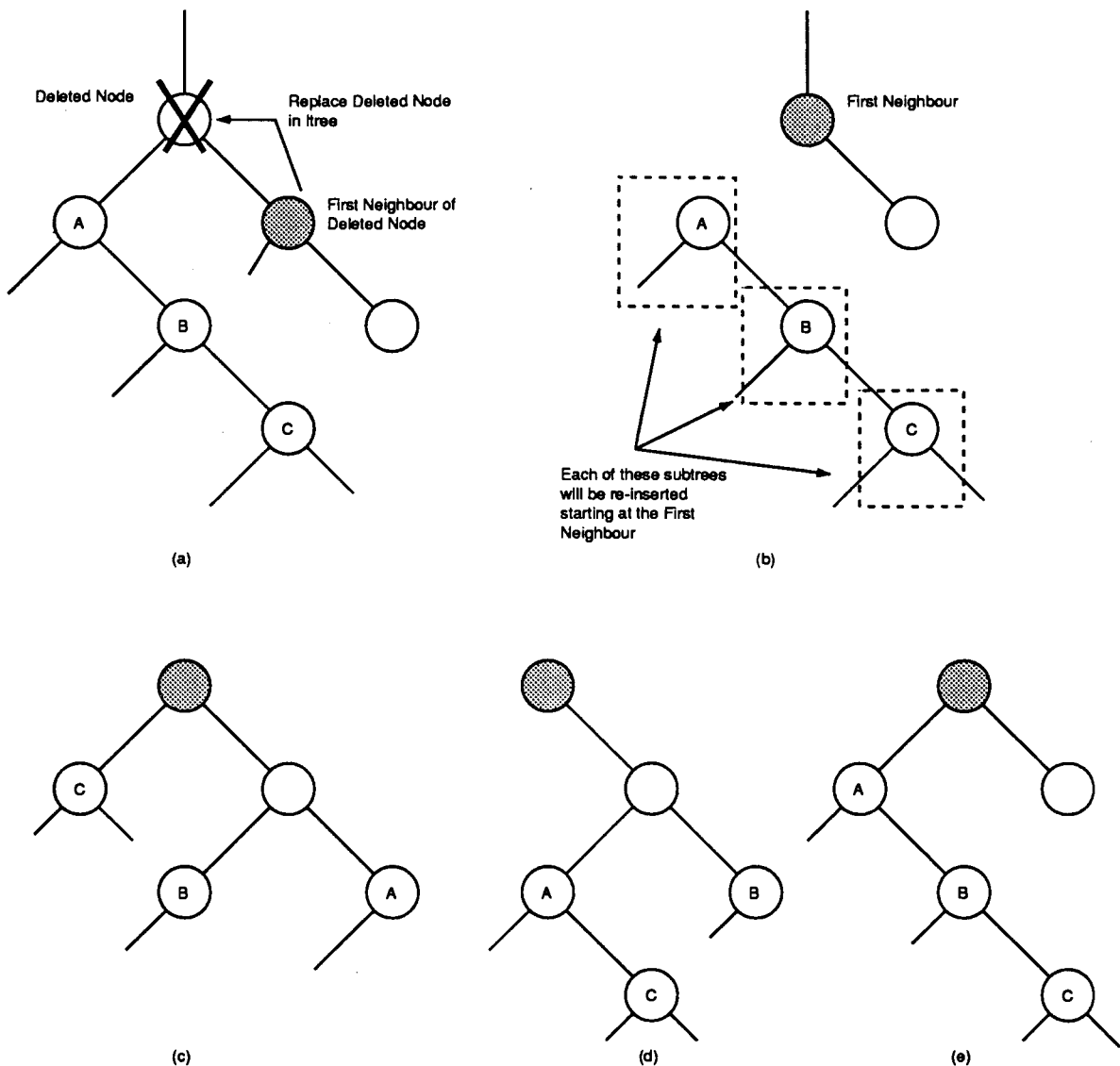


Figure 21. The Deletion Algorithm

To avoid these costly operations we introduced a concept we call *object transparency* into the implementation. As mentioned above, each node contains an *object-visibility* flag. With this flag set the object is *visible*. When it is reset the object becomes *transparent*. When the itree is traversed to redraw the image, transparent objects are passed over and not drawn. To the user, a transparent object is 'deleted'. Thus, deleting an object consists of first locating it, then resetting the object-visibility flag and finally undrawing it. No topology-altering manipulations need

to be performed. Furthermore, we maintain a list of the transparent objects so that we may easily recover 'deleted' objects. We call such a list a *release list*.

Transformation Algorithms

Each of the **scale**, **rotate** and **move** operations require deleting the `Current_Selection` and then inserting the transformed object. Thus, these algorithms have been implemented as **deletion/insertion** pairs. The only extra work involved here is recomputing and redefining the object in terms of the transformation. We also note that the **duplication** operation is identical to the insertion part of the **move** operation.

The Hide and Expose Algorithms

The **hide** and **expose** algorithms change the physical structure of the itree by operating on the `Current_Selection`. Since redrawing an image is done by traversing the itree, changing its structure may affect the way it appears on the output device. To see how these operations work refer to Figure 22. Figure 22(b) shows the itree corresponding to Figure 22(a) (the object in (a) are numbered but the numbers themselves are not objects - they are there for reference in (b)). We can see that objects 2, 4 and 6 form a sublist of neighbour objects with 2 at the *top* of the list and 6 at the *bottom*. The **hide** operation moves a node to the top of a neighbour sublist. Since objects are drawn pre-order a hidden object will be drawn before other objects in its sublist. Therefore, other objects will be drawn after (i.e., 'on top' of) the object, making it appear as though the object has been 'hidden'. Figure 22(c) shows the effect of **hiding** object 6. Figure 22(d) shows the itree after the **hide**. Similarly the **expose** operation moves a node to the bottom of a neighbour sublist. Figure 22(c) also shows the result of **exposing** object 2. In this case, however, the corresponding itree would again be different.

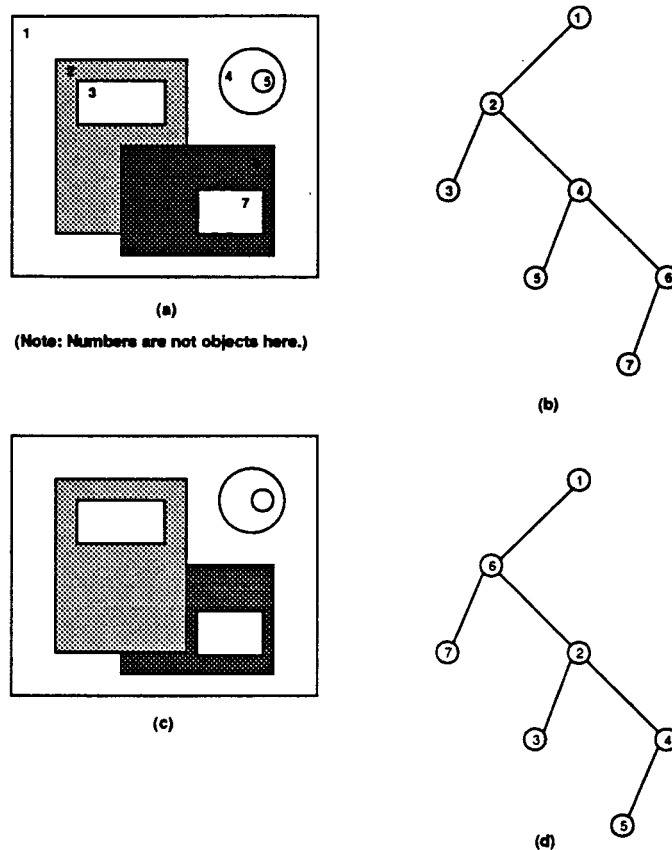


Figure 22. The Hide Operation

3.2.3 Object Selection and Locking

Object selection in the single-user system is often referred to as 'picking' in graphics texts. We use the term 'selection' to mean 'distributed picking'. We will describe the methods used to select an object which relate outer image activity to inner image nodes. We will discuss the methods we use to solve the selection ambiguity problem (how to select one of many coincident objects), and how they relate to the itree DS. We do not consider an object as 'selected' until it has been locked. Thus, locking an object is considered to be part of the selection process. We will discuss the distributed locking algorithm and its relationship to the itree.

Selection

As we have seen, many of the DS algorithms operate on the `Current_Selection`. In the above discussion the `Current_Selection` is determined by a rectangular-region key search. The rectangular region is generated in two ways in order to define the *selection granularity*: the *wide-area search* and the *narrow-area search*. In the wide-area search there is an initiation phase where the anchor point of the rectangular region is defined. The articulation phase determines the terminal coordinates of the region. With this search mode the entire display screen may be searched. In the narrow-area search the *size* of the rectangular region is already defined. It is a square with five pixels on a side. The initiation and articulation phases are combined into one phase when the user determines the square's position on the display.

Both search modes do an in-order traversal of the itree and determine whether or not the defined rectangular region intersects the MBR of each node visited. Since a rectangular region may intersect more than one object, when a node is found to intersect this region then the object is highlighted on the display screen and the user is prompted to determine if the highlighted object is in fact the desired one. User prompts are one method of dealing with the ambiguity problem when selecting objects. With wide-area searches rectangular region intersection with an MBR is enough to select the object for feedback from the user. With narrow-area searches, however, further refinements are necessary. We want to be sure that some portion of the object which is drawn on the display is within the small square. For example, rectangles and circles must have some part of their perimeter within the square. For lines and rectangles (and polygons when implemented) we use the *Cohen-Sutherland* clipping algorithm (see [FoV84]). The narrow-area search more clearly disambiguates the object which is being selected because the criteria for selection are more rigorous. However, because the selection square is so small, the fine physical movement necessary to choose the desired object is more critical.

We deal further with the ambiguity problem by introducing input parameters for the

selection routines. For each of the search modes there are two input parameters which may be set by the user. The first (which has been implemented) will filter objects according to their type. We call this the *selection type*. For example, the algorithm may only select objects which are circles. The second (which has not been fully implemented) will allow the user to select a set of objects instead of a single object. We call this the *selection multiplicity*. (Another possible parameter would be to allow the user to disable the feedback mechanism. In this case, all objects found by the selection algorithm would be returned without prompting. With this parameter set, disambiguity is implicit.) With all of these operations implemented the user would be able to select all objects of a certain type in one area of the display and perform some operation on them. For example, we could change the point size of all text objects on the left side of a diagram.

Note that because the search is in-order, family objects will be visited before parent objects. Thus, even though an object's MBR may be properly contained in another's, we may still easily select that object. Therefore, if we wish to select an object that is nested within several other objects, then that object will be selected for prompting before any of its ancestors. In this case, it is the itree structure itself that contributes to solving the ambiguity problem.

Locking

The Commit Action Subphase of the selection action begins with the client sending a message to the server requesting a lock on an object, after locking the object locally. Along with the locking request is sent the object name, which uniquely identifies the object in all copies of the DS. When the server receives the locking request it uses the object name as a key to search the itree. Thus, we see that the distributed search operation uses both types of keys to search the DS. As it stands, the server-side search is not benefitted by the fact that search is being carried out on the itree DS. As mentioned above, the reason for this is that the object name bears no relationship to the itree structure and therefore cannot be used to eliminate paths in the itree. When

the object is located the server checks the status of the lock by checking the object-lock field of the node. If it is not locked the server locks it and returns a message granting the lock to the client. Otherwise it sends a message refusing the lock. Note that in a two-user system all locks are exclusive.

3.2.4 Data Structure Management

The principal function of the Completion Manager (CM) is to separate user interaction from synchronization activity in order to minimize response time. While this is a logically sound move, it has more implications when we consider the actual implementation, because it is the CM that invokes the creation, deletion and manipulation of distributed DS objects. Therefore, it would appear that the CM should have direct access to the DS. However, the IDM also requires access to the DS in order to refresh the outer image. Furthermore, we will also need file access modules to load and store the DS. So many components of the system need access to the DS. Furthermore, we assume that there is no shared memory among processes. Thus, since the CIS DS is memory resident, if system components are implemented as individual processes then we must decide exactly *where* the DS will reside and *how* the various subsystems will access it.

To do this we must recognize that some actions require single input events³ in order to effect changes to the DS. And other simple actions (like **deletion**) may cause extreme modifications to the DS. These considerations would suggest that the CM would best be suited to handle the DS. However, the idea behind the CM was to optimize user response time by isolating the synchronous exchange of information from other user activity. If the CM becomes responsible for DS maintenance then it will need to have an interface to other modules in order to manage requests for DS access and take Synchronization Requests as well as a synchronization interface to the remote CM. This could defeat its intended purpose. These and other considerations will be

³ Like the press of a mouse button in order to undo the last operation.

taken into account when we evaluate this aspect of the implementation in Chapter 4.

3.3 CIS Communications Protocols

The CIS is implemented on top of Sun's Suntools windowing system. With this system the user creates a window with various attributes and display capabilities by issuing a series of Suntools subroutine calls. This 'master' window can be divided into a series of subwindows, each with a specific purpose (see Fig. 30). The Suntools system treats each subwindow as a 'pseudo device'; subwindow I/O has semantics similar to other UNIX file/devices. The main difference is that the window device drivers present input to the operating system as a stream of *input events* rather than a stream of bytes. Each event is a structure consisting of an event code which corresponds to a mouse movement, or a mouse button or keyboard press, the x-y position of the mouse within the subwindow in which the event took place, an event timestamp and some flags further refining the type of event. It is these events that are exchanged via the Image Data Protocol.

The Image Synchronization Protocol exchanges complex data structures. These structures are not events, as described above, but consist of a synchronization request (reply) code, some request- (reply-) specific data and a sequence number. We will also refer to synchronization requests/replies as *synchronization messages*. Synchronization messages are usually issued transparently on behalf of a user as the result of some user action.

Communication between users is implemented as a series of parallel connections between subwindows. The IDP is carried out across two pairs of subwindows and the ISP is carried out across a single pair of subwindows. Figure 23 illustrates this communication.

The communications transport is implemented via point-to-point User Datagram Protocol (UDP) sockets. The interface to the socket abstraction is via The Integrated Message Exchange System (TIMES) written by the author. The TIMES package is supported by a distributed name

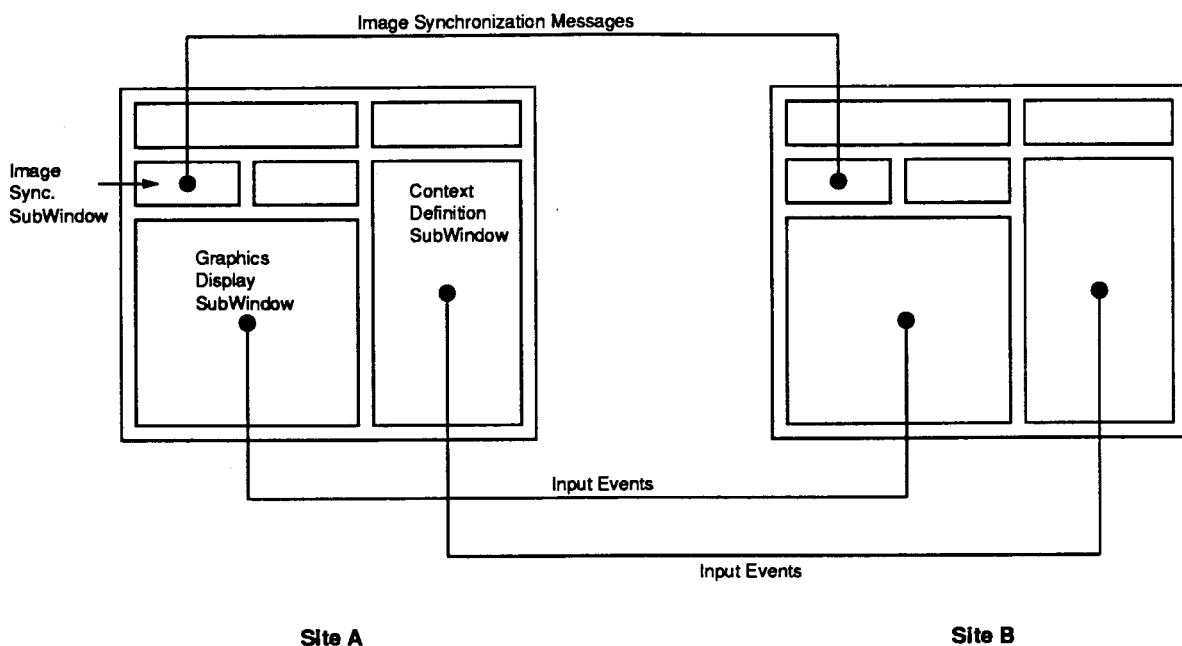


Figure 23. Communication Channels Between Subwindows

server which runs continuously on all machines in which the CIS may be invoked. The name server is used to implement the inter-window Connection Protocol, which allows the connections shown in Figure 23 to be made dynamically. We now give a more detailed description of the protocols.

3.3.1 Image Data Protocol

As mentioned previously the Image Data Protocol is implemented along the logical Channel 1. The actual implementation uses two UDP channels; Channel 1A to exchange Context Defining Events between the Context Definition Subwindows and Channel 1B to exchange Action Events between the Graphics Display Subwindow (the outer image). These Events consist of raw input events generated by the hardware and passed to the appropriate subwindow. The subwindow, via the Event Manager, immediately sends the event to the corresponding subwindow at the other site and then processes the event locally.

The Event Manager is responsible for distributing locally generated events and for

receiving remotely generated events on Channel 1A. Locally and remotely generated AE's are processed by the same set of routines. The correct routine needed to process a particular event is a function of the *event type* and the *action mode* (operation type).

The action mode is determined by the CDE's. A CDE is generated in the Context Definition Subwindow by placing the cursor over the desired operation type and pressing a mouse button. The window system determines which operation type has been chosen by the position of the cursor within the subwindow and the appropriate routines are then called. Thus, the action mode is a function of the input event's x-y coordinates. These events are passed along Channel 1B to the remote user. When the remote EM receives the event it passes it directly to the window system. Since both users are running identical window layouts, events received on Channel 1B are processed by the same routines for both users. That is, the same action is chosen by both systems for an CDE originating in one of the user's Context Definition Subwindows. The action mode chosen will be maintained by the State Manager as part of the *local state* for the originating user and *remote state* for the other user.

3.3.1.1 Distributed State Information

We have discussed the notion of locally and remotely generated action modes which are managed by the State Manager. The action mode is just one of the important parameters that the SM maintains. The SM keeps track of all data needed by the Visual Representation Agent in order to draw or update the display on behalf of either user. The state is stored as an array of structures, with one entry per user. Each event received by the SM is checked to determine its source. The event source is mapped to a global array index. When the event is passed to the VRA it is interpreted in the context of the state array with entry corresponding to that index.

The state structure may be divided into six categories: *page attributes*, *object attributes*, *selection data*, *undo operation data* and *graphics* and *text display data*. Page attributes include

the action mode, the current display page (see next paragraph), an action-in-progress flag and gridding (snap) information. Object attributes include line width, line type, fill type, font, and point size. Undo operation data includes pointers to the most recently created and most recently changed objects and the last operation performed by the user. Selection, graphics and text display data all include initial and terminal coordinates. Selection data also includes the current selection and the selection granularity, multiplicity and type. Text display data also includes a text buffer and a pointer to the current position within the buffer.

We have implemented the CIS to allow multiple images to be manipulated at one time. There are eight images called *pages*. Each page corresponds to an itree. This means that much of the state data must be maintained as an array. For example, the object attributes, undo operation, selection filter and gridding data may be different for each page. Thus, an event is not only interpreted according to the user that generated it, but according to the page on which it was generated. Paging information is exchanged as part of the IDP and is maintained as the *current display page*. Thus, as well as being an index into the distributed state array, the current display page is also an index into an array (a forest) of itrees.

The last detail of the IDP implementation involves the exchange of certain distributed state information. An event is passed to the Suntools system and processed as if the event was generated locally. In most cases the event is generated by positioning the mouse pointer over the appropriate symbol and pressing a button. But in many cases the value returned by the event depends upon a current value maintained by Suntools. Since the only value the Suntools system knows about is the *local* value, remotely generated events will (i) usually return incorrect values for remote state and (ii) generate new, incorrect values for the local state. To deal with (i) we have the client send an update message along with an attribute/value pair to server after the event has been processed locally. The server can then set the remote state accordingly. We deal with (ii) by having the server update its local state from the state array after receiving and processing

the remote event. Because these update messages have a structure different than an input event and identical to a synchronization message, they are sent along Channel 2. Unlike other Channel 2 messages, context update operations are carried out as soon as they are received.

3.3.2 Image Synchronization Protocol

The main purpose of this protocol is to provide consistency of the inner image in an efficient and deadlock-free manner. This is accomplished by setting up a synchronous session between the two users (or their agents) whenever it is necessary to update the DS. We would like these updates to occur transparently to the user without degrading response time. This implies that "vocal locks" or other kinds of verbal communication between users is unnecessary and undesirable. Each action will consist of a predetermined set of server and client subactions.

The descriptions and examples in the following section extend the data abstraction described above across the network. We describe what happens more specifically during the Commit Action subphase of the MUBAC by giving examples of some of the basic operations. These examples will indicate the synchronous nature of the completion phase and the amount of communication necessary to carry out the action.

3.3.2.1 Examples

We will describe our actions for a two-user implementation using diagrams, called Message Passing Diagrams (MPD's), with the key shown in Figure 24. The first actions we will describe are the selection and deselection actions.

Example 1. selection/deselection

Many actions require a selected object on which to perform an operation. The purpose of

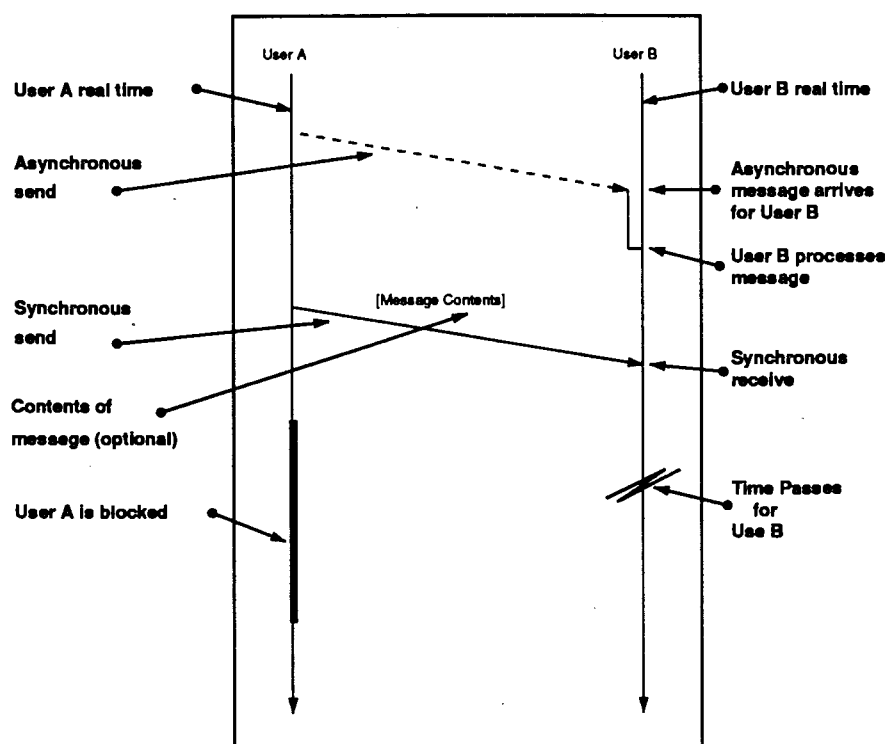


Figure 24. Key to Message Passing Diagrams

the **selection** action is to identify and gain exclusive control over an object.⁴ Control is signified in the form of an *object lock*. A locked object may not be deleted or otherwise modified by anyone other than the user who locked it. The currently selected object is called the `Current_Selection`. Both users maintain two `Current_Selections`: one for themselves and one for the other user as discussed in the section on distributed state information.

The basic action cycle for selection is an exception to the general description of the completion phase. All messages are asynchronous and a **Request_Synchronization** is never sent. This action takes a rectangular region of the outer image and determines all the objects that intersect this region. The user chooses one of the objects and both users' copies of the chosen object are locked. Some other operation is performed on the object (perhaps), and the lock is

⁴ In the general model we consider the **selection** of a group of objects. For now we will consider selection of single objects only.

released (the object is deselected).

Figure 25 gives a detailed description of the messages passed in order to indicate the flavour of such an exchange. This is not to be taken as the full description of the selection algorithm.

Selection Action Cycle

Initiation Phase

- determine the anchor coordinates of the rectangular region
- send input events

Articulation Phase

- determine the opposite corner of the rectangular region dynamically
- during this phase the outer image shows the rectangular region.
- send input events

Commit Action Subphase

- client attempts to lock object locally
 - if successful then
 - send a **Request_Object_Lock** message
 - receive remote object lock status message
 - if **Object_Lock_Request_Refused**
 - release local lock
 - abort selection action
 - else
 - abort selection action
- server receives **Request_Object_Lock** message and attempts to obtain local lock
 - if successful then
 - send an **Object_Lock_Request_Granted** message
 - else
 - send an **Object_Lock_Request_Refused** message

Deselection Action Cycle

Initiation Phase and Articulation Phase are empty

Commit Action Subphase

- client releases local lock and
 - sends a **Release_Object_Lock** message
- server releases local lock and
 - responds with an **Object_Lock_Released** message

Figure 25. *Message Passing During the Selection BAC*

Figure 26 shows the MPD for the selection action cycle.

Deselection is an implicit action. An object will be deselected automatically if another object is selected or if an action mode is entered that does not permit operations on selected

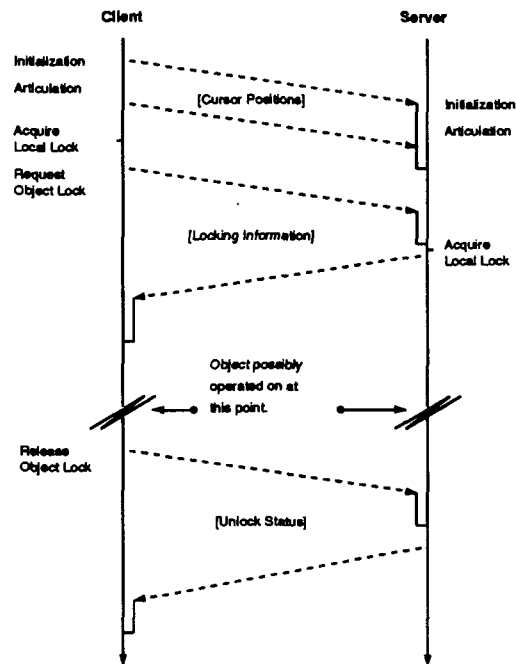


Figure 26. The Selection Operation MPD

objects.

Note that the server will find the object locked only if the server recently locked it itself (in a two-user system). If it receives a locking request from the client after that, this means that the client has not yet processed the server's locking request. In this case both server and client will refuse the locking request and the request will have to be resubmitted. That is, the object will have to be re-selected. This is the system as currently implemented. It should be noted that to send and receive a message takes about 20msec. While a search is being carried out no more messages are sent or received. Thus, client and server would have to submit their requests to lock the same object within about 20msec in order for the above simultaneous locking event to occur. Even if processing times were 25 times slower this would still mean that users would have to submit requests within half a second. The probability of this happening often enough to cause more than minor irritation to the users is small.

We should, however, describe a method of concurrency control that is already partially

implemented and that will ensure that one of the users will always get the lock. First, we need a global clock mechanism such the one described in [Lam78]. (This is already implemented and is used to resolve conflicting synchronization requests.) If clock values are equal then internet addresses are used to resolve conflicts. Since internet addresses are unique within LAN's we will always be able to grant one user or the other the object lock. At this point there are two possibilities for handling the rejected request. First, we may discard the request and inform the user accordingly. Second, we may queue the request and then grant the lock after the first user releases it. The second method ensures fairness. This would be more useful in a multi-user system; in a two-user system the second user could re-select the object when the other user finishes his operation. There are many reasons for choosing the first method over the second: the other user may not want the locked object after it has been manipulated - there would need to be extra user commands for explicitly dequeuing lock requests, complicating the user interface; there will have to be an entire data abstraction implemented that handles the request queue; the other user may not be in a state that is compatible with receiving the lock on an object. A full implementation of the second option would probably become a working example of the Law of Diminishing Returns.

Example 2a. insertion

The purpose of this action is to create new objects and to update the inner and outer images. The client creates the object and initiates the update. Figure 27 shows the MPD for the **insertion** action. The Initiation events determine the new object's anchor coordinates and the Articulation events the terminal coordinates. Once these coordinates have been determined the client queues the action and sends a **Request_Synchronization** message. The server eventually dequeues the request and sends an **Affirm_Synchronization** message in reply. This message also contains a description of the action being processed. When the client receives

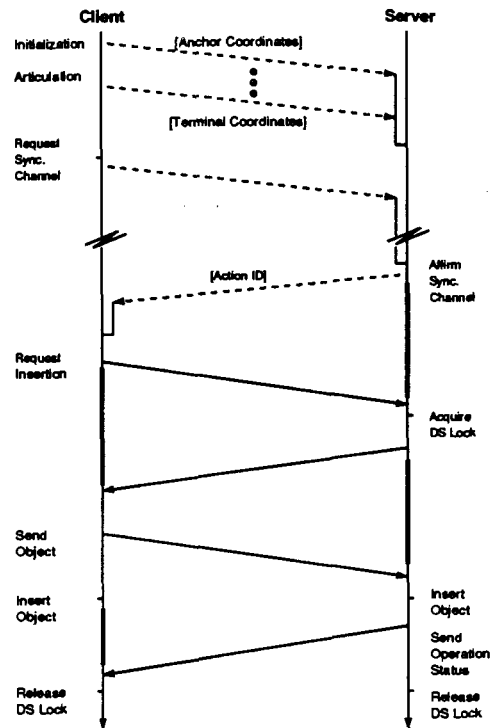


Figure 27. The Insertion Operation MPD

this message it locks the local copy of the DS and sends an **Insert_Object** message. The server responds to this message by locking the local copy of the DS, and sending the locking result back to the client. Now the client sends the object itself to the server and they both insert the object into the DS and release the DS lock. Finally the server sends an operation status message (i.e., **Operation_Successful**) back to the client and the action terminates.

Remark

After the articulation phase of the **insertion** action the server returns immediately to the initiation phase. The only indication that an action is in progress may be on the outer image. There has been no retention of anchor and terminal coordinates, object type, etc. The inner image has not been modified in any way. In other words, the server maintains no state about the current remote action; it has forgotten. The obvious advantages of this are that the server has less state to

manage and, if we allow insertion or articulation events to get lost, then only the outer image may be affected. Alternatively we could require the server to create and store the object at the end of the articulation phase. Then during synchronization the client would only have to send the object identifier (each object must have a unique name) instead of the entire object and this would cut down on the message size. Now in a fast network (i.e., 10Mb), unless the object size is extremely large relative to the size of an object identifier, the message size advantage is outweighed by the advantages of a stateless server.

Example 2b. duplication

The purpose of the **duplication** action is to duplicate a selected object elsewhere on the display. Before we may start the initiation phase of this action we must have selected an object. The initiation and articulation phases simply determine the position on the display which the duplicated object will occupy. When the articulation phase is over the client goes through the duplication process just as it would in a single-user system. That is, a new object is created and it is given the attributes of the selected object, the newly articulated coordinates and then it is **inserted** into the DS (the inner state is updated) and drawn on the display (the outer state is updated). The **insertion** operation is entered at the completion phase since all the work of object creation has already been done.

Example 3. synchronize_image

The purpose of this action is to make inner images and display context consistent between users. The first use for an action of this type is to put the users into the the same state at the beginning of a conference after one of the users has already read a file into the CIS. This action has implicit initiation and articulation phases. Image synchronization begins in the completion phase. Figure 28 shows the message passing diagram corresponding to this action. The action is initiated by the client who sends out a **Request_Synchronization** message,

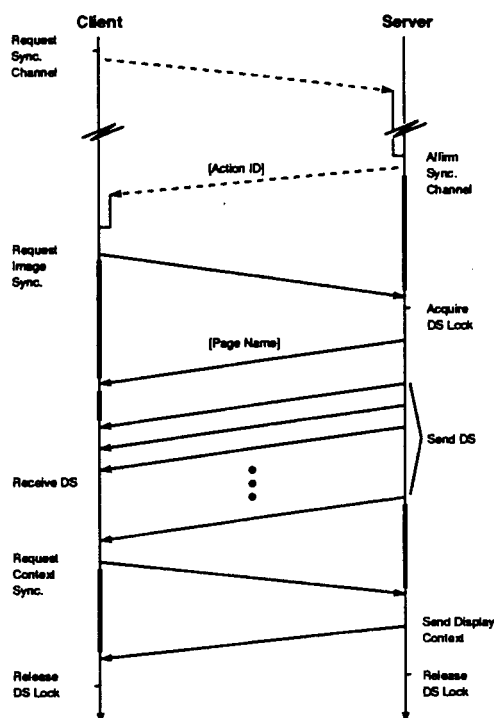


Figure 28. MPD for the Image Synchronization Action

queues the action and waits for a reply. When the **Affirm_Synchronization** arrives the client sends a **Synchronize_Image** message to the server. The server, who has been blocked since sending the **Affirm_Synchronization** message, receives the **Synchronize_Image**, locks its local copy of the DS, sends the name of the image (the filename) and then sends the entire DS to the client. After receiving the entire DS the client sends out a **Synchronize_Context** request. The server then sends the state corresponding to its local display context which has been determined by the most recent local context defining events or the default context.

3.3.3 CIS Connection Protocols

Connection between two users is very simple due to the TIMES package mentioned above. A user first gives a name to his portion of the CIS; the user *session id* (the name defaults

to the user's login id). This name is prepended to the name of each of the three windows shown in Figure 23. He then *EXPORTS* his windows to the distributed name server. The *EXPORT* procedure sends the window name along with the window *address*. The window address contains the internet address of the workstation and the (dynamically determined) port number for that window. The name server broadcasts this information to all other name servers on the network. Another user only needs to know the name, or *session id* (usually the original user's login id), of the exported windows in order to *IMPORT* those windows from his local name server. The *IMPORT* procedure returns the internet address and port number of the named windows. Once each user has exported his own windows and imported the other user's windows, inter-window communication may begin. The name server is no longer needed at this point. Note that for this protocol to work each user must have a unique session id. The Event Manager, now aware that a connection has been made, will send locally generated events to the other user and will 'look' for remotely generated events sent by the other user. When the session terminates normally, the local CIS will automatically *UNEXPORT* its user's windows. This causes the distributed name server to remove those window names from its internal tables. No one can now attempt to conference with that user. If the other user remains active then he will soon be informed that the connection has been lost and the EM will clean up accordingly.

3.3.3.1 Maintaining the Connection

Should there be a network or machine failure or if one user simply exits the CIS prematurely the Event Manager of the remaining user must be informed so that it can stop sending locally generated events. Also, the EM can stop looking for remotely generated events. The algorithm for determining remote user presence is shown in Figure 29.

```
Initialization:
    Set Alarm for N. seconds
    Timeouts = 0
```



```

Main Loop:
  If Alarm timeout
    Timeouts++
    If Timeouts < MaxTimeouts
      Send "Are You There?" message on Channel 2
    Else
      Inform Event Manager and User that connection is lost
  ElseIf Channel 2 Message Received from Other User
    Timeouts = 0
    Reset Alarm for N seconds
    If Message = "Are You There?"
      Send "I Am Here" message

```

Figure 29. *Maintaining the Connection*

We can see from the algorithm that the Connection Protocol uses the ISP channel to exchange messages. Any ISP message received on Channel 2 is an indication that the other user is still connected, and the timer and timeout counter are reset. We use Channel 2 instead of Channel 1 because we do not want the extra Connection Protocol overhead to be included within the busy input event loop. If either user is editing then there will be synchronization activity on Channel 2. This indicates to both users that the other is still present as seen in the 'Elseif' part of the algorithm.

If neither user is busy then eventually one will timeout and send an "Are You There?" message. If the other user receives this message before timing out then it will send an "I Am Here" message. Otherwise he too will send an "Are You There?" message. In either case the first user should receive some message from the other soon after it times out.

Because of network and local processing delays the first user may timeout again before receiving a message from the other. The current implementation uses a `MaxTimeout` count of four and a timeout interval of ten seconds. Thus, if a user times out four times in forty seconds without receiving a message from the other user, then it is assumed that the connection has been lost. The timeout count of four was arrived at empirically: if two users are both idle then one user may run up a timeout count of three before receiving a message from the other user. Why a count

of three seems to work and whether or not the timeout interval should be different are questions for future research.

3.4 CIS User Interface

As stated in Chapter 1 "Combining interactive graphics, distributed processing and data base management should appear as simple, to the user, as using a favourite text-processing or picture-drawing package.". This effectively embodies both the goals of WYSIWIS and multi-user functionality. As always, it is only with these goals in mind that we address the user interface. The desirable features and ergonomics of user interfaces in electronic conferencing are discussed in detail in [HiT78] and [JVS79]. For completeness, Figure 30 shows the CIS prototype window as it appears at present.

3.4.1 Graphical Functionality

We have not attempted to create a general system for building computer conferencing systems with user-defined graphical interfaces. Our purpose was to design and implement a specific drawing tool to be used for brainstorming or educational purposes. As such we must address the question of graphical functionality. Just what capabilities do we wish our system to have? This question must be answered in light of our basic design goals. Now our design goals diverge from the principles behind conventional drawing systems because of the distributed nature of the CIS. It is here that we must temper functional desirability with reality.

3.4.1.1 Objects and Operations

The object types currently supported by the CIS include lines, rectangles, circles and text. Implementation of other object types such as arcs, ellipses and polygons would be relatively straightforward. Spline implementation would be less straightforward. Partially implemented is the ability to do bit manipulations. Individual pixels may be turned on or off, but subregions of

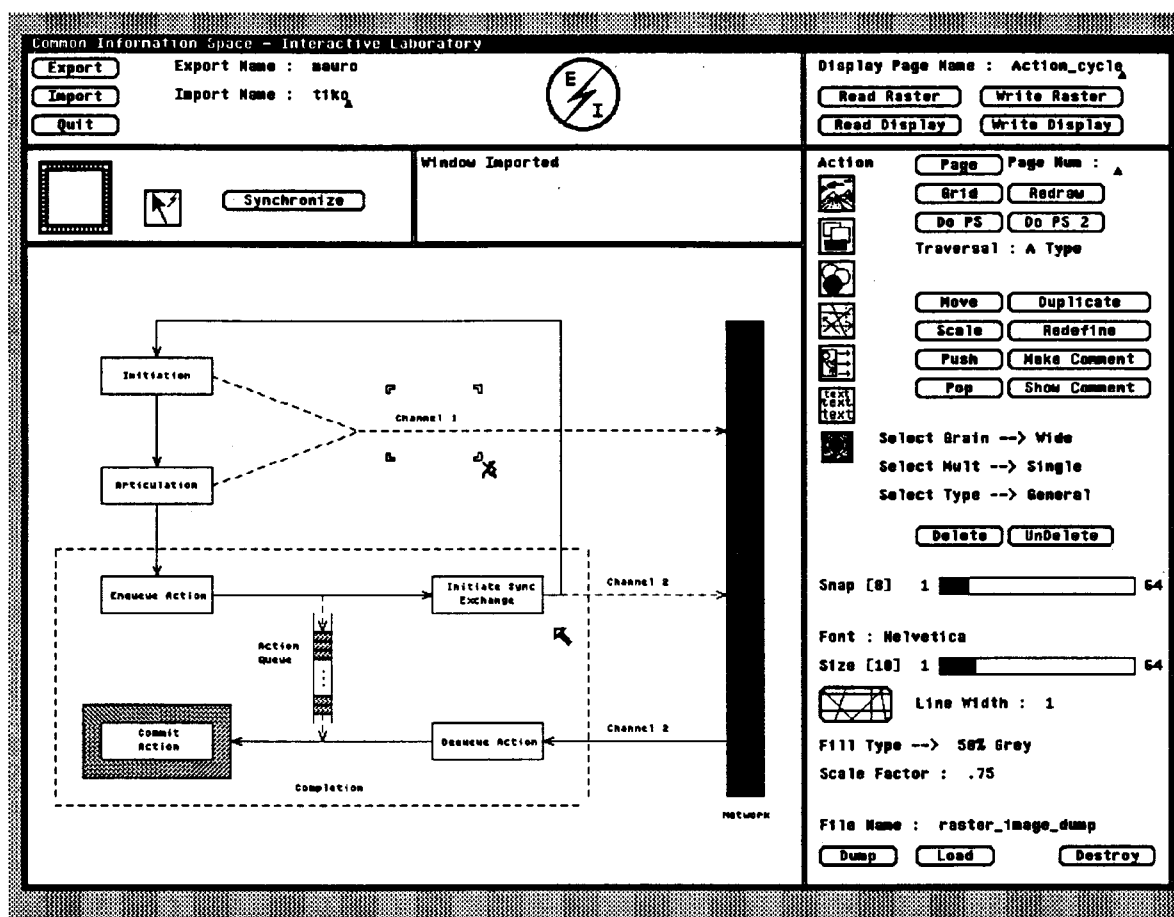


Figure 30. The CIS Display

the display represented by bit maps have not been implemented, as objects, into the itree abstraction.

The default mode for the object editor is **insert**. Objects are **deleted** and **undeleted** using the object transparency scheme mentioned earlier. Other supported operations (action modes) include **move**, **duplicate**, **scale** (partially implemented), **redefine**, **hide**, **expose**, **select**, **shift**, **make_comment**, **show_comment** and **undo**. Most of these operations have already been described or their function is self-evident. The **redefine** operation regenerates an object in the context of the current values of the relevant object attributes. For example, **redefining** a selected

text object will set that object's font type or point size to the current values. **Make_comment** allows a few lines of object-sensitive text annotation to be created for a selected object. This text may be viewed by selecting the object and pressing the **show_comment** button. The annotation text appears in a box to the lower right of the object MBR. The **shift** operation allows the entire image to be shifted on the display. This allows for repositioning of the image within the display or for creating images which are larger than the display.

All operations discussed so far are object-oriented. There is also a set of page-oriented operations which have an effect on the display activity as a whole. These operations include **page**, **redraw**, **overlay** and **destroy**. The **page** operation 'turns' the display page; the current image is removed and a different itree corresponding to the new page is displayed. Currently, there are eight pages available. Action mode and attributes are maintained for each display page. The **redraw** operation refreshes the display from the corresponding itree. If we **destroy** an image then the itree structure is deallocated, the display is cleared and certain state variables are reinitialized. In the current implementation **destroy** only affects the local image.

One design feature that bit images or *raster* images have that is not available with itree structures as yet is an **overlay** operation. That is, we simply choose a display location for a raster file bit image (these files are discussed below) and then read it into the display. The pixels of the new image overlay the pixels currently in place. The overlay operation combines the source and display pixels according to the user-defined *overlay logic*. There are four possibilities: we can simply replace the display pixels with the raster image or we can combine the source and destination pixels using the logical operations *OR*, *EOR* or *AND*. Using the overlay we may 'compose' bit images from a library of user-constructed images. At present, overlay operations cannot be undone.

We may create combination itree/raster images as follows. First create the desired itree image using the standard objects (and save it). Then add pixel information to get the desired final

image by using display bit manipulations or by reading bit images into the display from a raster file. Finally, save the entire image as a **raster file**. If we wish to modify the image we first read in the raster file. Then we may insert/delete *new* standard objects or bits into the image. Standard objects created during the first session are now simply collections of pixels and may only be accessed as such.

3.4.1.2 Attributes

Like operations, attributes come in two flavours, object- and page-oriented attributes. Object-oriented attributes include the self-explanatory font type, point size, line type, line width and fill type. A line segment may take on any of the basic line types including solid, dashed, solid arrow and dashed arrow. Rectangles and circles may be solid or dashed and take on the current fill type. Also, we consider the selection parameters granularity, multiplicity and type to be selection attributes.

Page-oriented attributes include grid, snap and traversal type. The grid attribute is a toggle which draws or undraws a square grid on the display. Snap rounds x- and y-coordinates off to the nearest coordinate which is a factor of the snap value.⁵ This is useful for aligning objects on the display. Finally, traversal type determines the way an itree is traversed during redraw and image-output operations; either family-first or neighbour-first. This affects the order in which overlapping objects are redrawn and hence, the overall appearance of the image. We will discuss itree traversals more in Chapter 4.

3.4.1.3 Functionality and Distribution

We will discuss the CIS implementation vis-a-vis the distribution of functionality by considering an example. What we hope to demonstrate is that simple actions in a single-user system

⁵ In traditional graphics terminology what we call "snap" is called "gridding".

may have to be redesigned when implementing them in a distributed environment. The example will indicate the nature of the decision-making process needed when determining how to implement certain CIS activities. In the example we consider an aspect of the **undo** operation in the context of our real-time distributed graphics editor.

First, recall that the internal image representation has an effect on the way images are processed locally. One of the questions that must be asked when deciding on the itree node structure concerns the representation of display coordinates. The first implementation represented each object in terms of its absolute position on the display w.r.t. the **display** origin (both MBR and object-specific coordinates). If the image was **shifted** then each object in the itree had its coordinates recomputed in terms of its new position w.r.t. the display. The advantage of this was that any time an object was accessed no transformation needed to be done on its coordinates in order to carry out an action (e.g., redrawing or computing the MBR of the entire image). This would be satisfactory as long as the frequency of shifts was small. If, however, we wished to **undo** the operation then every object would have to have its coordinates recomputed back again. Since **undo** is carried out simply by pressing the right mouse button after an operation, clicking the button several times in a row after a shift would cause an annoying delay. When this action is then considered in terms of the completion phase for the distributed CIS the delays become unacceptable.⁶

For these reasons we introduced the traditional notion of "world coordinates" into the implementation. The world coordinates give the real origin of the image in terms of the display origin. All object coordinates are then set w.r.t. this origin. Then, whenever an object is created or accessed, its internal coordinates must be transformed w.r.t. the world coordinates in order to arrive at its absolute position. A **shift** operation would then cause only the world coordinates to be transformed; objects would then have their new display coordinates computed in terms of the

⁶ The undo action for shift operations has not been implemented.

new origin. **Undoing** this operation would mean restoring the world coordinates to their previous value.

3.4.2 File Maintenance

In the CIS implementation we have not attempted to address the questions of file access rights, file consistency, multiple files, etc. In the two-user system the main problems will occur when an image is written. If both users are using the same directory for their session then one user may overwrite the other's copy of the file. If the images have not been recently synchronized this may lead to undesirable results. We may overcome this by including the user session id in the filename. Since we have seen that session id's are unique, writing images to disk will always produce different files for each user. The problem that arises here is the same problem that arises if the users are working out of different directories. That is, having possibly inconsistent multiple files representing the same image. Since the current implementation only supports two users, these problems should be easily reconcilable.

3.4.2.1 File Types

At present the CIS supports or partially supports three input file types and seven output file types. Each file type is composed of the user-supplied Display Page Name and a special suffix. For the purposes of discussion we will assume that we are working with a file called Foo (what else?).

The Foo.cis File

The Foo.cis file is the actual itree internal representation of the Foo image. A pre-order traversal of the itree is made and each node arrived at is stored in the Foo.cis file. Not only is the image part of the node structure stored, but the subtree pointers are stored as well. These pointers are needed when the itree is reconstructed during a read (actually, we only need to

to know whether or not the pointer is NULL).

The Foo.rast File

Currently the raster dump operation takes a selected rectangular area of the image and saves it as a standard Suntools raster file. The format of this file is well defined and is used by other Sun utilities. For example, the `screendump` system program yields a raster file of the Sun workstation display. This utility was used to generate Figure 30. Once generated, the image may be read into the CIS and manipulated on the bit level. Images read in as raster images have no itree structure associated with them and therefore cannot be operated on by any of the standard operations. A raster image is simply an array of pixels each of which may be turned on or off. In the current implementation the maximum size of the raster image is the same as actual display size.

The Foo.iis File

The `Foo.iis` is a special bit map file of a format used by the computer vision laboratory at Simon Fraser University. The CIS interface for files of this type is extremely primitive. Currently, files may be read in and viewed only. However, the image may be larger than CIS display. In this case the image may be shifted in order to view the entire picture.

The Foo.ps File

The CIS interface provides a method for generating a PostScript file from an itree. The `Foo.ps` file is printable without modification or further processing by any printer which interprets PostScript language input files. These files are particularly useful for proofing diagrams that are to be included in a text document.

The Foo.ips.so and Foo.ips Files

These files are necessary if the image is to be gracefully included in a troff text document⁷. The Foo.ips.so file is a *troff* file which includes the vertical size of the image and name of the PostScript file to be included - in this case the Foo.ips file. The latter file contains PostScript code which describes the image itself, as well as translation and scaling information. The translation is a function of the image width (so that the image will be centered between the margins) and the current vertical position of the text on the page. The scaling factor is determined by the user when the file is generated. These files must be used in conjunction with some *troff* macros which were written by the author. In the text document itself all that is needed is a line of the form

```
.,I "Foo.ips.so" "The Foo Abstraction"
```

This will cause the Foo image to be included in the diagram, with the label "Figure xx. *The Foo Abstraction*" centered underneath, and will create an appropriate entry in the "List of Figures" table.

The Foo.CKP.cis File

The CIS provides an image checkpointing facility which, at present, is specified as a startup parameter. There are two options available. First, the facility may be turned off completely (default is on). Second, the user may specify what the checkpoint interval is (default is 10 seconds). The CIS will write the file Foo.CKP.cis to disk once per interval only if the image has changed since the last write.

⁷ All diagrams in this theses were included using *.ips.so and *.ips files.

4

Observations, Evaluation and Further Research

In this chapter we present a qualitative discussion of the design, implementation and testing of the CIS. The discussion will reveal the strengths and weaknesses of the design, the pros and cons of the implementation and will point out the subjective and objective aspects of the testing procedures. This will lead to some conclusions about the viability of distributed conferencing with graphics interfaces and to some of the areas which may be further researched in order to improve or enhance the performance of such systems. Our discussion will run roughly in parallel with the material presented in the last Chapter.

4.1 Itree Motivation, Rationalization and Analysis

As we have seen, the adaptation of good data abstractions will improve system response time and throughput, thereby helping to preserve the basic goals upon which we based our design. We motivate the design of the itree structure for distributed graphics editing and compare the itree with other candidate structures. Our comparisons will be with respect to common graphics editing operations such as searching, insertion and deletion.

The basic problem with conventional structures is that they are not designed for storing two-dimensional data. The study of convenient data structures to store and analyze two-dimensional data is a research problem in its own right. (For example, [Sam84] has done extensive research into the *quadtree* data structure.) We have found that most of the structures studied degenerated into linked lists under certain collections of input data. We have also found that

many of the structures were designed to **analyze** already-existing graphical bit images. Utilization of still other structures would require making assumptions about the nature of the input data that would not hold in our case. Still others were very application-dependent. Thus, we designed our own data structure which we hoped could be optimized for the CIS; the itree. The previous chapter gave a detailed description of the itree structure and many of the operations defined on it.

4.1.1 Linked Lists and the Itree Structure

If we always insert newly-created objects at one end of a linked list, then the position of an object within the list will depend upon when the object was created, not upon its relationship to other objects or upon its position within the display. While **insertion** and **deletion** operations in a linked list are relatively simple, every object in the list may have to be searched to find a particular object because the structure as a whole does not reflect the outer image. That is, the linked list with insertions at one end reflects the *temporal* relationship among objects while a more desirable structure would reflect the *spatial* relationship of an object within its environment. We will show that we may improve upon the linked list structure w.r.t. searching with minimal degradation of the other operations.

Searching

To improve searching performance there are various methods we could use. One such method is to retain the linear data structure and begin searching at some point within the structure which is determined by a separate map or index, or a hashing routine. Another method is to use hierarchical data structures such as 2-3 or binary trees or hybrid structures such as multi-lists or buckets. Or perhaps we should impose some sort of 'order' on the objects and then maintain the list as a sorted structure. To improve searching performance it is not difficult to see that either the structure of the list will have to change, turning it into something other than a linked list, or

the complexity of other operations will increase due to the extra maintenance overhead.

We saw in Chapter 3 that in order to select an object from the display the user defines a rectangular region which "intersects" the object and then the Display Structure (DS) is traversed to see which objects have Minimum Bounding Rectangles (MBR's) which intersect the defined region. The algorithms which carry out the intersection checks reduce to determining the inclusion relationship between a point, or a pair of points, and an object's MBR. If it is found that a point is **not** contained in the MBR then it will **not** be necessary to check any of the nodes in that node's family subtree - because inclusion is transitive and the objects represented by all nodes in a family subtree are contained in the object represented by the parent. Therefore, if the image is represented by an itree where some nodes have non-empty family subtrees, then many such subtrees may be skipped during a search. If each neighbour object in the itree has a non-empty family subtree then searches may take at most half the time, on average, compared with searches of linked lists of the same objects.

Redrawing

Redrawing from a linked list will always redraw the image in the order (or the reverse order) that the objects were created. If we include extra information within the linked list nodes to mitigate redrawing, or if we build auxilliary data structures to indicate redrawing order then the cost of maintaining the image data will rise. With the itree, as with a linked list, the objects in neighbour subtrees are drawn, initially, in the order in which they were created. That is, the order in which they are encountered in an pre-order traversal of the itree. The redraw traversal visits neighbour, or right, subtrees first, by default, and so it would be possible to create a family object (with fill) that obscures a neighbour object of its parent when it is redrawn. Thus, our implementation includes an option that allows the user to change the order in which subtrees are visited during the redraw traversal. Traversing an parent object's family, or left, subtree first causes all

objects in the family subtree to be drawn before any (descendent) neighbours of the parent are drawn. Then all family objects may be drawn 'underneath' all neighbours of the parent. Furthermore, this increase in redraw flexibility does not involve altering the itree topology in any way. We call a neighbour-first traversal an *A-type traversal* and a family-first traversal a *B-type traversal*.

Unimplemented are A- and B-type traversals of the itree in an *in-order* fashion. Post-order traversals would be all wrong in any case; parent objects would always be drawn *after* family objects, thereby obscuring them completely. Also, B-type in-order traversals would not make any sense for the same reason. Thus, we have three distinct and useful ways to draw an image without altering the topology of the itree and without introducing auxilliary structures (e.g., true 3D graphics).

Each of the traversal types may produce a different image if there are overlapping objects⁸. Again, no such flexibility is available using linked lists unless much more information is added to the list structure.⁹ It is amusing to note that the last object to be redrawn will be drawn on top of any objects which it overlaps and therefore, Figure 31, for example, would be impossible to draw as a collection of four rectangles only.

Other itree operations that affect the way an image appears on output are **hide** and **expose** discussed in Chapter 3. These operations, in conjunction with the traversal type, provide flexibility w.r.t. redrawing that is not available with linked lists. Note that these operations preserve the itree topology w.r.t. the neighbour subtree upon which the operations are being performed. Also, family subtrees remain intact. The conclusion from these observations is that changing the position of a node (and its family subtree) does not have the same effect as a **deletion** operation followed by an **insertion** operation. In other words, the number of

⁸ There are $n!$ ways that n mutually overlapping planes may be arranged.

⁹ Possibly turning it into an itree?

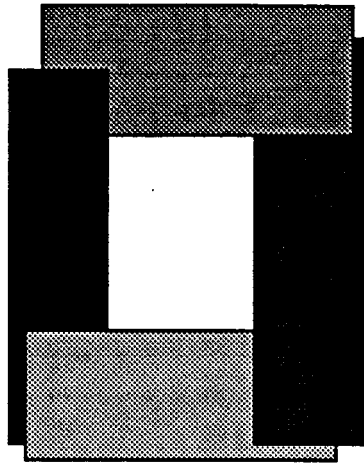


Figure 31. An Impossible Image to Draw?

itree nodes which need to be moved is constant. Again, this functionality may be built into a linked list at the cost of increasing its complexity and hence, the cost of all its other operations.

Finding Minimum Bounding Rectangles

In many cases we may wish to compute the MBR of an entire image. For example, this is useful in order to carry out the **shift** operation mentioned earlier. With linked lists we would have to check every object in the list in order to compute the MBR. With an itree structure we need only notice that the MBR of a family subtree is contained in the MBR of the parent node. So we simply start at the root and find the 'outermost' neighbours in order to compute the MBR of the entire image.

4.1.2 Itree Pathologies

Structure

In the worst case an itree is just a linked list. Figure 32 shows three examples of images whose itrees reduce to a linked list. Figures 32(a) and (b) reduce to linked lists of neighbours and families respectively. In Figure 32(c) the numbers are not part of the diagram but correspond to

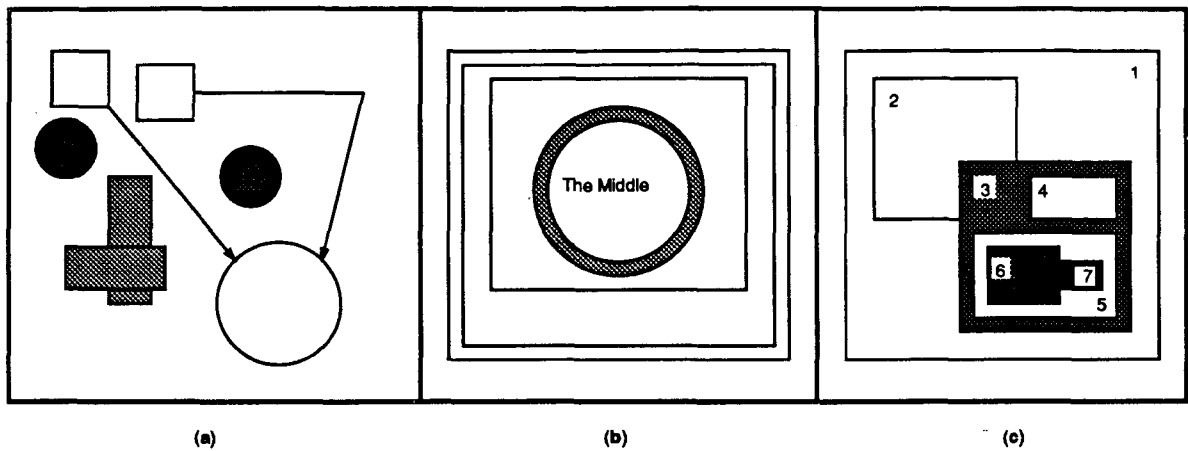


Figure 32. Images Which Exhibit Linked-List Behaviour

the order in which objects are inserted into the tree. In this case the itree reduces to a 'zig-zag' linked list where 1 is the root and each node has only one subtree and the subtree branches alternate between family and neighbour as we move down the tree (try it). Recall from section 3.2.1 that the search algorithm uses points as search keys. In the case of Figure 32(a) the itree structure would be of no use at all in a search. The inclusion or exclusion of the point within a MBR would yield no information about its possible containment in neighbour subtrees. In the case of Figures 32(b) and (c), however, as soon as we reached a node in which the point was not contained in the MBR of the object being searched, we could terminate the search if that node had no neighbour subtree. This will be the case for all nodes from Figure 32(b) and for half the nodes in Figure 32(c). Thus, in some cases, we can expect improved search behaviour using the itree even when it has degenerated into a linked list. Unfortunately, we believe intuitively that most itrees that exhibit linked list topology will derive from images of the type seen in Figure 32(a). Later, we will discuss an enhanced itree that will improve performance even in these cases.

Insertion and Deletion

We call an *itree manipulation* any operation that adds a node to or removes a node from the itree. These manipulations will require changing a bounded number of links (≤ 3) in the tree. Therefore, the number of itree manipulations can be considered as proportional to the time required to perform them. If a new node is inserted as a leaf node then a single itree manipulation is necessary. If the new node contains other nodes then the number of manipulations is potentially $O(n)$ where n is the number of nodes in the tree. This situation will occur, for example, when itree structure is a linked list of neighbours and the new node (call it X) has an MBR that contains all other MBR's. In this case, the root of the itree (call it Y) becomes the one-node family subtree of X , the neighbour subtrees of Y become the neighbour subtrees of X and then inclusion comparisons are made between these nodes and X . Each node will thus be moved from X 's neighbour subtree to Y 's neighbour subtree. Alternatively, we could have left the neighbour subtree of Y where it was and made the comparisons there. Then, if it was found that an object is not contained in the new node, X , it could be moved to X 's neighbour subtree. In this case, however, we may again have $O(n)$ manipulations of the itree. This would occur if the itree structure is a linked list of neighbours and the new node contained only the first element in the list.

Because the deletion operation is in many ways the reverse of the insertion operation, we can see that under certain configurations (i.e., the deleted object contains all the other objects in a linked list of neighbours) this may require all remaining nodes to be reinserted, also leading to $O(n)$ manipulations.

By comparison, linked list structures require constant time for each insertion. No searching for the appropriate location within the list needs to be done; the new node is simply inserted at the beginning (or end) of the list. Deletions will also take constant time but in this case the object will have to be located first, which requires $O(n)$ time on average.

Move

This algorithm operates on the `Current_Selection`. Ostensibly the algorithm involves deleting the object from its place in the itree and inserting it in the new location. However, physical deletions and insertions require potentially $O(n)$ manipulations each. Thus, a move could require $O(n)$ manipulations to take place. As discussed in section 3.2.2 there is a simple method to handle the problem of deletions, but, in theory, insertions may still cause problems. In practice, the worst-case scenarios would often have little to do with the average user response time. The reasons for this are twofold. First, pathological insertions (or deletions) require a situation where (i) the itree resembles a linked list of neighbour nodes and (ii) the node being inserted includes most other nodes. It is difficult to imagine a situation where both of these conditions arise consistently. The second reason that we may not need to worry about pathological cases is that if the itree does not have a large number of objects, then even pathological cases will not take a lot of time to a user. In an experiment, **insertion/deletion** response time of the actual implementation was negligible for images with hundreds of objects in an itree organized as in (i) and (ii) above.

By contrast, a move operation in the linked list structure is painfully simple. Once the chosen object is found in the list, the fields describing its location and MBR in the Display Structure node are altered to reflect its new position. Again, every node in the list may have to be searched in order to find the desired object.

Hide and Expose

These two operations do not come without a tradeoff in other areas. In particular, the **deletion** algorithm would have to be modified. Currently, the deletion algorithm reinserts the root and the neighbour nodes of the deleted node's family subtree starting at the position in the itree where the deleted node used to be (refer to section 3.2.2). Because the **hide** and **expose** algo-

gorithms rearrange neighbour subtrees, the starting point for the reinsertion must change in order to preserve inclusion relationships. Specifically, the starting node should be at the highest ancestor of the deleted node which is also a neighbour of the deleted node. To find this node, start with the deleted node's parent, if the parent is a neighbour of the deleted node, and traverse parent links up the itree until a node is found whose parent is not its neighbour.

4.1.3 Improving Itree Operations

While there does not appear to be any obvious way to optimize **insertions**, there may be a way to avoid costly **deletion** operations. As mentioned in section 3.2.2, each node contains an *object-visibility* flag. With this flag set the object is *visible*. When it is reset the object becomes *transparent*. When the itree is traversed to redraw the image, transparent objects are passed over and not drawn. We call this effect *object transparency*. To the user, a transparent object is 'deleted'. Thus, deleting an object consists of first locating it, then resetting the object-visibility flag and finally undrawing it. No topology-altering manipulations need to be performed. Furthermore, if we maintain a list of the transparent objects then we may easily recover 'deleted' objects. We will call such a list a *release list*.

The major disadvantage of the object transparency scheme is that the itree may become 'loaded down' with transparent objects. The effect of this would be to degrade the performance of all subsequent itree operations. We propose two methods of dealing with this. Method I would put an upper bound on the number of transparent objects that may exist within an itree at one time. The release list would be maintained as a FIFO list. Deleted objects would be inserted into the release list until the the number of deleted objects reaches the upper bound. The next deletion would cause the object at the end of the release list to be removed and physically removed from the itree. The newly deleted object would then be placed at the beginning of the list. Of course, physical removal operations would be subject to the pathology mentioned above.

Method II would be to have no upper bound but would periodically rebuild the entire tree without including the transparent objects. This method would have the same complexity, per deleted node, as the pathological deletion. In a user friendly interface such rebuilding should be transparent to the user. Therefore, rebuild operations should be triggered by some event. For example, every time the tree is saved to disk or the images are synchronized, the tree could be rebuilt prior (or subsequent) to the data transmission. Or the rebuilding could take place when the number of transparent objects (or when the ratio of transparent to visible objects) reached some upper limit.

With Method I we could control the amount of 'dead weight' within the itree, thereby minimizing the negative effect that transparent objects have on itree operations. Furthermore, it provides an effective method for recovering deleted objects. The advantage of Method II is that it may be possible to choose a time that does not conflict with interactive user operations in order to perform the potentially expensive updates. Determining an optimal upper bound for Method I and/or an optimal upper limit for Method II and/or an optimal time to rebuild in Method II is a topic for future research.

Another area of future research concerns a question we have not dealt with at all in the implementation. That is, is there an efficient algorithm for rebuilding the itree after deleting nodes while preserving the overlap relationship among the objects? To understand this question it is important to note that deleting an object using the algorithm described above may alter the itree in such a way as to change these relationships. This would clearly be unacceptable.

Tied in with the question raised in the previous paragraph is the question of tree balancing. Is there an efficient method of balancing an itree which preserves inclusion and overlap relationships among objects? If we refer to Figure 19, for example, we see that the depth of nodes 4 and 6 has increased by one from 19(b) or (c) to 19(d). All three itrees represent the same collection of objects, but with different orders of insertion. Therefore, it is clear that balancing is

possible. A deeper, more general question asks: Is there some intrinsic relationship among the itree objects which is understood by the user and reflected in the itree structure which would be destroyed by tree balancing?

We may also improve the Image Synchronization Protocol (ISP) performance by taking advantage of the itree's searching capabilities. Recall that locking mechanism in the Commit Action subphase of the selection action required that the client send the *name* of the selected object to the other user to be used as a search key in an itree search. We noted that in this case the itree structure itself could not enhance performance. We may improve on this behaviour by including a point from the selected object's MBR in the client's lock-request message. Then the server could expedite its itree search by using the point as a degenerate rectangular region. When an appropriate object is found its name could then be compared to the key.

4.1.4 Other Display Structures

The itree as discussed above has been implemented and appears to perform well. As we have seen, the itree will improve object search times except, in some cases, when the itree degenerates into a linked list. We would like to improve upon the search capabilities without compromising other aspects of the DS. To do this we will describe an *enhanced itree* which always gives improved search capabilities, even in pathological cases.

The enhancement to the itree structure involves 'splitting' the neighbour subtree into two subtrees, a middle and a right subtree, and leaving the left, or family, subtree as it is. The middle subtree contains objects whose MBR's intersect the parent MBR and the right subtree contains objects whose MBR's are disjoint from the parent MBR. We call the middle subtree the *relative subtree* and the right subtree the *neighbour subtree* of the parent.

Recall that the itree speeded up searches because if it was found that a point key was not contained in an object's MBR then we could eliminate the object's family subtree from the

search. Recall that the point in question could either be (i) one of the opposite corners of the rectangular region search key or (ii) a degenerate rectangular region consisting of a single point¹⁰. We call the search in case (ii) a *single-point search*.

First consider case (ii). With the enhanced itree if it is found that the point is contained within the MBR of an object then we may eliminate the neighbour subtree from the search. Therefore, at every node we may eliminate one subtree - either the family or the neighbour - from the search in this case. Now consider case (i). In the case of a search operation if it is found that the rectangular region (both the points at opposite corners of the rectangle) is contained within the MBR of an object then again, it would not be necessary to search the neighbour subtree. If the rectangular region does **not** intersect the MBR of the object then we may eliminate the family subtree from the search.

Recall that the original itree could not improve linked list behaviour for images like those in Figure 32(a). From the above discussion we can see that if we use single-point searches in these cases, then an enhanced itree will yield improved behaviour on average.

Can we continue refining the itree by splitting subtrees in order to improve overall performance? Initial study indicates that this will not be the case. To be sure, we may continue splitting subtrees based on the spatial interrelationship among objects. However, we found that simple extensions to the (enhanced) itree involved at least doubling the number of subtrees needed. While this will almost always result in increased search speeds, the complexity of insertions and deletions increases considerably. At this point it is not clear at all whether or not the trade offs involved will yield any significant improvements.

Going in a completely different direction we may wish to study alternative structures. For example, with *point quadrees* [Sam84] each node has four subtrees. We would need to iden-

¹⁰ It should be pointed out that during the implementation and testing it was found that the 'single point' search was extremely practical. We need only place the cursor in the 'vicinity' (within the MBR) of an object and click the mouse button once in order to

tify a unique point within each object (i.e., the centre of a circle, midpoint of a line segment). This point should divide the rectangular region in which it resides into four quadrants. Each quadrant corresponds to a subtree. The initial object will divide the display into four quadrants. The next object will divide one of those quadrants into four and so on. When inserting a new node we start at the root and check into which quadrant the new object's unique point falls. We move down the quadtree subtree which corresponds to the quadrant which contains the point. We continue recursively in this manner until we reach a node where the next appropriate subtree is empty; then the object is inserted there. The subquadrant occupied by the new point is further subdivided into four quadrants.

As we can see, insertions will be very efficient with such a structure. The maximum depth of the subtree will be proportional to the logarithm of the minimum Euclidean distance between all pairs of unique points. We see, however, that the worst case topology is still a linked list. Furthermore, deletions are apt to be messy: what do we do with the four subtrees of a deleted node? Finally, point searches seem to lose their meaning. Unless we actually start with an object's unique point as the key, then the point we do choose, on the object's perimeter or within the object's MBR say, will have no relationship to desired object's position within the tree. That is, the chosen point may ultimately be in a different subquadrant from the unique point. Ironically, point quadtrees fail most dramatically for point searches. Also, it is not clear how we are going to deal with object overlap. Whether or not we can adapt such structures to an interactive graphics environment such as the CIS is still an open question.

Yet another approach to storage structures involves buckets. We divide the display into n by m rectangular regions. The regions are represented by an n by m array of pointers to a linked list of pointers to objects. The objects themselves may be stored in a linked list. An object is represented in such a linked list if its MBR intersects the corresponding region (also a linked list

begin the selection process. We can pick the point intelligently to facilitate the search.

structure in the worst-case). Inserting an object would be a question of determining all regions that intersect the new object's MBR ($n \times m$ regions in the worst case), and inserting a pointer to that object in the list. Similarly, deleting an object would mean finding all regions that intersect it and removing the object pointer from the list. In this case we would need to search each region list in order to find the correct object, with a worst case of $N \times n \times m$ operations where N is the number of objects. Point searches could be done efficiently because the correct region could be computed directly from the point's coordinates. Again, overlapping could be a problem.

There are numerous modifications, enhancements and improvements that may be applied to the bucket approach as there are to the point quadtree approach. These changes may be analysed analytically or they may be implemented and tested for response times under various conditions. The latter approach is favoured because in this case, if response times do not vary a great deal from one method to another and one approach strongly outperforms others in some areas (such as simplicity of implementation) then this is the approach we should adopt. In other words, sometimes the worst case behaviour of an algorithm does not translate to unacceptable behaviour in a practice.

4.1.5 Itree Consistency

One of the questions we were to address was that of itree consistency: to what extent can we tolerate inconsistent itree structures and still preserve our basic design goals? Put another way: if it turns out that maintaining consistent itree structures in real-time is critical to the overall functioning of the CIS, how may we relax the constraints imposed by our basic design goals, if necessary, in order to achieve this? We have seen that the itree structure affects its representation on the display if there are overlapping objects. Thus, inconsistent itrees could lead to different displays, which violates the goal of WYSIWIS.

If we have implemented the *enhanced itree* mentioned above, then will we be able to

tolerate order inconsistencies between two lists of neighbour subtrees? At first it would appear as though this may be the case because nodes in these lists correspond to objects which do not overlap. Since a newly created object may be contained in at most one object in a list of neighbour subtrees, insertions would not be affected by order inconsistencies. If the new node contained objects in the list of neighbour subtrees then all these objects would be stored as neighbours in the new object's family subtree.

Now suppose that we have two non-intersecting objects, A and B, and that the position of these two objects is reversed in two different itrees. Also, suppose that we create a new object which is a relative to each of these objects. Then the new object would reside in the relative subtree of object A in one itree and B in the other. If we now redraw, traversing relative subtrees before neighbours (a B-type traversal) then we will end up with inconsistent displays. This situation is shown in Figures 33(a) and 33(b). If instead we traverse neighbour subtrees first (an A-type traversal) then the display will look like Figure 33(c) in both cases. Thus, we could maintain outer consistency if, in the distributed CIS, we allow only one traversal type. This, however, violates the goal of multi-user functionality.

Another place where itree consistency may play an important role is in deadlock prevention when locking a group of objects. (This action is not implemented but is accounted for in the design.) One efficient method for preventing deadlock is to have an ordered locking scheme [Had68]. In this scheme resources (objects) are given unique priorities and in order to obtain a lock on a group a client must obtain the lock on higher priority resources before lower priority ones. If a client finds a resource locked then it has the option of either waiting for the lock to be released or releasing all the locks obtained so far and trying again later. This scheme prevents deadlock because two clients cannot each be holding a lock that the other requires. For one of the resources must have a higher priority than the other and only one of the clients may hold the lock on it. Thus, the other client could not have passed this point in the locking process and so could

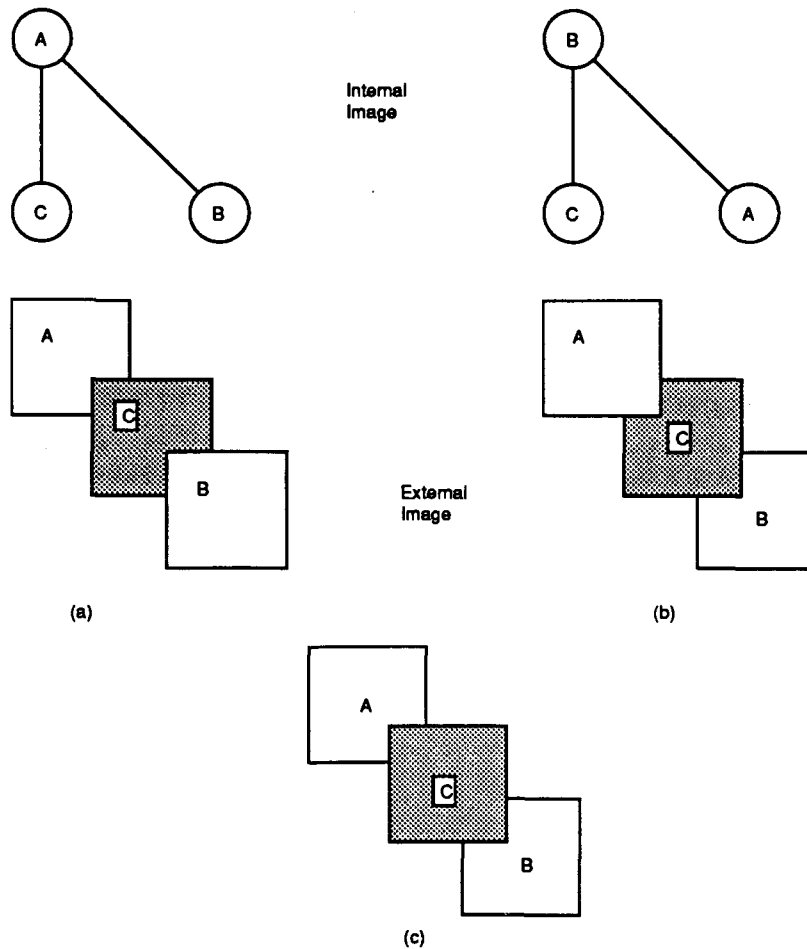


Figure 33. Inconsistencies in an Enhanced Itree

not have gained the lock on the lower priority object. In order to implement this scheme in the CIS we need to 'priorize' or linearize the itree objects. This could be done simply by a certain traversal of the itree. Now we can see that in order for this to work at all the itrees must be consistent.

It should be noted that by having a client wait we again undermine the goal of multiple user functionality. Even if we implement a 'reservation' scheme¹¹ where the user does not have to block while waiting for locks, this goal will still be violated.

With the current hardware and software technologies, multiple object manipulations in a

¹¹ In this scheme a user puts in a reservation for an object or group of objects and then goes about other business. When the

distributed environment cannot be carried out with the same functionality as a single-user system if we are to maintain inner and outer image consistencies. It appears that in order to maintain one of our basic design goals we must trade it off against the other. Consider the previous example. If we wish to maintain multiple-user functionality then we should not have users waiting for distributed actions to complete. In this case, each user would continue as if he were a single-user. This may violate WYSIWIS because of the inconsistent images which may result (albeit temporarily). This means that actions that conflict must be resolved at a later time. We must have a way of backing out of or undoing a series of actions. Thus, a user may find his display changing to reflect the global decision making policy. This violates multi-user functionality. And so on...

We will see more examples of these trade-offs in the next section.

4.2 Multiple Images

The very existence of multiple images appears to violate the goal of WYSIWIS; if you are editing a different image then What You See Is NOT What I See (WYSINWIS). However, the high-level concept of a common information space is still intact. With multiple images the *extent* of the shared space has increased beyond the physical display capacity of our system. We now use the display as a window into the CIS. The goal of WYSIWIS exists now as a potential. We say that the goal of WYSIWIS has been *relaxed*.

As we saw in the last section, we needed to trade off each of our basic design goals against the other in order to obtain the desired functionality in different areas. If there are times when we may relax WYSIWIS then perhaps we can regain those parts of our multi-user functionality which we traded off. For example, if each user is editing a different page then there need not be any question of a user waiting for distributed actions to complete. That is, there is no immediate need for the users to have consistent inner images (the displays will be completely dif-

reservation can be honoured, the objects are locked and the user is informed accordingly.

ferent so there is no question of outer image consistency). Here are some more examples.

- To obtain the lock on an object it is only necessary to check that the object is not locked locally before operating on the object - there will be no chance of contention if the other user is operating on a completely different itree structure. The locking request will still be sent out just in case the other user returns to this page before the operation is complete. To further streamline this procedure we could require that all locks be released when a user turns the page.¹²
- If the other user is not viewing the same page then it will not be necessary to distribute Channel 1B input events. This will decrease communications and local processing overhead, thereby enhancing the response time for both users. This further supports the goal of multi-user functionality.

It should be pointed out that the positive aspects of goal relaxation may have to be traded off again if we increase the functionality of the CIS. Two ways that we might increase functionality would be to introduce inter-page overlays and multiple simultaneous displays such as split screen or multiple-window viewing. Inter-page overlaying is simply object duplication across page boundaries. Selected objects from one page are duplicated on another page. This implies that a user could hold locks on pages which that user is not viewing, which in turn implies that the ISP would have to be maintained across a set of itree structures.

With split screens the user would be able to divide the display into two or more smaller non-overlapping displays each with an outer image of an itree associated with it. Multiple-window viewing would allow the user to create two or more overlapping windows each with a view of some itree. In either case the implementation would be considerably more complicated and we would again be in the position of requiring that WYSIWIS be observed within any image that all users were editing concurrently.

¹² This may or may not be a good idea. If there are logical connections between objects on different pages, or inter-page operations then there may be good reasons for locking an object before turning the page, and retaining that lock after the page has been turned.

4.3 Non-Blocking Completion

If we refer to Figure 8 in section 2.4 we see that the client returns to the top of the Basic Action Cycle after requesting synchronization with the server. From Figures 27 and 28 we see that when the server blocks in order to carry out synchronization the client is not yet aware that this has happened. Should the client itself block indefinitely for other reasons (like responding to a system prompt) then the server will also block indefinitely. This runs counter to the goal of multiple-user functionality. We propose three options which may be incorporated into the current implementation.

The first option has the client unable to perform any blocking operations while it has synchronization requests outstanding. This will simplify the CIS process architecture but may impose severe limitations on the graphical functionality because actions that require user interaction¹³ would be prohibited until the request has been serviced. Not only would this limitation violate the goal of multi-user functionality but it would require that all activity within the system be filtered to determine whether or not blocking is possible. This in itself does not appear to be a reasonable functional specification. The second option involves implementing the Completion Manager as a separate *client* process in order to carry out the completion phase. The CM would carry out the completion phase concurrently with whatever else the client was doing. Therefore, if the client blocks then the CM could carry on with the synchronization. The third option is similar to the second except that the *server* implements a separate Completion Manager process. Then only the server's CM process would block. The second and third options would complicate the CIS process architecture, thereby increasing system overhead and local communication costs. However, the drawbacks of the first option clearly outweigh the disadvantages of the second and third.

In section 3.2.4 we briefly discussed the problem of access to primary CIS data abstractions. In view of the above discussion it does not appear that the CIS DS should reside in the

¹³ Such as a system prompt for user feedback during the selection process.

CM. Thus, it would seem reasonable for the Image Data Manager to maintain the DS. The current CIS is implemented as a single process. The reason for this is performance. To divide the system into concurrent processes will obviously slow some parts of the system down because of the extra context switching and local inter-process communication involved. The indefinite blocking state described above was never experienced. This was probably due to the relatively simple implementation of the completion phase and the luck of the draw during testing. It is clear, however, that to address some of the problems discussed above we may have to trade off some performance in order to prevent the indefinite blocking described above. Optimizing the modularization in terms of process functionality and responsibility is another question for future research.

4.4 Distributing IDP State Information

As we saw in section 3.3.1 there was a certain degree of awkwardness in distributing Context Defining Events. The initial implementation led to incorrect and inconsistent state due largely to the vagaries of the lower level support. The solution to the problem introduced some inefficiencies into the system but provided some generality which made extensions relative simple to implement.

The reason for this was because it seemed more expedient to send the event to be interpreted remotely by exactly the same routines that interpreted it locally. The idea was that identical treatment would lead to identical states; that a single event would be 'duplicated' and processed concurrently in each CIS by a common set of routines. This is still the method that is used to process Action Events generated during the initiation and articulation phases. However, while the CIS maintains local and remote state the underlying system, Suntools, only maintains local state. Like any good state machine the next state is determined by the input and the current state. Therefore, remote input is quite meaningless to the local Suntools and the resulting Context Window updates reflected this. In other words, the new state was a combination of the current local

state and the remote input; this led to corruption of both the local Suntool and remote CIS states. Thus, after each such event two actions were performed. First, the remote CIS updates its Suntool-managed state from the CIS data structures and, second, a *context-update* message is sent to the remote machine to update the CIS remote state data structure.

A better way to deal with the problem would be to have no CDE's sent at all. Instead, changes to local context state would be followed by a context-update message only. This is already done in some cases, as mentioned above. The only disadvantage of this approach is that it makes the system less easily extensible - and this would not be a problem if enhancements were infrequent. The main advantage to such an approach is that it takes the system as whole one step closer to machine independence. Also, because the CDE's themselves would no longer need to be sent, we could expect an improvement in the overall performance of the system.

4.4.1 Maintaining Remote State

In Chapter 3 we saw some of the advantages of maintaining a (relatively) stateless server. The discussion in the previous section indicates some of the problems that may arise trying to maintain distributed state. In this section we will discuss the possibility of further decreasing the amount of state that needs to be maintained about the remote CIS.

As we have seen we may divide up the distributed state into six categories. Of these six the graphics and text drawing data are generated by AE's. This information is needed only while an action is in progress. Afterwards the server may 'forget' it. Consider the creation of a new object, for example. A newly created object will be passed to the server during the Completion Phase. All object attributes are contained within the object structure. The server does not need to know anything about the state of the remote CIS. However, after the object has been articulated it needs to be displayed on the outer image. To do this we need the object attribute data. Either these remote attributes must be maintained locally or they must be sent along each time the articulation phase ends. Not only will the latter result in more messages being sent but it will mean

more local processing at both ends in order to implement the protocol for the attribute exchange. Similar comments would hold for the selection operation and selection attribute data.

In general, information now retained as remote CIS state may be updated just before an operation that now uses that state and forgotten immediately afterwards. This would result in a completely new Image Data Protocol. The Basic Action Cycle would need to be redesigned to take into account this new ordering of events. This in turn would lead to a restructuring of the MUBAC. In other words, the maintenance and distribution of system state affects all dimensions of the CIS - just as we would expect; this is the essence of distributed software.

4.5 Connection and the Distributed Name Server

In the CIS implementation connecting with another user is extremely simple. The reason for this is the TIMES package, mentioned in the last chapter, and the presence of a Distributed Name Server (DNS). As we have seen, connection can be established with the other user by first entering the session id of each user and then pressing the **Export** soft button followed by the **Import** soft button (see Figure 30). The **Export** and **Import** buttons send and receive (resp.) internet addresses and port numbers of the CIS windows to/from the DNS. Since we are using UDP sockets there is no need for any further explicit connection activity.¹⁴

Note that attempting to import another user's windows before they have been exported will obviously meet with failure. We may improve upon this situation by having the connection protocol continue to attempt import until the other user does export. In the meantime, the importing user could continue image editing. To this end we could remove the **Import** and **Export** buttons and replace them with a single **Connect** button.

¹⁴ Connection using TCP sockets would be more complicated to implement but the procedure would be the same at the user level. The extra connection activity would probably make the process a little longer.

4.5.1 The Advantages of a Distributed Name Server

The principal advantage of a DNS is that within a local area network users do not need to be aware of the machine address/name or the port numbers associated with a CIS application in order to establish connections. That is, the network location and local ports of one user are totally transparent to the other. A user need only provide the CIS with a name, which we will call the *session id*, that is known to the other user. The CIS asks the operating system to dynamically assign its port numbers and then sends the session id along with its machine address and port numbers to the DNS. Machine address and port numbers are essential in order to establish a connection. A machine's address is a network constant but since port numbers are dynamically assigned they may change from session to session. *It is important to note that a port number may only be used by a single user (process) at any one time.* If a user knew in advance what the other user's machine address and port numbers were then there would be no need for a DNS. Since the machine address is constant, DNS-less connection may reduce to knowing, in advance, a user's port numbers.

The obvious disadvantage of using a DNS is the existence of the DNS itself. That is, there must be name server running on every machine that wishes to use the CIS for distributed imaging. There must exist network tools for the distributed support and maintenance of the DNS. We will discuss four connection methods that do not need the services of a DNS.

The first method would use hard-coded port numbers. In this case a user would need only specify the name of the machine with which a connection is to be made. The major disadvantage of this method is that if any of the six ports (three on each machine) is already in use then that port is unavailable and no connection can be made. The second method would be to hash user names (or session id's) into a set of three port numbers. The same problem as hard-coded port numbers would apply in this case. The only advantage over hard-coding is that a user could respecify his session id in order to get three more ports. But then the new name would have to be passed on to the other user so that he could also determine the new port numbers. The third

method would be to have system support in a scheme of reserved port numbers. These port numbers would be reserved for use by the CIS only and would probably be stored in a distributed file. The drawbacks here are that the distributed file would need to be maintained and only one user per machine would be able to use the CIS. We could possibly get around this by having sets of reserved port numbers that could be tried in turn. With the fourth method we could have the CIS get dynamically assigned port numbers from the system as is done currently. These numbers would be taken by the user and conveyed by external means (like telephone or UNIX 'talk') to the other user who would input them to the CIS. We won't discuss the drawbacks of this approach. With all of these schemes we would still have to specify a machine name.

All of the alternative methods discussed above lead to unreliable and/or awkward connection protocols. From the user-interface point of view the DNS is the cleanest and most reliable way to establish a connection. We should point out that the The Integrated Message Exchange System supports non-DNS connections. In this case we must specify a machine name and a port in the **Import** call.

4.6 Long Haul Connections

Enhancements for connections between users on different networks could be implemented with little or no change to the existing system. We introduce into the system a *local connection agent* and a *protocol mux/demux* (PMD) process. The **Export** command would be a signal to the local connection agent to establish a single connection with its remote counterpart. When the connection has been established the connection agent would turn communications over to the PMD process. The end result of the connection process would be two PMD processes talking to each other on the one hand and to their local CIS on the other. A CIS would send a message on one of the three channels to the PMD. The PMD would 'package' the message along with a channel identification code and send the new message to its remote counterpart. The remote PMD would check the channel identification of the received message and then forward

the message to its local CIS along the appropriate channel. To the CIS, the message-passing interface in the long haul situation is identical to that in the local area network; the local PMD looks exactly like a remote CIS w.r.t. the message passing interface.

Clearly, the response time will suffer considerably. Not only are there two extra communication links and extra message processing overhead, but the inter-PMD link is likely to be considerably slower than the local area communication (e.g., 1200 baud vs. 10 Mbits). There would be considerable need for optimizations. For example, when drawing a new object only the anchor point and terminal point for the action would be sent. It may be necessary for the PMD to send messages to its remote counterpart only in groups or in certain time intervals.

It is not clear whether or not the system could be enhanced to the point where the delays become acceptable enough to justify the costs involved. The extension to the CIS as described so far do not involve any changes to the existing software. However, it may be necessary to modify the CIS in order to optimize those portions of the system that interact synchronously with the other user. In general, any message received that requires a response should be processed in an optimal way. The relaxation of our basic design goals may reach the point where we will need to re-design for long haul connections.

4.7 Multiple Participant Conferences

Almost every aspect of the CIS, from model to implementation, would need to be re-examined if we introduce the notion of multiple (> 2) participant conferences (MPC's). We would still work with our two basic goals in mind, though we expect that they will need to be further relaxed in order to maintain acceptable performance levels. Our basic model would still be that of a fully distributed system as described in section 1.1.3. We briefly discuss some of the more interesting aspects of a multiple-participant conferencing system (in no particular order). See also [Sar84].

1. Subgroups

With the introduction of multiple participants comes the possibility of conference subgroups. Within a subgroup - where all participants are working on the same image - we would expect our basic goals to be upheld. However, modelling across subgroup boundaries would require redefining these goals with subgroup activity in mind. Furthermore, we would need new goals to govern subgroup-subgroup and participant-subgroup interactions. Refer also to [SBF86]. As well as being considered in its own right, each of the following aspects of an MPC will also need to be considered in the light of subgroups.

2. Communications

Communications within the CIS may be divided into three categories: connection and maintenance; image data protocol; and image synchronization protocol. We would probably need to impose a logical structure on our conference sites, such as a ring or a completely connected graph, and then draw on established distributed algorithms to provide the basis for our protocols.

a. Connection Initiation and Maintenance

Using the distributed name server we could gather address information about conference participants in the same way as the two-user system. We would maintain an array of participant address information which would be used each time a message is sent. For example, either one message is sent to a single participant, in the ring model, or one message is sent to each of the other participants in the completely connected model. Broadcasting is not possible because under the current port number assignment scheme, each site will have different port numbers; each application within a distributed system must have identical port numbers for broadcasts to work.

In order to implement a system that has broadcast capabilities we would introduce the **join** operation into the CIS. Then connection would proceed as follows. A conference would be given a pre-agreed upon name (replacing the session id in the two-user system). A participant

would **join** the conference using this name. The **join** operation would cause the CIS to get three port numbers and register these, along with the given name and broadcast address, with the DNS. Each user would then **import** the broadcast address in the same way as in the two-user system. Note that if more than one participant attempted to execute a **join**, one of them would be rejected by the DNS because the DNS requires unique names. Timestamps would be used to resolve ties. No **imports** would be allowed until all components of the DNS 'agreed' upon the name, address and port numbers.

Connection maintenance could be done in a manner similar to the two-user system by imposing a logical ring structure on the conference. Each site would use the algorithm discussed in section 3.3.3.1 to keep track of its neighbour in the ring.

b. Image Data Protocol

As we saw in sections 3.3.1 and 4.4 the IDP implementation appeared awkward and inefficient - especially w.r.t. Context Defining Events. Clearly, in an MPC such inefficiencies would be unacceptable. The exchange of CGE's in the two-user system involved implicit information contained in the input events, as well as explicit information sent afterward (see section 3.3.1.1). An MPC would probably require that all data exchanges contain explicit information (as discussed in section 4.4.1). The frequency and content of the exchange would still need to be determined.

Action Events would still be sent as is, though the number of these events sent would need to be reduced. This could be done by sending just anchor points (as suggested in section 4.6 for long haul conferences), by sending only every n -th AE or by sending AE's at certain time intervals.

c. Image Synchronization Protocol

The ISP would probably require the most extensive changes from the two-user to the multiple-user conference. The two-user model, which is only partially implemented, relies on the inherent simplicity of these systems. For example, to ensure fair, starvation-free concurrent access to system objects, multiple participant systems would require more sophisticated locking algorithms (as discussed in section 3.3.2.1, Example 1). We could use a global clock to resolve conflicts but we would need more machinery to ensure fairness. This also holds for synchronization channel requests. Also, once a synchronous exchange begins, the one who requested the exchange, the client, must have a protocol for dealing with multiple responses.

Locking granularity itself forms the basis for other areas of future research. For example, would there be any benefit to locking i) entire subtrees? ii) only family subtrees? iii) a region (i.e., a quadrant) of the display? iv) a page?

With respect to i), above, we point out the following. Because the topology of an itree may be dependent upon the order of insertions, the serialization of itree manipulations is critical if we are to maintain itree consistency across the network. Thus, if it can be determined that two synchronization operations are to be carried out on two mutually exclusive subtrees, then it is not critical that the order of these operations be the same at every site. Therefore, the serialization requirements may be relaxed and, hopefully, the amount of distributed processing needed to carry out serialization may be reduced. Of course, both these subtrees would need to be locked.

Special cases also start to arise with the introduction of more than two participants. For example, the **synchronize image** operation requires the active participation of only *one* other participant, aside from the client (refer to section 3.3.2.1, Example 3). Furthermore, it is not clear whether or not the DS will need to be locked on *all* participant sites during the exchange between the client and server. This is because we could block all updates to the DS by having the client and/or the server refuse a locking request during the synchronization procedure.

One major area for future research is the investigation into the semantics of failure. All

areas of the protocol must be analyzed in order to identify and classify the points where failure may occur. Then failure prevention and/or recovery procedures could be added to the protocols.

The many different elements which make up the ISP and connections among these elements bring up many of the classical problems and possibilities of distributed computing theory.

3. Data Structures

The data structures for an MPC would not be significantly different from a two-user system. Since all of the topology altering operations take place synchronously, the DS and the operations on it would not have to change for an MPC. The requirement of itree consistency across multiple participants will be still be critical in order to maintain our basic goals (see section 4.1.5).

4. Graphical Functionality

As we saw in section 3.4.1, some functional elements of the graphics display may have to be re-designed in order to 'fit into' a distributed environment. This concept will carry even more weight in an MPC. This will be particularly important in operations such as **undo**, where a single button-press event can be used to reverse, or cancel, the effects of several, possibly hundreds, of other events.

5. User Interface

As usual, the user interface must not betray the vagaries of the implementation. An MPC user interface should be as simple and straightforward as a two-user conference interface, which in turn should preserve the goal of multi-user functionality. That is, the interface should 'hide' the details of the multiple-user setup and maintenance of the inner and outer images, while at the same time providing the functionality and intuitive ease of use that can be built into a single-user, stand alone program - a tall order. This is not to say that the interface should hide the existence of an MPC. For example, a user should be able to bring up a list of all conference participants. This

list may provide, for example, the name, location and 'conference status' of all other participants.

We have discussed just a few of the important aspects of multiple participant conferences. There are other areas of interest which we have not discussed. These include conference control - having a conference chairman whose function will depend upon the conference type and built-in conference control capabilities of the system; MPC system architecture; file access and maintenance; and conference security - who may participate, who has access rights to what objects or display pages and so on.

5

Conclusions

The fundamental question which must be addressed asks whether or not the idea of electronic conferencing using graphics as the medium of information exchange can be applied in a pragmatic manner while maintaining the basic goals of WYSIWIS and multi-user functionality. The answer is yes.¹⁵ In order to answer this question we developed a data abstraction, called the itree, to provide local storage of graphics images, and a communications protocol to tie itrees together across a local area network.

The itree structure was designed to reflect partial spatial interrelationships among the objects that it represents. Because the objects in a graphics display have obvious spatial interrelationships it was hoped that the itree structure would yield fast, unambiguous user selection of these objects. This in turn would expedite the distributed locking and subsequent manipulation of the graphics display. We demonstrated that itree searches were, on average, very efficient compared to linked list structures (which are often used in graphics drawing tools to store images). This was borne out in the implementation testing where no appreciable delay was noticed for selecting and locking an object across the network. It was also discovered that the itree provides a great deal of flexibility for redrawing a display when the display contains overlapping objects. There are three distinct ways to draw the display from the same itree, and there are efficient operations for changing the overlap relationship among objects. The itree also enables us to compute minimum bounding rectangles for its subtrees very quickly. In particular, the MBR of the entire image may be computed in order to carry out the **shift** operation - a

¹⁵ Along with this answer go all the usual qualifications concerning available hardware and software technologies.

potentially expensive action.

We discovered that itree in the worst case resembles a linked list. In this case there are situations where operations which involve insertions or deletions may exhibit pathological behaviour. However, we found that the prototype performance was not significantly reduced even in these cases. We also found that in many cases the itree would still outperform traditional linked lists. It was noticed that, in general, any increase in the graphical functionality of the system usually meant an increase in the computational complexity of the itree algorithms. This of course leads to degradation of overall conferencing performance. We hypothesize, however, that an appropriate partitioning of the prototype into concurrently executing processes, together with a load sharing algorithm implemented to allocate computationally expensive activities to the least 'graphically active' site, may serve to hide performance bottlenecks from the participants.

We also extended the itree concept by introducing the enhanced itree. The enhanced itree was designed to produce even better search capabilities by refining the spatial interrelationships among objects. Of course, the improved searching performance due to the increased information content of the data structure must be traded off against the increased complexity of itree manipulations.

We conclude that the itree is a reasonably good candidate for distributed imaging applications. Future research may indicate an optimal candidate. Experimentation would have to be carried out on two levels: the first would involve the theoretical and applied quantitative analysis of different data abstractions; the second would benchmark the various candidates under actual conferencing conditions in order to measure the relationship between the computational complexity and the real performance of an implemented abstraction - these relationships would then be compared across the various data abstractions.

Other areas of itree research include the design of efficient algorithms to rebuild itrees which contain transparent objects and algorithms to balance itrees while preserving the spatial and overlapping relationships of the image which are reflected in the itree structure.

In order to distribute the itree over the network in a consistent way, we developed protocols for the efficient construction, distribution and maintenance of these structures. The basic aim of the protocol design was to subject both users' internal state to the same input set, resulting in the same next-state for both users. This was to be accomplished by sending one user's input to the remote user where it would be processed as if it were generated locally. This would lead to consistent inner states and would naturally preserve the goal of WYSIWIS. This will work perfectly in practice if only one user at a time actually manipulates the display.¹⁶ Since this is not the intention of electronic conferencing, this rather naive guideline was subject to extensive modification in some areas.

The basic problems of what information to exchange and when to exchange it motivated the development of two protocols which were implemented over three different communications channels. The first protocol (IDP) is responsible for asynchronously exchanging information about the graphics display as it is being manipulated. This protocol is divided between two communication channels: the first channel is for the exchange of context-defining information (CDE's) and the second is for the exchange of image-manipulating information (AE's). The second protocol (ISP) is responsible for synchronously updating the display after a user has carried out an operation.

We found that sending the input events which were generated by object creation and manipulation actions (AE's) worked quite well in terms of the basic goal. These events are not actually needed to preserve inner image consistency across the network - they are needed to maintain outer image consistency (WYSIWIS). Thus, these events have no meaning outside of the context of immediate object manipulation. It was found that since the low-level graphics support software (Suntools) maintains state about the display, CDE's could not be handled in the same way. After many modifications to this part of the protocol it was decided that the basic protocol design aim could not be preserved for CDE's. Instead, sending explicit context-update

¹⁶ Such may be the case in an educational or demonstration environment.

messages after changes to local context state appeared to be a satisfactory solution. However, we noted that since CDE's are only necessary in order to allow an image to be drawn consistently at both sites after all the AE's have been sent, perhaps the entire IDP should be re-evaluated.

As we mentioned above, the basic aim of our protocol design could be applied in a system where only one participant at a time actually manipulates the display. In Chapter 1 (section 1.1) we pointed out that this was actually done in some cases by using a "vocal lock" or by having participants "negotiate externally by voice". In the CIS we attempted to realize the maximum functionality of an electronic conferencing system by making such 'negotiations' invisible to the user (this is also in line with the goal of multi-user functionality). To this end, the ISP was designed and implemented. The ISP guaranteed inner image consistency by synchronously updating itree structures on each user's copy of the structure. While updates are in progress user activity is curtailed, which could produce delays or interruptions in drawing activity for both users. In experiments, however, the updates did not create any noticeable delays except in the case of image synchronization operations (where the entire itree plus associated context is exchanged) on relatively large itree structures. Since this operation should be quite rare, we do not consider these delays to be serious. We also found that the system cannot tolerate any degree of itree inconsistency if we are to preserve WYSIWIS. Furthermore, it was noticed that itree consistency would be critical for some deadlock prevention schemes.

We have touched on a few of the many issues surrounding electronic conferencing. We have shown that real-time graphics conferencing which preserves the basic concept of a multiple-participant, face-to-face meeting without sacrificing individual creativity, is viable within existing computing environments.

References

[AGN87]

H.M. ABDEL-WAHAB, S. GUAN and J. NIEVERGELT.
Rapid Prototyping of Remotely Shared Workspaces.
Technical Report-87-01, North Carolina State University, January 1987.

[Ado86]

ADOBE SYSTEMS INC.
PostScript Language Reference Manual.
Addison-Wesley Publishing Company Inc., Reading, Massachusetts, 1986.

[BaN81]

A. BALLARD and J. NIGHTINGALE.
A Computer Conferencing Program (UBC FORUM).
U.B.C. Computing Centre Technical Report, November, 1981.

[FoV84]

J. D. FOLEY and A. VANDAM.
Fundamentals of Interactive Computer Graphics.
Addison-Wesley Publishing Company, 1984.

[For85]

H. FORSDICK.
Explorations into Real_time Multimedia Conferencing.
Proceedings of the Second International Symposium on Computer Message Systems, BBN
Laboratories, September, 1985.

[Hav68]

W. HADENDER.
Avoiding deadlock in multitasking systems.
IBM Systems Journal, 7(2), 1968.

[HiT78]

S.R. HILTZ and M. TUROFF.
The Network Nation, Human Communication via Computer.
Addison-Wesley Publishing Company Inc., Reading, Massachusetts, 1978.

[JVS79]

R. JOHANSEN, J. VALLEE and K. SPANGLER.
Electronic Meetings: Technical Alternatives and Social Choices.
Addison-Wesley Publishing Company Inc., Reading, Massachusetts, 1979.

[Lam78]

L. LAMPORT.

Time, Clocks and Ordering of Events in a Distributed System.
Communications of the ACM, July, 1978.

[LaN84]

K. A. LANTZ and W. I. NOWICKI.
 Structured Graphics for Distributed Systems.
ACM Transactions on Graphics, 3(1), January 1984.

[Lan86]

K.A. LANTZ.
 An Experiment in Integrated Multimedia Conferencing.
Proceedings of the Conference on Computer-Supported Cooperative Work, December, 1986.

[NeS79]

W. M. NEWMAN and R. F. SPROULL.
Principle of Interactive Computer Graphics.
 McGraw-Hill Book Company, 1979.

[Sam84]

H. SAMET.
 The Quadtree and Related Hierarchical Data Structures.
ACM Computing Surveys, 16(2), June 1984.

[Sar84]

S.K. SARIN.
Interactive On-Line Conferences.
 PhD Thesis, M.I.T., December, 1984.

[SaG85]

S. SARIN and I. GREIF.
 Computer-Based Real-time Conferencing Systems.
Computer, October 1985.

[SpT74]

R. F. SPROULL and E. L. THOMAS.
 A Network Graphics Protocol.
Computer Graphics, 8(3), ACM, Fall 1974.

[SBF86]

M. STEFIK, D.G. BOBROW, G. FOSTER, S. LANNING and D. TATAR.
WYSIWIS Revised: Early Experiences with Multi-User Interfaces.
 Intelligent Systems Laboratory, Xerox PARC, Palo Alto, February 1986.

[SFB87]

M. STEFIK, G. FOSTER, D. BOBROW, K. KAHN, S. LANNING and L. SUCHMAN.
 Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings.

Communications of the ACM, January 1987.

[Sun86]

SUN MICROSYSTEMS.

NeWS Preliminary Technical Report.

Sun Microsystems, Inc., Mountain View, California, 2 October 1986.

[Tho84]

G. B. THOMPSON.

Information Technology: A question of perception.

Telesis, BNR, 1984.