Spatial Contexts:
An Interactive Environment for Personal Design

by
Jonathan Scott Linowes

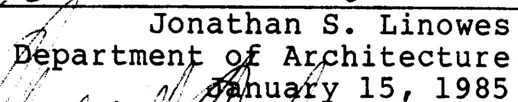
Bachelor of Fine Arts
Syracuse University
Syracuse, New York
1979

SUBMITTED TO THE DEPARTMENT OF ARCHITECTURE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE
MASTER OF SCIENCE IN VISUAL STUDIES AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

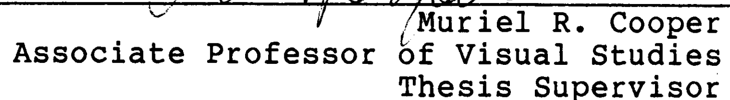
February, 1986

© Jonathan Scott Linowes, 1986

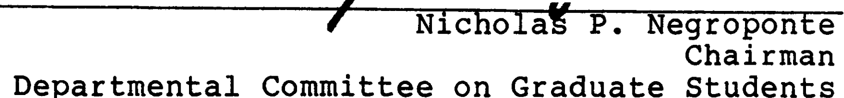
The Author hereby grants to M.I.T. permission
to reproduce and distribute publicly copies
of this thesis document in whole or in part.



Signature of the author ______________________
Jonathan S. Linowes
Department of Architecture
January 15, 1985



Certified by ______________________
Muriel R. Cooper
Associate Professor of Visual Studies
Thesis Supervisor



Accepted by ______________________
Nicholas P. Negroponte
Chairman
Departmental Committee on Graduate Students

Archives
MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

FEB 0 3 1986

LIBRARIES

Room 14-0551
77 Massachusetts Avenue
Cambridge, MA 02139
Ph: 617.253.2800
Email: docs@mit.edu
http://libraries.mit.edu/docs

DISCLAIMER OF QUALITY

Due to the condition of the original material, there are unavoidable flaws in this reproduction. We have made every effort possible to provide you with the best copy available. If you are dissatisfied with this product and find it unusable, please contact Document Services as soon as possible.

Thank you.

The images contained in this document are of the best quality available.

Spatial Contexts:
An Interactive Environment for Personal Design

by
Jonathan Scott Linowes

Submitted to the Department of Architecture, MIT
on January 17, 1986
in partial fulfillment of the requirements for the degree
Master of Science in Visual Studies

ABSTRACT

Personal workspaces are places where one comfortably gets work done organizing information and exploring ideas. Information is organized by moving things around, drawing associations and making piles. Ideas are explored by creating sketches and plans and then editing and refining them, moving between levels of detail and abstraction. For personal computers to support individual styles of problem solving and design they must exhibit the qualities that make ordinary workspaces feel personal.

The system design for an interactive environment, Spatial Contexts, is presented that provides a framework for building computer-based personal workspaces, drawing upon a number of interaction metaphors. A prototype system has been developed and demonstrated on a color graphics computer workstation with an object-oriented software architecture.

In Spatial Contexts, three principal classes of objects are defined: Elements- singular objects referencing specific information; Containers- objects that contain other objects for compositions, groups and hierarchies; and Tools- functional objects that are applied to other objects. Complex workspaces can be built from these, for applications that include page layout, animation storyboarding, spatial data management and semantic knowledge representation.

Thesis Supervisor: Muriel Cooper
Associate Professor of Visual Studies
Director, Visible Language Workshop, MIT

ACKNOWLEDGMENTS

It is a pleasure to thank the people who have assisted me in producing this thesis.

My thesis supervisors, Muriel Cooper and Ron MacNeil, provided continued support and encouragement as I struggled to focus my ideas, develop software, and help build the VLW laboratory research environment.

The thesis committee showed much patience while offering a critical eye, included Steven Benton, Lois Craig, Henry Leiberman, and Patrick Purcell. Don Hatfield and Alan Kay offered stimulating discussions early in the development.

Research assistantship support were granted principly by Dr-Ing. Rudolf Hell, GmbH of Kiel, Germany, with additional support from Toshiba Corp.

Special thanks to the graduate and undergraduate students who played a part in the discussion and implementation of these ideas. In particular, Dimitry Rtischev's and Robert Mollitar's programming wizardry were invaluable.

Finally, my family was an unlimited source of support, especially Lisa who married me during these maddening times, and our cats: Winchester, Fractal, Ubu, and Sid who put up with my obsessive persistence (and I with theirs).

TABLE OF CONTENTS

INTRODUCTION 6

Section 1. CONCEPT

1.1 PERSONAL DESIGN WORKSPACES

1.1.1 Personal workspaces are dynamic inhabited spaces where one organizes things and develops ideas 10

1.1.2 Work environments are composed of many integrated workspaces that evolve as one works 13

1.1.3 Objects in the environment are organized spatially and symbolically 16

1.1.4 Structured design methods provide guidance by constraining the process and limiting choices 19

1.1.5 Informal design methods encourage experimentation and innovation....................................25

1.1.6 Various spatial, temporal and detail representations are used during design 27

1.2 COMPUTER METAPHORS

1.2.1 Computer graphic workspaces rely on the sensation of direct manipulation of simulated objects 33

1.2.2 Electronic workspaces are metaphors for real world workspaces 35

1.2.3 Workspaces can be explored and edited by perusing and arranging data resources 37

1.2.4 Workspaces are programmed by assembling objects into functional models 40

1.2.5 New and refined objects are defined relative to existing ones 43

1.2.6 Objects in the workspace are associated by property values and arbitrary semantic connections 45

1.2.7 Electronic workspaces need to integrate with our everyday workspaces 47

Section 2. SCENARIO

2.1 USER MODEL

2.1.1 Users sitting at the Spatial Contexts system are encouraged to explore, synthesize and massage their work 50

2.1.2 A new environment contains the Main Cabinet and a Main Toolbox for the system's elements, containers and tools 52

2.1.3 High level application environments can be built from these primitives as the user works 55

2.2 TASKS

2.2.1 Setup a new job by collecting objects into a Job container 57

2.2.2 Setup workspaces by arranging the job components and tools 59

2.2.3 Begin sketching a few ideas for the layout 61

2.2.4 Refine a sketch into a layout 63

2.2.5 Edit an image on a page by cropping and color retouching 65

2.3 OBJECTS

2.3.1 Elements are singularly defined units of information with properties and form 67

2.3.2 Containers are objects that contain and constrain other objects 73

2.3.3 Tools are functional objects one uses to edit other objects and peruse the environment 78

Section 3. ARCHITECTURE

3.1 SYSTEM ORGANIZATION

3.1.1 Spatial Contexts software is organized in a layered architecture with objects 90

3.1.2 The system is demonstrated on a color graphic personal computer workstation 95

3.1.3 Application level objects are built from graphic elements, containers and tools 99

3.2 OBJECT ARCHITECTURE

3.2.1 All objects have the same basic structure, and are classified by their Class and Type 103

3.2.2 An object's Profile holds maintenance properties ..106

3.2.3 An object's Form defines device level properties for high speed display and interaction 107

3.2.4 Links define qualified associations between objects, include "isa", "home" and "contained by"..108

3.2.5 Methods are operations an object performs in response to messages 109

3.3 SUB-SYSTEM FUNCTIONS

3.3.1 Main System Driver 114

3.3.2 Object Management Subsystem 116

3.3.3 Display Subsystem 119

3.3.4 Input Subsystem 127

3.3.5 Memory and Database Subsystems 130

CONCLUSION 131

BIBLIOGRAPHY 133

INTRODUCTION

Design is both a cognitive and physical process of discovery and creation. It is not limited to the jobs of professional designers, such as graphic artists, architects and engineers, but is integral to everyday human activities of planning, problem solving and organization. Where there is decision making, there is design. Design involves planning what to do, deciding how to do it, and working to do it right.

Computers allow us to look at situations in ways never before possible, for modeling, simulation and information organization. They are changing the way we work. If tasks are to be moved to the computer workstation, so must the qualities of design and creativity that make the work challenging, rewarding and personal. Workstations must become personal workspaces.

Page layout design has a history in conventional media, yet it is an application whose tasks are increasingly being supported in personal workstations. Graphic designers are typically not computer users, preferring hands-on work. They like to play with paper as they play with ideas, arranging visual elements to explore new compositions.

Such interactivity with one's work is also true for many other tasks, including those based in today's electronic information media. The role of designer (author, producer) and user (reader, consumer) are converging as each of us becomes more active in the selection, production and presentation- the personal design- of our own information.

This thesis is an exploration of design as an interactive personal process and the implications for electronic computer-based workspaces.

Section One, CONCEPT, asserts that spatial organization and spatial thinking are integral to problem solving. The notion of "personal design workspaces" is defined as both the physical and mental organization of objects during the design process. Then, a number of key characteristics of computer-based workspaces are identified that can support personal design, drawing upon recent metaphors of human-computer interaction.

Section Two, SCENARIO, presents an experimental user interface system, Spatial Contexts, as an example of an interactive environment that becomes an evolutionary personal workspace. A sequence of interaction scenarios

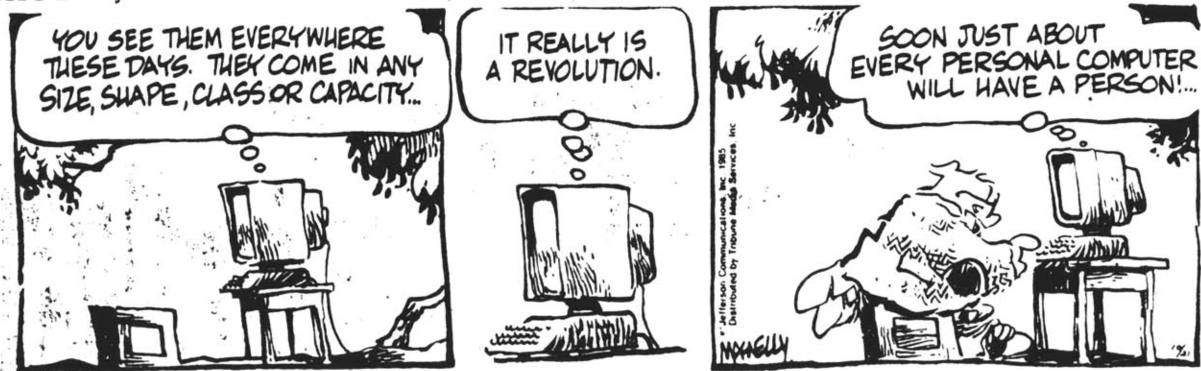
demonstrate how workspaces evolve as one's work develops. The properties and behaviors of particular objects are defined.

Section Three, ARCHITECTURE, specifies the implementation of the system using a color graphic computer workstation and an object-oriented software architecture. The user works with top level application-specific objects built from the principal object classes: graphic elements, containers, and tools. The application is supported by the sub-system libraries for object, display, input, memory and database management.

The software program that accompanies this document is a prototype for a Spatial Contexts system, demonstrating many of the key system concepts defined here.

"All who use computers in complex ways are using computers to design or to participate in the process of design. Consequently we as designers, or as designers of design processes, have had to be explicit as never before about what is involved in creating a design and what takes place while creation is going on." Herbert A Simon [SIMON'81].

SHOE by Jeff MacNelly



Section 1. CONCEPT

1.1 PERSONAL DESIGN WORKSPACES

1.1.1 Personal workspaces are dynamic inhabited spaces where one organizes things and develops ideas.

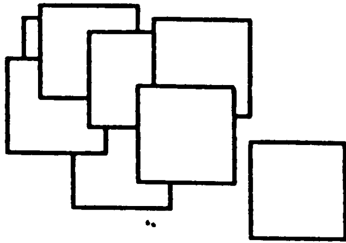
When you move into an empty new office, it may feel uncomfortable at first, equipped with a strange desk and bare shelves. But soon the office becomes your own as you move in, get to work and begin to inhabit the space. It becomes your personal workspace, where you organize papers and notes, make piles, file things away for later retrieval.

Thomas Malone has examined ways people organize their offices, and the implications for the design of office systems [MALONE'83]. He asserts that spatial organization of work is a function of both the working style of a person and his particular task.

The organization of material on one's desk is not just to facilitate information retrieval, but to be reminded of tasks to do. Certain piles represent work in similar states of completion. Materials on a desk are edited, shuffled around and accumulated dynamically as work progresses.

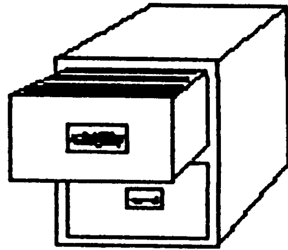
Malone notes that "the cognitive difficulty of categorizing information" is an important factor in explaining how people organize their desks. Piles are loosely defined groups whereas files contain named and ordered elements. Piles are less tedious to use than files, but are inexact and informal (Figure 1-1).

Electronic information workspaces must allow the user to build piles, and should assist the user with information categorization and filing.



Piles

organic
unnamed
associated
unordered
spatial



Files

organized
named
grouped
sorted
symbolic

Figure 1-1. Piles vs Files

1.1.2 Work environments are composed of many integrated workspaces that evolve as one works.

At the studio office, the graphic designer has a personal work environment that has evolved through the day to day habitation of the space. It is made of work areas for editing ideas, reviewing work in progress, arranging information, and collecting tools (Figure 1-2).

The drawing board is the space for generating and editing ideas. We all have had to "go back to the drawing board" now and then. Plans and prototypes are designed in this primary editing space. There may be several drawing boards simultaneously when (1) more than one problem is worked on at a time, (2) there are several approaches to a single problem, and (3) a single idea is represented in various levels of abstraction and refinement.

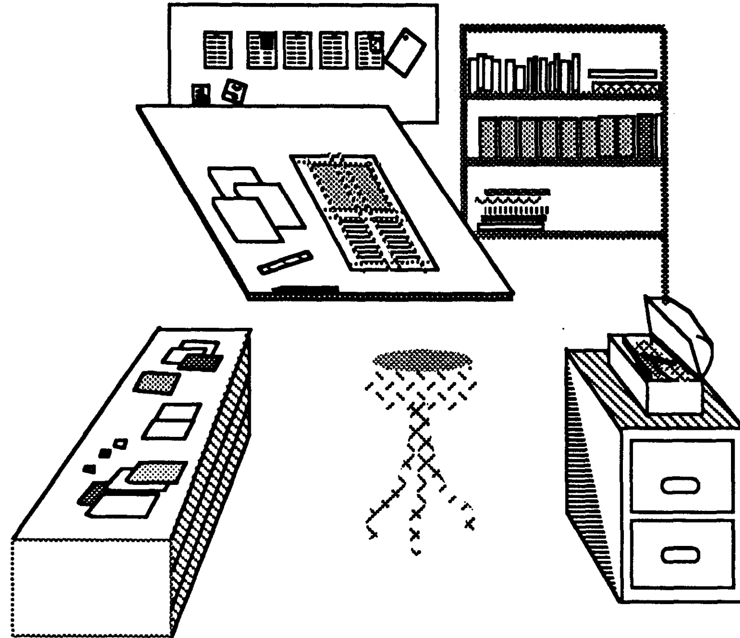
As work progresses, one often needs to step back and evaluate the overall flow and rhythm of a project or design. Sketches and miniatures are posted and reviewed on the tackboard. This is useful for sequencing, as well as tracking progress and communicating with others on the project.

Other spaces include cabinets, shelves and drawers. Design and problem solving more often requires the arrangement and combination of existing material than the generation of completely new work. Components are stored, retrieved, collected and arranged in these reference spaces as they are brought into play.

Toolboxes contain various tools and other small items. Designers have many tools, though only a few are used most frequently. These tend to be collected near the work areas, in the context of the work. The usage of a tool depends on its context: as a function of the tool, the object it is applied to, and the purpose of the action. For instance, a pencil is ordinarily used to make marks on paper, however it can also punch holes in the paper. Tools are used to transform components of the design, edit things, and even modify other tools, as a pencil sharpener sharpens pencils.

Personal workspaces are made of task components and tools arranged into work areas. One should be able to similarly configure their own electronic workspaces.

Figure 1-2. Sketch of a Graphic Design Studio



Personal Design Workspaces

- DRAWING BOARD: edit, compose, experiment
- TACKBOARD: evaluate, sequence, review
- REFERENCE SPACE: store, retreive, collect, arrange
- TOOLBOX: tools, utensils, gadgets

1.1.3 Objects in the environment are organized spatially and symbolically.

Chunking and layering are primary principles of organizing information in the human mind [MILLER'76]. A chunk is a symbolic object of arbitrary complexity, built on hierarchical layers, and handled as a single entity. Chunking allows one to synthesize mental models that are manipulated as unique entities, and perceive trends and patterns in information. Personal workspaces reflect the mind's chunking and layering activity, as items are spatially arranged and hierachically grouped.

Mental organization can be highly symbolic, but is greatly aided by spatial references and visualization. Inherently non-spatial concepts can be graphically represented, providing visible maps of the mind [FILLENBAUM'71, HAMPDEN'81, SCHUBERT'83]. Figure 1-3 shows a formal psychological study where emotional names were spatially arranged and clustered, and, more personally, a map of activities in the author's life. Expressing concepts geometrically helps make them more concrete and tractable, as the spatial-visual 'right brain' supports the linguistic-symbolic 'left brain'.

The inverse is also true- physical objects can be related to one another in many ways other than simply their physical position. Tools may be scattered about an office, but each is still a kind of tool. A photograph may lie in a pile of photos for a particular document, but still is associated with others by the same photographer, of the same subject, etc.

One draws both spatial and symbolic associations between objects. Electronic workspaces should allow users to freely express and edit association maps between objects.

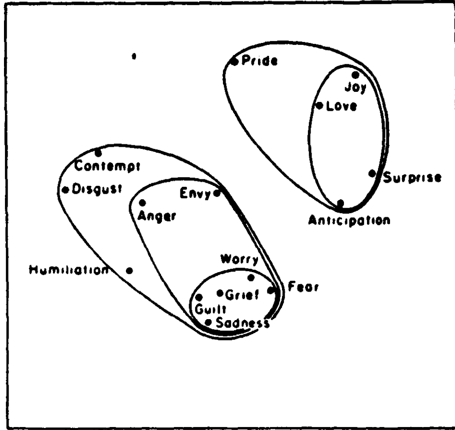


Fig. 6-9. Two-dimensional Euclidean representation for Group ECS.

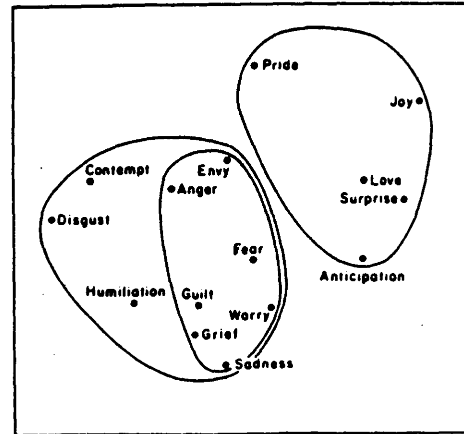
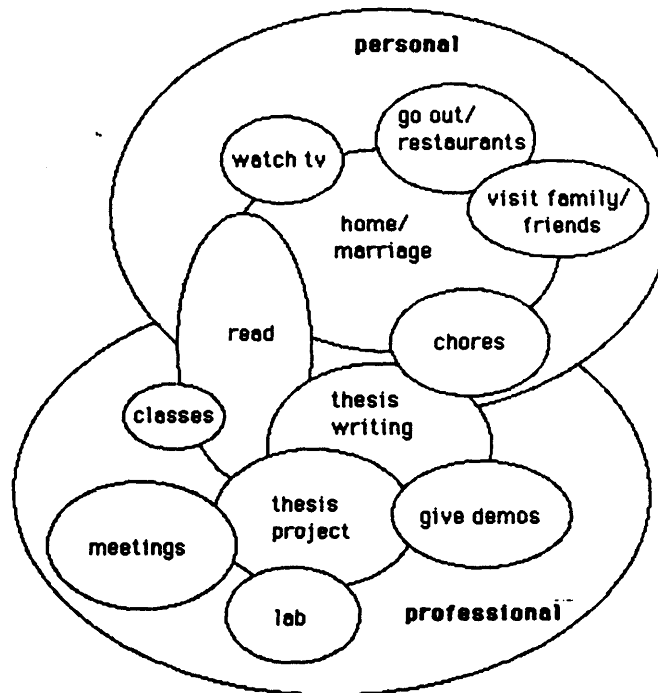


Fig. 6-11. Two-dimensional Euclidean representation for Group FR

[from Fillenbaum, Structures in the Subjective Lexicon]

Spatial arrangement and statistical clustering of emotional names from four groups of human subjects.



A Map of Activities in the Author's Life

Figure 1-3

1.1.4 Structured design methods provide guidance by constraining the process and limiting choices.

Structured design methods help us understand the design process and offer effective techniques. Design activities are organized into components and stages of completion. The problem domain is structured as though it is inherently organized and defined, just waiting to be discovered. To find a solution means to map a path through the objects, operations, subgoals and constraints that make up the problem domain. Figures 1-4 and 1-5 show models for graphic design and software design processes, respectively.

One method of structured design is the top-down approach with incremental refinement. One starts out defining the task in general terms, identifying the overall goals, inherent constraints and prescribed specifications. The problem is decomposed into a set of subtasks, and the process of refinement is continued as each level resolves the goals of the one above until the solution is satisfactorily mapped out. Nicholas Wirth explains this in terms of software design: "The programmer sets up a hierarchy of abstractions, viewing the program first in broad outline and then attending to one part at a time while ignoring the internal details of other parts." [WIRTH'84].

Graphic artists have many structured systems to work through design decisions of varying complexity. Karl Gerstner, for example, considers the grid to be an excellent example of a systematic design tool [GERSTNER'64]. Offering a structured approach to spatial layout, the grid is a "proportional regulator" that offers many intelligent choices through tight constraints, as in Figure 1-6. Other structured systems take the form of symmetrical geometry, ordered tables and matrices, notation systems and three dimensional models. These are manifestations of the problem space in which solutions can be mapped.

Graphic designer Allen Hurlburt comments that solving a design problem is much like running a maze. "The designer selects a line to follow only to learn that the constraints he encounters send him back to probe another direction until he finds a clear path to the solution." [HURLBURT'78]. Finding a solution is then reduced to a search problem [SIMON'81]. Gerstner reduces it even further, to simply a matter of making the right choices.

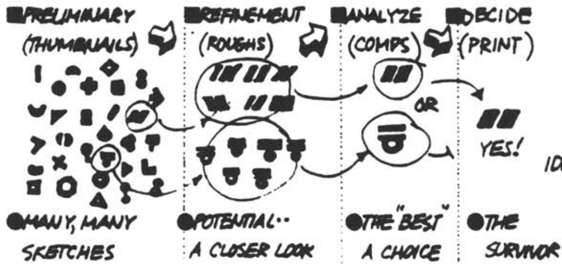
> "To describe the problem is part of the solution. This implies: not to make creative decisions as prompted by feeling but by intellectual criteria. The more exact and complete these criteria are, the more creative the work becomes. The creative process becomes reduced to an act of selection. Designing means: to pick out determining elements and combine them." [GERSTNER,p8.10]

Enforcement of structured design systems should be available in electronic workspaces to help constrain the user to predefined sequences of operations, limit choices at specific decision nodes, and maintain components of the work as they are developed.

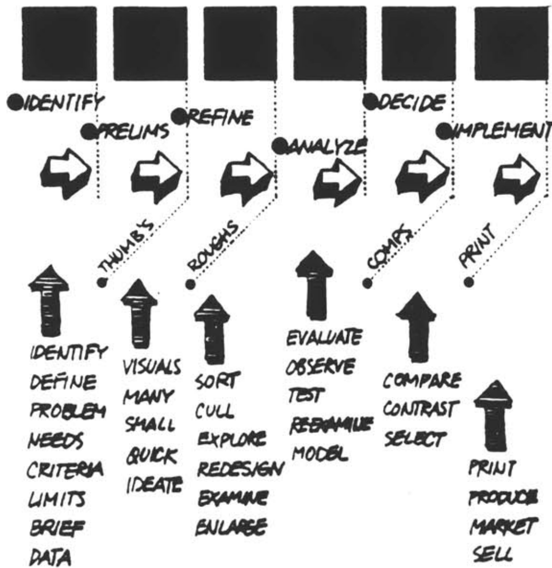
PROCESS

THE DESIGN PROCESS CAN BE AS SIMPLE AS MAKING A COLOR CHOICE OR AS COMPLEX AS FORMATTING A SERIES OF SCIENTIFIC TEXTBOOKS. IT CAN RANGE FROM SELECTING A TYPEFACE TO DESIGNING A GRAPHIC CONTROL ENVIRONMENT FOR A MASSIVE WATER CONTROL PROJECT. ABOUT THE ONLY THING CONSTANT IN GRAPHIC PROBLEMS IS THE FACT THAT EACH PROBLEM HAS UNIQUE DIFFERENCES. YET CERTAIN COMMONALITIES DO HELP DESIGNERS TO STRUCTURE THEIR ATTACK ON A PROBLEM.

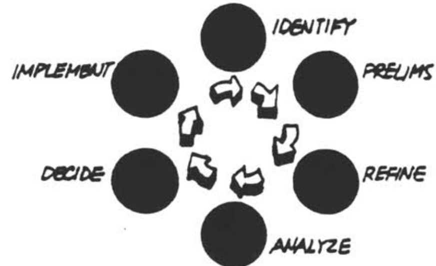
ALTERNATE SOLUTIONS · ANY PROBLEM HAS AN INFINITE NUMBER OF POSSIBLE VISUAL SOLUTIONS, IF WE CAN ACCEPT THIS FACT, AND CAN GENERATE VISUAL ALTERNATIVES, A GOOD DEAL OF OUR DESIGN ACTIVITY CAN INVOLVE MAKING VISUAL CHOICES.



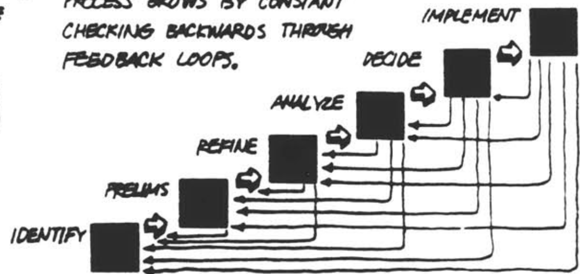
LINEAR PROCESS · ONE STAGE FOLLOWS ANOTHER IN A STRAIGHT LINE.



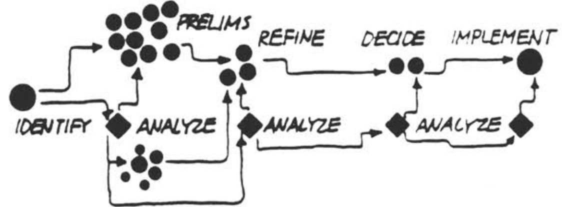
CYCLIC PROCESS · PROCESS MOVES IN A CYCLE OR CIRCLE WITH NO CLEAR START OR FINISH.



FEEDBACK PROCESS · LOOKING BACKWARD HELPS THE PROCESS ALONG. CONCURRENT PROGRESS. PROCESS GROWS BY CONSTANT CHECKING BACKWARDS THROUGH FEEDBACK LOOPS.



BRANCHING PROCESS · CERTAIN STAGES TRIGGER PROCESS GROWTH IN MORE THAN ONE DIRECTION LIKE, A TREE.



PRIORITY PROCESS · IN THE DESIGN PROCESS, THE ESTABLISHMENT OF PRIORITIES IS ESSENTIAL. DESIGNERS MUST BE ABLE TO JUDGE AND GAUGE THE RELATIVE IMPORTANCE OF FACTORS AS THEY RELATE TO ONE ANOTHER. PRIORITIES SET THE FUNCTIONAL AND VISUAL CRITERIA IN COMMUNICATIONS.

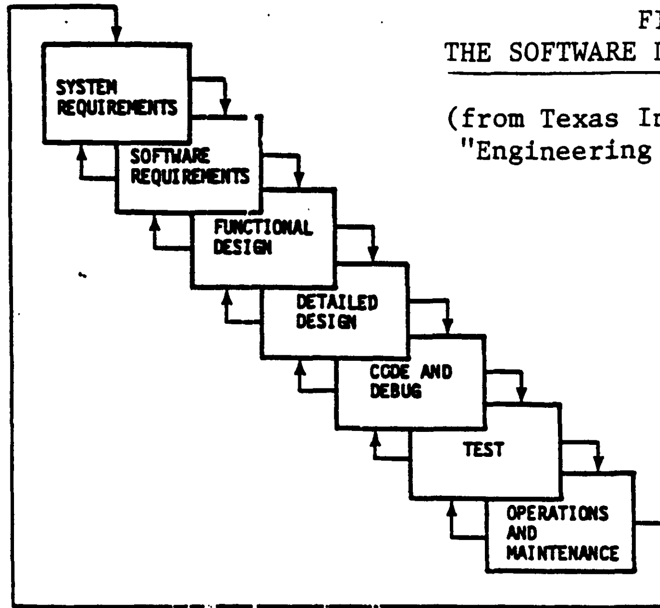
7

(from Barryman, "Notes on Graphic Design and Visual Communication")

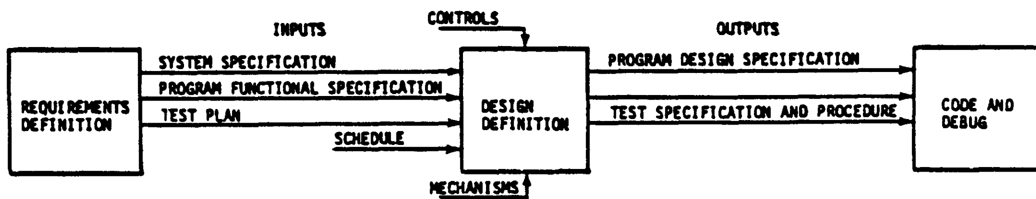
FIGURE 1-4 GRAPHIC DESIGN PROCESS

FIGURE 1-5
THE SOFTWARE DESIGN PROCESS

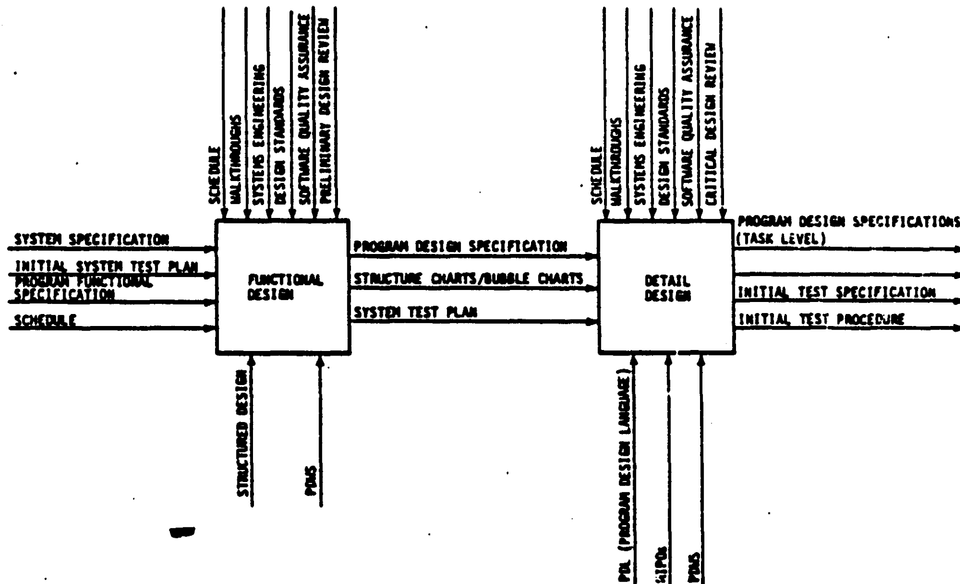
(from Texas Instruments, Inc.
"Engineering TI Style" manual)



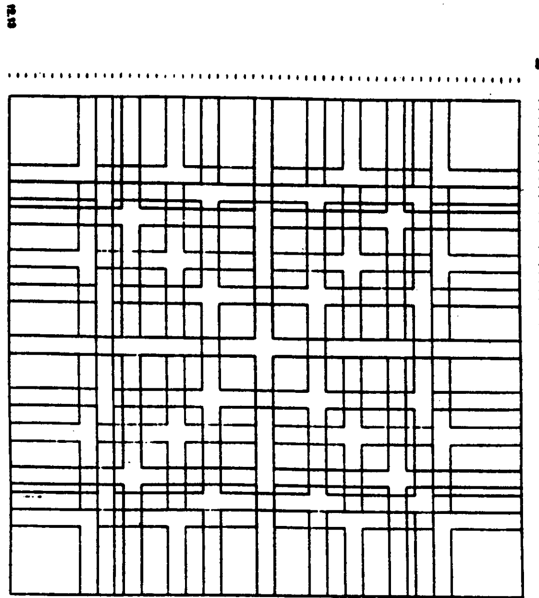
Software Development Process

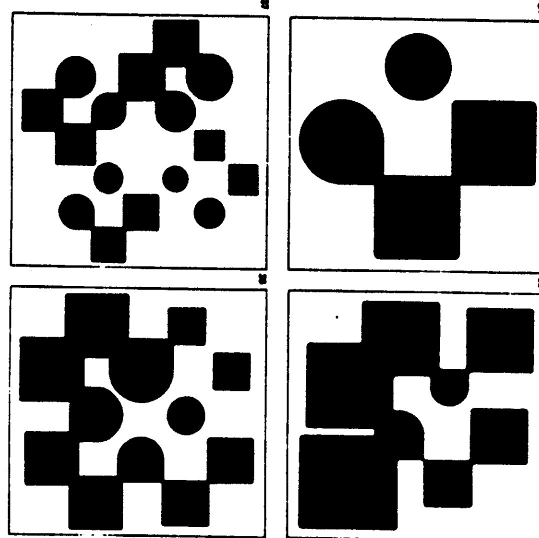


Design Phase Inputs and Outputs



Design Phase Parameters





(from Gerstner, "Designing Programmes")

FIGURE 1-6 GRIDS AS A GRAPHIC DESIGN SYSTEM

1.1.5 Informal design methods encourage experimentation and innovation.

Structured design methodologies try to engineer the design process, providing discipline and rules. They can be quite useful and effective, but if taken too seriously in practice they can become stiff and confining. When a particular design strategy is rigidly imposed, it turns from a tool of great potential into a straitjacket.

Informal design allows for intuitive discovery and evolution of technique. Informal design methods encourage experimentation and innovation. Accidents are allowed to happen.

There is no single correct solution to any design problem. Individuals make for individual differences, though a good solution will be widely recognized as such. One relies on techniques and rules learned and shared in the cultural environment of peers and experts. These statements of conventional wisdom are respected because they work, most of the time. Whereas structured design rules are meant to be enforced, informal rules are often stretched to their limits to control contrast and harmony dynamics.

Design problems are evolutionary, not static. The problem domain changes as the solution emerges, not only in the sense that the next step is different from the previous, but the objectives of the design may evolve as well. Flexibility with old ideas and reception to new ones is critical.

The user interface to electronic workspaces should permit the user to relax structured constraints and work loosely through a problem. Informality is essential for one to work comfortably in his or her personal workspaces.

1.1.6 Various spatial, temporal, and detail representations are used during the design process.

Consider the process of designing the layout of a multi-page pamphlet. A 'page' contains information in different levels of material, style and content components, as in Figure 1-7. Material components include the background or paper characteristics. The style sheet specifies constraints on the format of the content, including typographic, graphic and grid specifications. The contents of a page include graphics, images and text data. The page is part of a larger whole, sequentially linked to other pages in the document.

Designers are concerned not only with the spatial layout of individual pages, but the continuity, flow and rhythm of the document as a whole, as graphic designer White explains in Figure 1-8 [WHITE'82].

In this sense, page layout shares characteristics of temporal layout of animated sequences. Figure 1-9 shows two graphic layouts of animated films. The top one shows the components of a single shot, including separate photographic images, drawings and masks, and how they are combined within

a frame and within the sequence. The lower illustration is an excerpt from a film showing key frames and verbal descriptions of the actions and sound in each shot.

The structure of a page layout can be represented at various levels of refinement and detail. Figure 1-10 shows the four basic levels of a page layout. Initial ideas of the design are explored with thumbnail sketches, quickly drawn and easily arranged for comparison and sequencing. As the number of ideas are narrowed, more detailed rough sketches are developed that help the designer determine balance, scale and format relationships in context of the text and

At an early stage, a basic grid is defined. The grid provides a framework for the layout. Grid lines offer guidance for the alignment of components on the page and consistency between pages. The strict use of grids is optional and unique grids often must be developed for different layout strategies.

Electronic workspaces intended to support interactive design tasks should allow for varieties of spatial, temporal, and detail representations of the work.

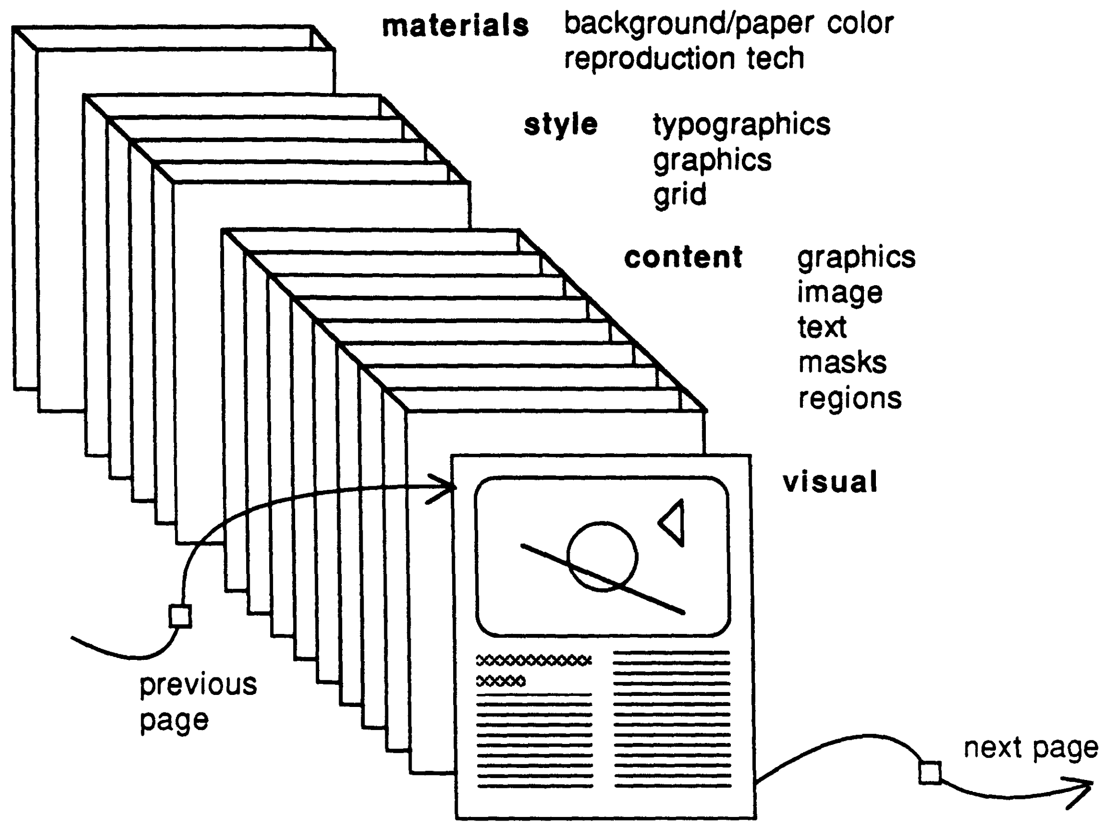
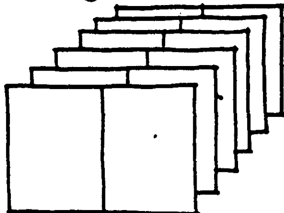
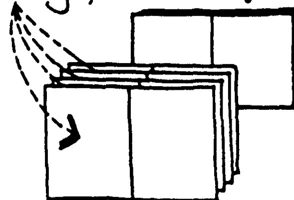
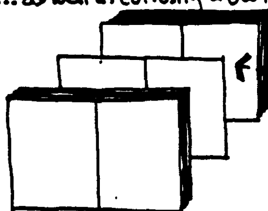
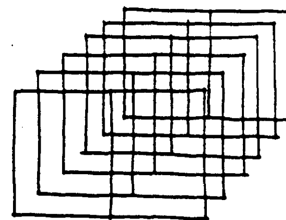


Figure 1-7. Layered Components of a Page





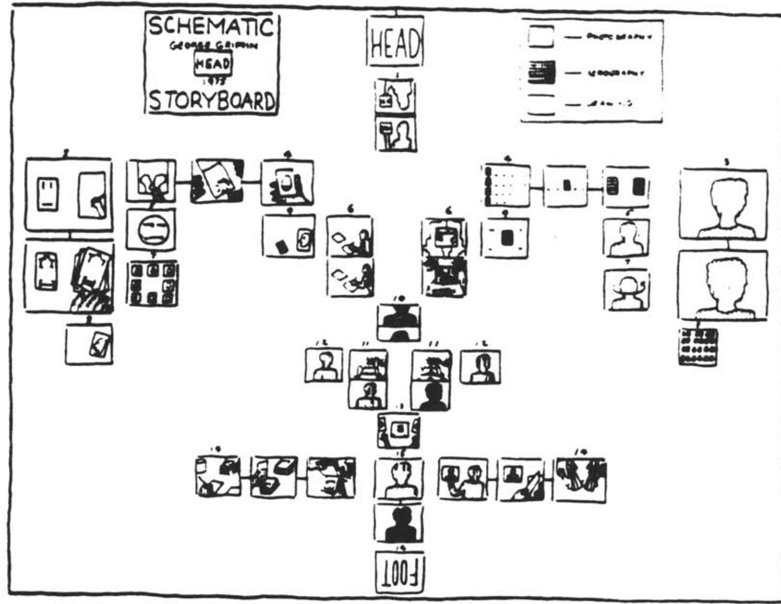




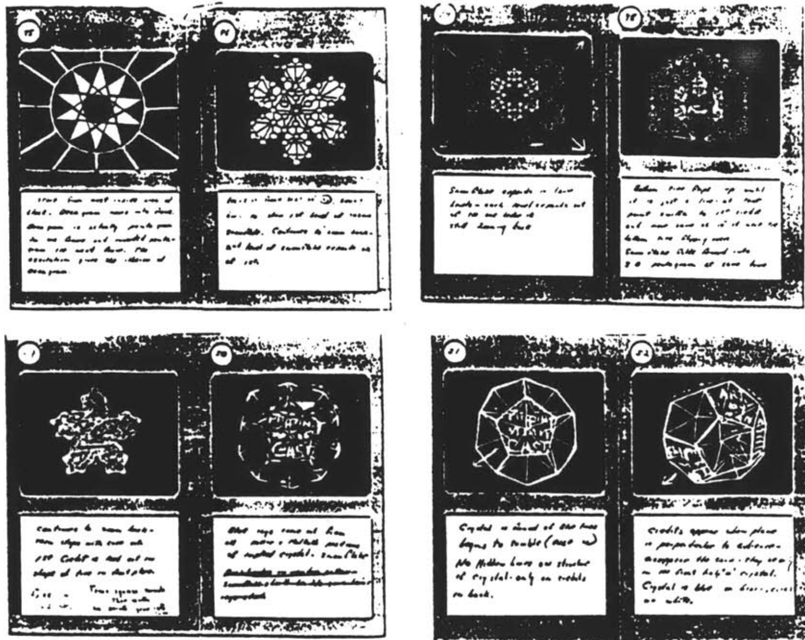
(from White, "Editing by Design")

"Design each story, no matter how long or how short, as a unit. Do not fall into the trap of working on one page of a multipage story until you have solved the basic pattern for the entire story, start to finish. This is where the value of that kitchen-counter worktable is most evident: use it to organize the raw material into piles coordinated to the pages on which the material is planned to fall; then work out an overall design that will accommodate the material within the story's matrix". [WHITE,p25].

FIGURE 1-8 THE TEMPORAL DIMENSION OF PAGE LAYOUT



(from Laybourne, "The Animation Book")

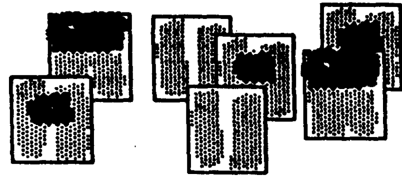


(excerpt from Bruce MacCurdy's film "Flipping")

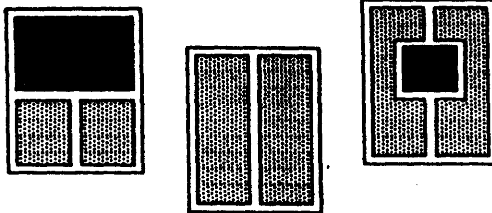
FIGURE 1-9 ANIMATION STORYBOARD STYLES

Figure 1-10

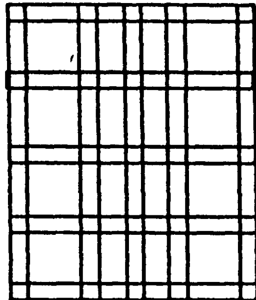
VARIOUS LEVELS OF REFINEMENT OF A PAGE LAYOUT



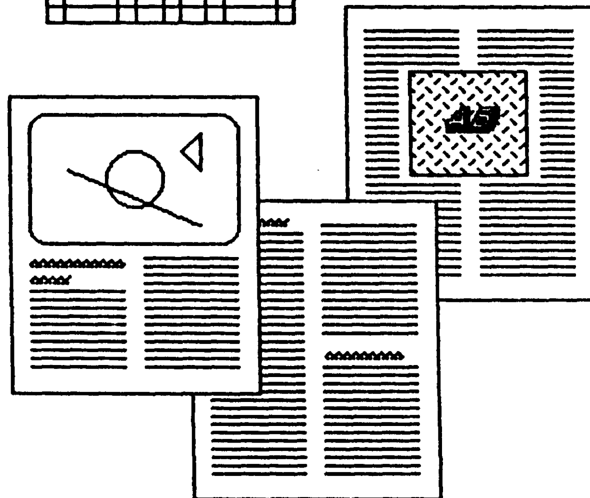
Thumbnail Sketches
small
quick
preliminary
exploratory



Rough Sketches
larger
more refined
more detail
balance
scale
patterns



Grid
spatial constraints
columns
horizons
margins
gutters
allignment



Comprehensive Sketches

mockup
approx. final product
final touchup
crop images
flow text

1.2 COMPUTER METAPHORS

1.2.1 Computer graphic workspaces rely on the sensation of direct manipulation of simulated objects.

Interactive computer graphics provides a medium for exploring object-oriented design, where a user manipulates the model of an object while its picture is continually updated on the screen. Ivan Sutherland's Sketchpad system was the first computer aided design system to allow the interactive creation, editing and constraint of graphic layouts [SUTHERLAND'63].

The MacIntosh computer brings the computer graphics paradigm into the hands of consumers. The MacDraw program is a typical graphics editor where graphic objects, such as lines, curves and text, are selected from a menu and added to a composition. The component is easily edited by performing an operation selected from a menu or by using a surrounding "handle".

Another MacIntosh program, FileVision, extends the concept so that objects are associated with fields in a database. No longer simply components of an illustration,

the graphic objects become symbolic references or representational images in the context of an information domain.

Traditional computer-aided design systems exploit the integration of computer graphics with domain-specific databases. Two and three dimensional designs can be modeled and incrementally transformed. These may be high quality simulations of real world objects, for what-you-see-is-what--you-get (wysiwyg) interaction.

Beyond wysiwyg, alternative views of the same data can be represented. In graphic design, for instance, page layouts need to be viewed at different levels of refinement (thumbnails, roughs, comps), as layers of components, and as temporal sequences, as discussed in the previous section.

The direct manipulation metaphor of interactive computer graphics offers a simulated physical world with tactile eye-hand feedback. Computer graphics allows design by construction and editing of objects.

1.2.2 Electronic workspaces are metaphors for real world workspaces.

The metaphor of using the computer screen as an "electronic desktop" was first introduced by the Learning Research Group at the Xerox Palo Alto Research Center (PARC) in the early 1970's. The graphics screen corresponds to a desk, with overlapping spatial regions, or windows analogous to pieces of paper. Each window is a separate work area with its own data and functions. Alan Kay explains the invention of these windows,

> "In many instances the display screen is too small to hold all the information a user may wish to consult at one time and so we have developed "windows" or simulated display frames within the larger physical display. Windows organize simulations for editing and display, allowing a document composed of text, pictures, musical notation, dynamic animation and so on to be created and viewed at several levels of refinement." [KAY'77,p234].

Icons are small symbols and images that represent data and functions. For instance, to print a document, copy its icon onto the icon of the physical printer. To edit a file, open it's icon into a new window running its application program.

These concepts have been extended from business office environments to graphic design studios. The computer screen acts like an "electronic light table" [BLOMBERG'84]. The computer screen contains multiple viewports to layouts of objects on the tabletop. Two-dimensional graphic components, including slides, photographs, drawings, headlines and text can be overlapped, combined and masked. Like their real-world counterparts, each component has form, transparency and intensity attributes.

Workspace metaphors allows one to design directly in his problem domain. Non-technical users can easily approach the system, feel comfortable exploring its capabilities, and go about working in familiar ways. The ability to arrange and combine these components into meaningful piles and work areas facilitates building personal workspaces.

1.2.3 Workspaces can be explored and edited by perusing and arranging data resources.

Systems for data resource management have been developed which present the user with an interactive "dataland" he can travel through. Figure 1-11 shows a variety of dataland configurations. The ZOG system developed at Carnegie Mellon configures separate screenfuls of information, or frames, linked to other frames [ROBERTSON'79]. Frames were limited on text screens, although graphics and images should be included. From any one frame, the user can move through the network making choices between forward linked frames, or backing up to the previous frame.

Facilities for traveling through the ZOG frame-base and authoring new frames and links were available to anyone wanting to explore the datalands and develop their own frame-bases. A significant factor in its success was the speed of the ZOG system, since new frames could be displayed almost instantaneously allowing perusal of the database in interactive real time.

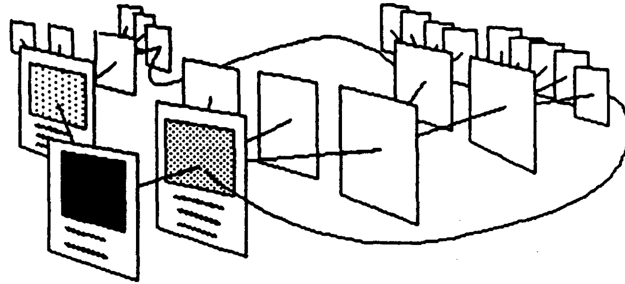
The Spatial Data Management system from MIT presents a dataland mapped onto the surface of a torus or doughnut [DONELSON'78]. One can pan up and down and around the space, looking through the "window" of the computer screen. If one zooms into an object, say a telephone, it becomes activated and the user can make a call. Other similar systems allow one to enter other datalands through portals, like trap doors in some exotic adventure game [HEROT'80].

Such systems allow one to build personal workspaces by organizing and customizing their dataland. Richard Bolt explains,

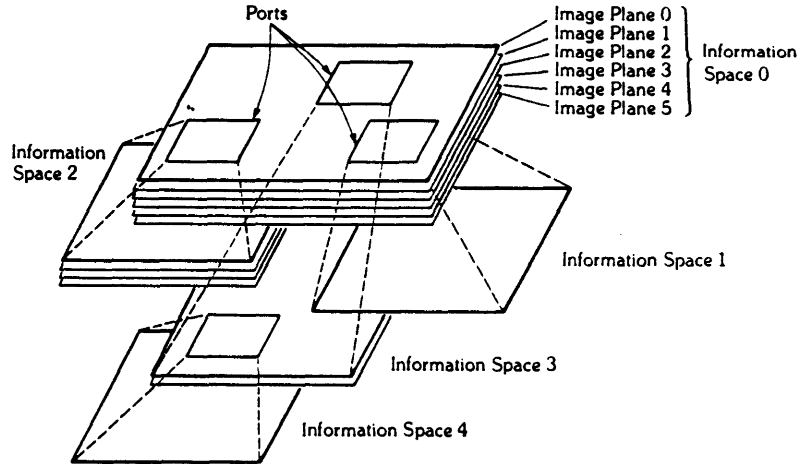
> "Your personal Dataland would look different from mine or anyone else's. There would be different items in different arrangements, just as the everyday desktops of people reflect their individuality. What would be common to all Datalands, however, is that the data types dwelling in them would be presented as images in specific locations." [BOLT'84,p11].

Datalands are an interactive world to explore, build, and design within. Structure is supplied by limiting the choices where one can go next, and by arranging and categorizing data into hierarchical files. Design becomes the organization and selection of objects in an electronic terrain.

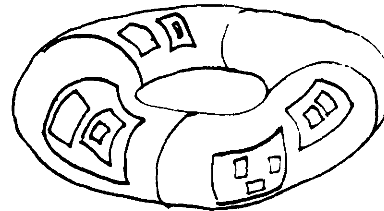
FIGURE 1-11 A VARIETY OF DATALANDS



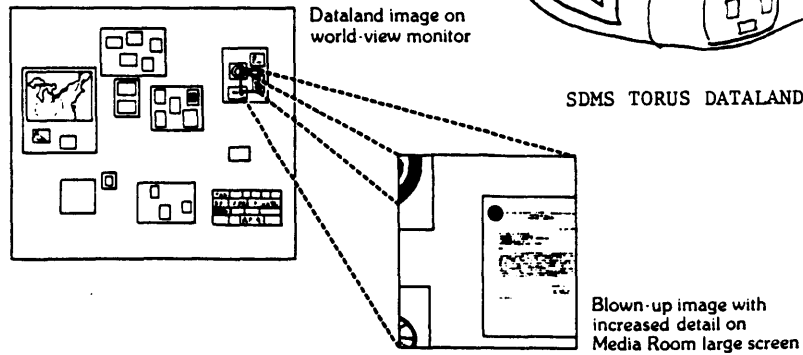
Frames in ZOG's Dataland



CCA SYSTEM WITH PORTS (from BOLT)



SDMS TORUS DATALAND



SPATIAL DATA MANAGEMENT SYSTEM, MIT (from BOLT)

1.2.4 Workspaces are programmed by assembling objects into functional models.

When behavior and responsiveness can be programmed into objects, the environment becomes a microworld with a finite number of objects and the ability to build new ones. Microworlds offer a toolkit for assembling objects into programs.

ThingLab is a microworld simulation laboratory for dynamic experiments in geometry and physics [BORNING'71]. The top of Figure 1-12 shows a Centigrade-Farenheit temperature conversion program built from basic arithmetic objects and sliding "thermometers" for input and output. ThingLab allows interactive programming by defining constraints between graphical objects.

There are other examples of graphical object programming. With the LOGO language, children have an "object to think with" and draw pictures by instructing a 'turtle' with a pen how to move [PAPERT'80]. ChipWits, a game on the MacIntosh, has the player program the behavior of a robot by assembling functional "chip" icons into programs. Rockey's Boots, an Atari video game, lets the

player collect electronic components and plug them together to build simple simulated machines. Electronic financial spreadsheets, such as VisiCalc and Lotus 1-2-3, are toolkits for financial modeling and simulation.

In each of these examples, programming is integrated into the user interface environment, in context of the particular user task. One learns to program without realizing it, since the goal is simply to draw a picture, win a game, or calculate a financial model.

By assembling objects, constraint relationships are defined that direct the behavior of the objects. There may be models and constraints given to the user, plus the facility to develop them for him or herself. Design becomes programming of constraints, experimentation and assemblage.

FIGURE 1-12 MICROWORLD PROGRAMMING TOOLKITS

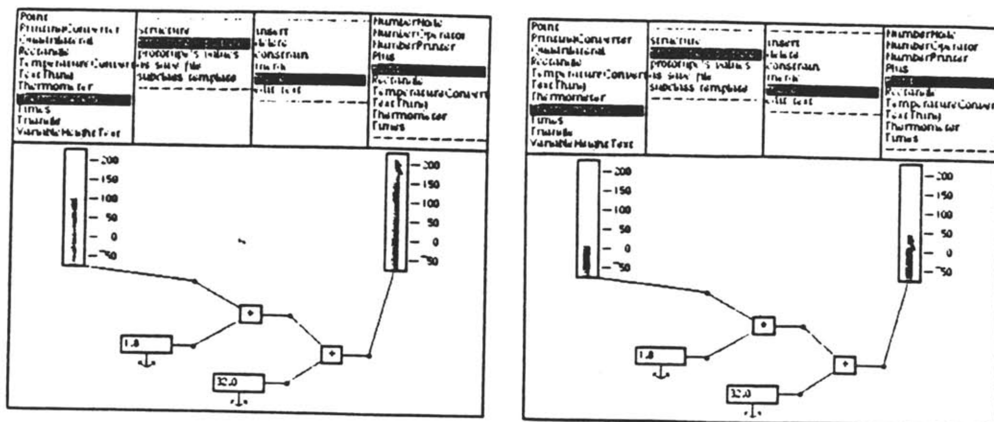
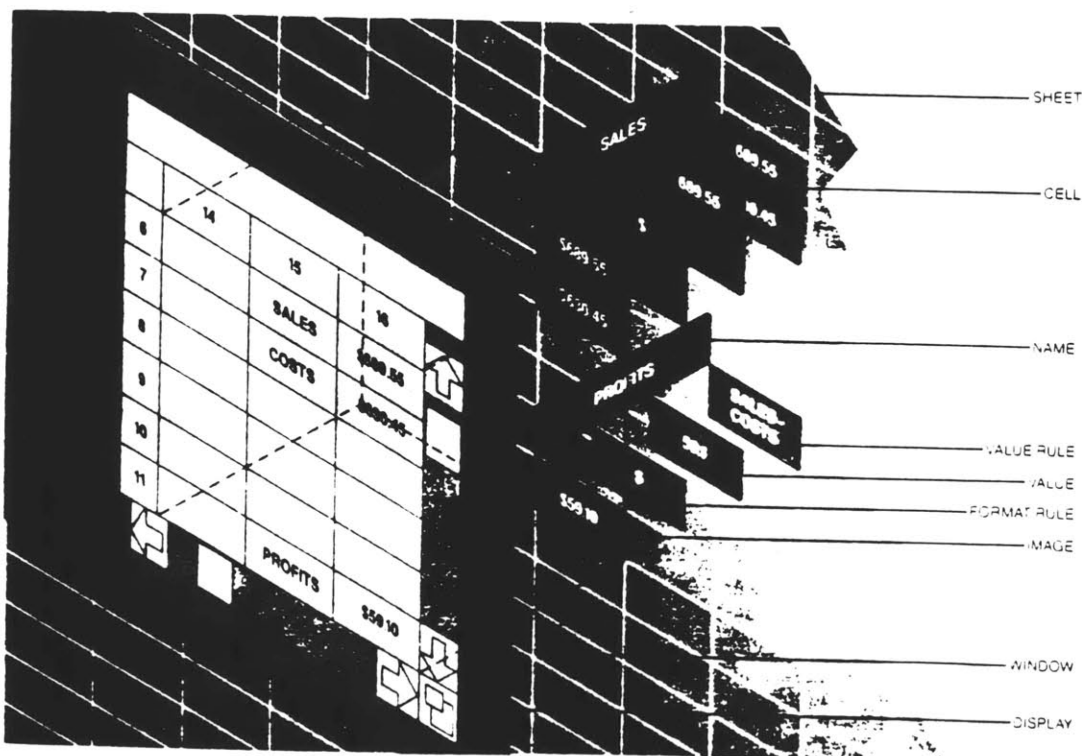


Fig. 13. The temperature converter with thermometers for input and output.

(from Borning, "The Programming Language Aspects of ThingLab")

A TEMPERATURE CONVERSION PROGRAM



DYNAMIC SPREADSHEET is a simulation kit: an aggregate of software objects called cells that can get values from one another. The window selects a rectangular part of the sheet for display. Each cell can be imagined as having several layers behind the sheet that compute the cell's value and determine the format of the presentation. The cell's name can be typed into an adjoining cell. Each cell has a value rule, which can be the value itself or a way to compute it; the value can also be conditional on the state of cells in other parts of the sheet. The format rule converts the value into a form suitable for display. The image is the formatted value as displayed in the sheet.

(from Kay, "Computer Software")

DYNAMIC SPREADSHEET AS PROGRAMMING TOOLKIT

1.2.5 New and refined objects are defined relative to existing ones.

Object-oriented programming languages provide a system for building taxonomic relationships between objects by defining objects as belonging to particular classes. New objects are created from existing ones by saying "this one is just like that one, except ...", and then enumerating the differences.

The Simula-67 language [DAHL'68] first introduced the 'class' construct, whereby sets of generic operations and properties are associated. New classes are be derived from existing ones, allowing inheritance to determine their operations and property sets. In Simula, these relationships are determined at compile time and are not dynamically alterable.

The Smalltalk language developed at Xerox PARC generalizes these ideas and supports a high degree of consistency, uniformity and integrity of object management [GOLDBERG'83, SHOCK'79]. Everything in the system is an object, including classes themselves. Objects can be altered dynamically, allowing the development of highly interactive

programming environments that encourage evolution and testing.

Multiple inheritance occurs when an object is defined from more than one parent. Conflicts must be resolved when different values are offered from different parents for the same property. It could be predetermined that one take precedence, or a method is defined to resolve the conflict at the time the property is referenced.

This ability to define new things relative to existing ones supports the evolutionary refinement of objects in the design environment. Ideas can be drawn from earlier work. Sketches can be refined. Work can progress in stages.

1.2.6 Objects in the workspace are associated by property values and arbitrary semantic connections.

Other relationships can be drawn between objects by matching property values, and by creating explicit semantic links.

Relational database systems provide a mechanism for collecting objects based on property values. Sets of objects of the same type can be selected based on a range of property values. Furthermore, sets of different types of objects can also be selected, when they share the specific properties matched against.

Rather than searching on property value ranges, explicit connections can be made between objects. Such links can be given a symbolic name, and be used by operations that recognize the link. Symbolic links drawn between discrete objects form semantic networks to represent relationships within specific knowledge domains [WOODS'83, BRACHMAN'83a, BRACHMAN'83b].

Relational and semantic connections are a means of defining symbolic associations between objects. In a

personal workspace, this permits loosely associating objects in a pile, creating containers of related files, searching for specific objects in a dataland, and support of design constraints requiring definition of specific links.

1.2.7 Electronic workspaces need to integrate with our everyday workspaces.

The physical spaces for these computer metaphors are as important as the conceptual models themselves. After all, if your chair is not comfortable, you will not get much work done regardless of how effectively you understand the task. Furthermore, a comfortable chair in an alien setting is inhibiting also.

For instance, a user sitting in the middle of a 'media room' has an electronic multi-sensory workspace that, in practice, would likely be too sterile and uncomfortable. More practical implementations of electronic workspaces need to integrate better with our everyday ones.

In the an office, an electronic workspace may look like a desk blotter, allowing one to move with ease between electronic workspaces and real pieces of paper, folders and telephones. In the home, consider a thin lightweight tablet about the size of a newspaper that you hold on your lap as you sit comfortably in your living room. Very portable workspaces should fit in your pocket or be worn like jewelry.

Computer-based workspaces must be useful, practical and personal. This discussion has identified a set of characteristics of personal design workspaces that all workspaces share. A set of computer metaphors have been identified that can support these requirements in electronic computer-based workspaces. The following section presents a user interface that begins to integrate these ideas.

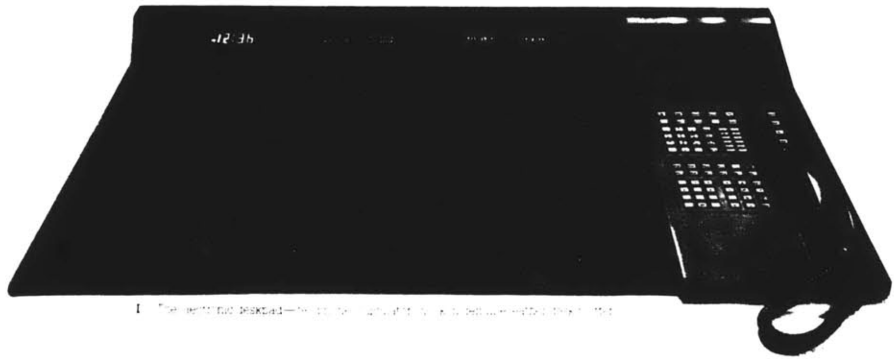


Figure 1-13. Electronic Workspaces Must Integrate with our Everyday Workspaces

Section 2. SCENARIO

2.1 USER MODEL

2.1.1 Users sitting at the Spatial Contexts system are encouraged to explore, synthesize and massage their work.

For interactive computer environments to be truly effective, the technology itself must become transparent allowing the user to work more directly in his problem domain. The user's primary concern is solving some design or management problem, not how to use the computer.

The objects in the Spatial Contexts system are familiar to the user and relevant to the task. They can be moved, modified, edited and used in different ways. Users are encouraged to organize their work spatially, creating different work areas and piles at will.

While composing a page of a document, a designer may need to choose one of several images for an illustration. The candidate images are collected in a pile and then inserted one at a time onto the page. If the image is sitting on the page or in a pile, it can be cropped and adjusted within that context. Contexts can constrain the

effect of tools on the object, and may even determine the visual representation of an the object. For instance, the image in the pile may be full color, while on the page it is black and white.

Users sitting at the system are encouraged to explore, synthesize and massage their work. One learns to travel through the datalands, examining objects, trying out tools and synthesizing his or her own models. The system is interactive and conversational as the user arranges the environment while being constrained by the workspaces. Easily moving between several workspaces at a time, one can organize things spatially, hierachically, and in other more symbolic configurations. It allows visualization of form with regard to spatial context.

2.1.2 A new environment contains the Main Cabinet and a Main Toolbox for the system's elements, containers and tools.

The Spatial Contexts simulated user environment, itself an object, contains all the objects the user is currently working with and access to all other objects in the system. The environment is responsible for intercepting user input and signaling appropriate objects that an event occurred.

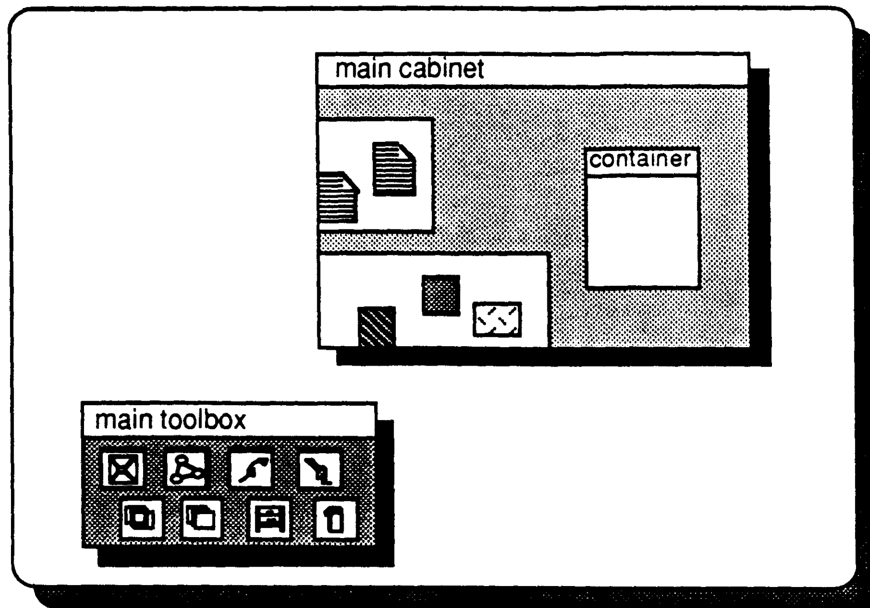
Figure 2-1 shows the default new environment with two open containers: a Main Cabinet portal to the system's dataland, and a Main Toolbox with a complete collection of tools on the system.

Data appears in the system as element objects which include graphics, images and text. Elements represent discrete units of information spatially arranged in a container. Their visual form and properties can be edited. Symbolic links can be drawn between them. Elements are discussed in Section 2.3.1.

Environments are a special type of Container object. Containers are objects that contain other objects, for making compositions, groups and hierarchies. Other types of

containers include Cabinets, Toolboxes and Pages. Containers are discussed in more detail in Section 2.3.2.

Tools are used to interact with objects, using a mouse input device. There are three ways to select tools: by picking one up, by moving the cursor over an attachment site, and by selecting it from a menu or button box. Tools are discussed in more detail in Section 2.3.3.



Default New Environment contains

Main Cabinet

principal port into the system's dataland
including any data files (pictures, text)
and an empty container

Main Toolbox

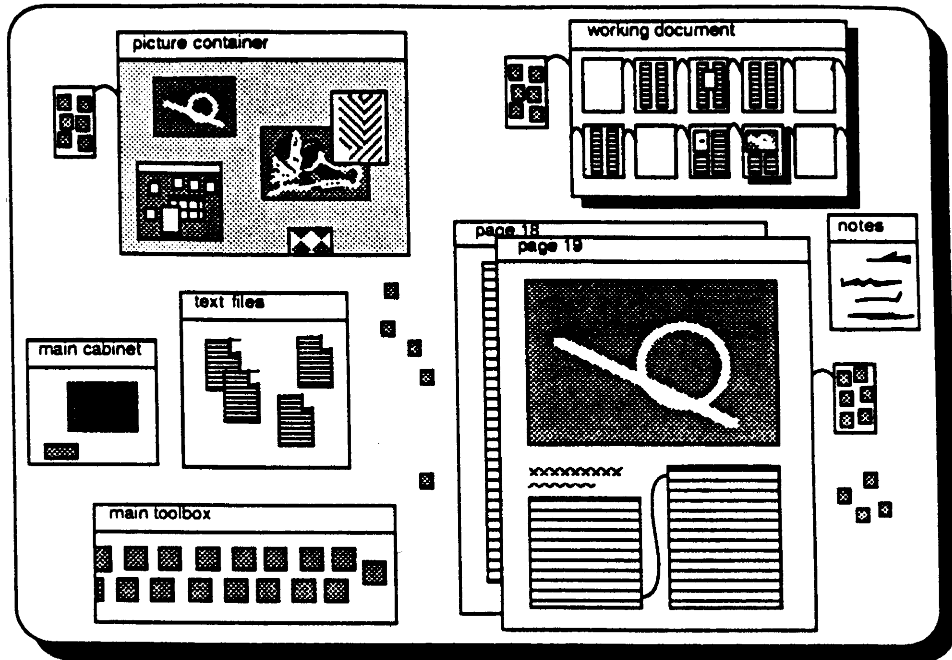
principal container of tools in the system
including Scale, Group, Sequence, Paint, etc.

Figure 2-1

2.1.3 High level application environments can be built from these primitives as the user works.

Examine how the workspaces are setup in Figure 2-2, where a user is designing the page layout for a document. Pages are constructed and composed largely on the right side of the screen, in the Drawing Board area. On the left is the Reference area with containers of images, text and other things. The upper right is used as a Tackboard area where pages of the document are sequenced and there is a notepad for notations and reminders. On the lower left is the Toolbox area. The user has conveniently setup other toolboxes nearby each work area. Miscellaneous tools and other objects are scattered about.

Starting from a new environment as in Figure 2-1, the following scenarios show a user building a Spatial Contexts environment as in Figure 2-2, in the natural course of working on the page layout job.



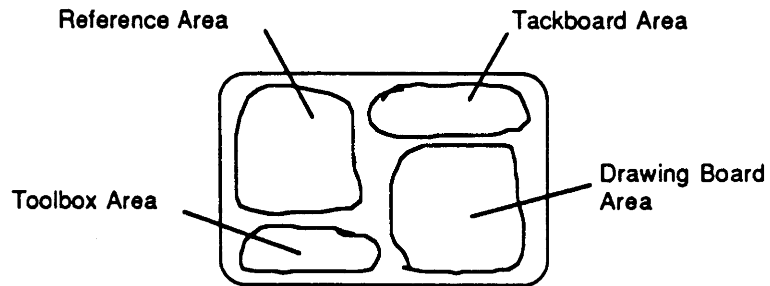


Figure 2-2. Sketch of a Spatial Contexts Environment

2.2 TASKS

2.2.1 Setup a new job by collecting objects into a Job container.

Starting from a new default environment, the user begins work creating a Job container by duplicating an Empty container in the cabinet and naming it "Job". A "Duplicate" tool from the toolbox is used.

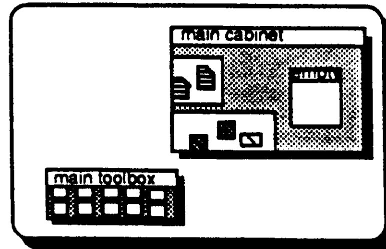
The Job container is then opened using the Open tool. Any objects now moved onto the container fall inside it; whereas when the container was closed, an object moved onto it would just overlap, and not be inside.

The user then begins to browse around the existing dataland database, scrolling, zooming, and entering sub-containers. Pertinent objects are selected and put into the Job container, including picture libraries, text files, and predefined stylesheets for layout formats. Copies of material from earlier jobs may be collected as well.

Figure 2-3 **Setup a new job by collecting objects into a job container**

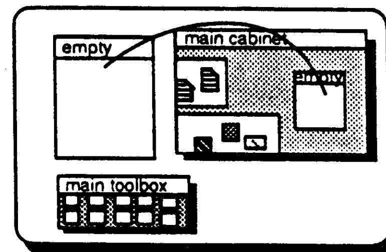
Begin with new environment

Starts out with a Main Cabinet and Main Toolbox, both open.



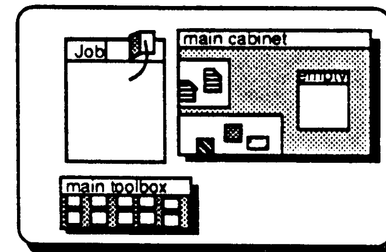
Take an empty container out

Drag a duplicate from the cabinet Object gets proportionally bigger.



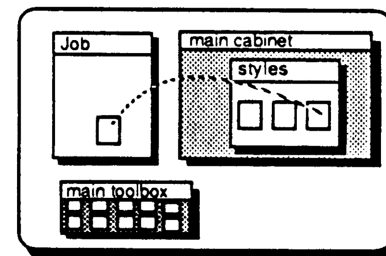
Open container and name it "Job"

Move cursor over title bar to become an Open tool (by attachment).
Use keyboard to rename the container.



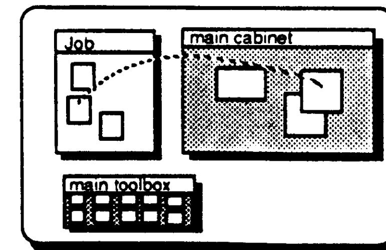
Select a Stylesheet for the Job

Pan through the main cabinet, find the box of existing stylesheets, and copy one into the Job container.



Collect other relevent objects

Other components include text files, images, and a Document container.



2.2.2 Setup workspaces by arranging the job components and tools.

The user begins to arrange the contents of the Job container within the Environment. First, the Main Cabinet is scaled down and put aside. The Job container is enlarged. It contains an empty Document container, a Stylesheet container, and various image and text files.

The Document container is copied from the Job and opened, to be used for collecting and posting page designs as they are developed.

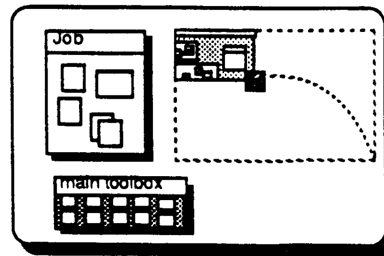
Stylesheets specify the page size, proportion, grid, typography, and other constraints. They become prototype pages when put in a Document. Several duplicates of the Stylesheet are made in the Document container, for prototype blank Pages.

As items are arranged, the user begins working in more than one workspace at a time. Rather than moving across the screen each time to pick up tools, copies can be made and placed near each work area.

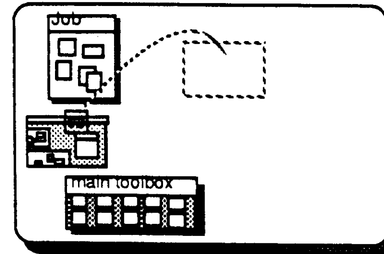
Figure 2-4: **Setup workspaces by arranging components and tools**

Setup a Reference area on the left

Close and scale down the Main Cabinet

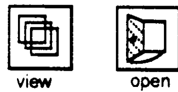


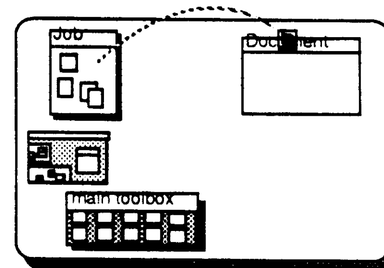
Move the Main Cabinet to the side.



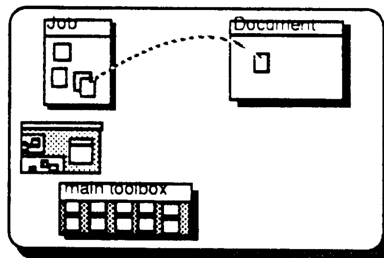
Setup Tackboard area

Copy empty Document from Job and open it.



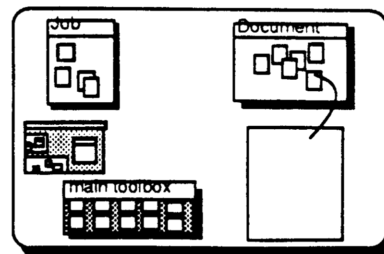


Copy a Stylesheet into the document, as a prototype page.
Make several duplicates for later use.



Setup the Drawing Board area

Make a view copy of a page, dragging it into the environment.
It enlarges to a working size.



2.2.3 Begin sketching a few ideas for the layout.

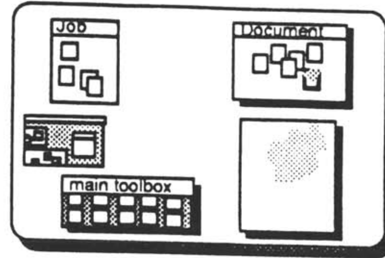
The designer is now ready to begin laying out pages of the document. An empty Page is brought out from the Document and enlarged to a comfortable working size. It is opened, and with various marking tools, the designer begins sketching out spatial formats and content regions.

Additional thumbnails are generated as the designer continues exploring layout ideas. Good sketches are put back in the Document container, and others are thrown out.

Figure 2-5 <u>Begin sketching a few ideas for the layout</u>

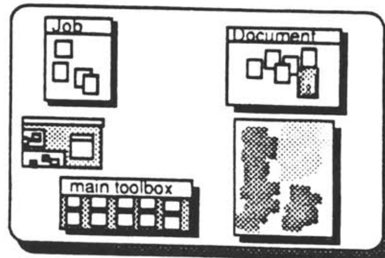
Open page in the drawingboard area

Page on the drawingboard is opened.
Begin sketching.

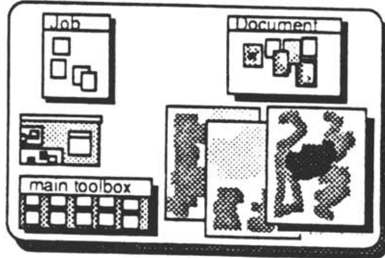


Make a thumbnail sketch

Use a Paint or Charcoal writing tool..
Any changes on the large page view are propogated to the original page.

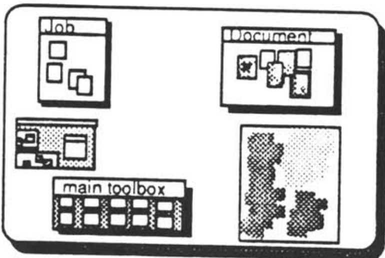


Do a few more



Put away some of the sketches

using the Put-away and Throw-away tools, remove pages from the pile.



2.2.4 Refine a sketch into a layout.

A thumbnail sketch is selected and refined as the designer firms up the sketched regions. The edges are straightened against the Page's default grid using a Straightener tool. This editing tool re-shapes objects according to grid and orientation criteria.

The designer defines content Regions on the Page. Regions are "generic" or empty graphic elements with arbitrary boundaries. Their content can be added later.

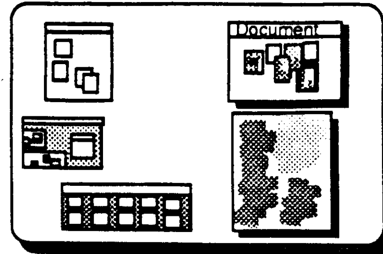
The regions to be used for text are directionally linked, using the Sequencer tool, to indicate the flow of text between columns. When text is put into a region, it flows from the bottom of one column to the top of the next.

As the designer works on these sketches, other pages can be worked on concurrently. Pages are sequenced by linking them together. Contents can be added or removed any time for visualization.

Figure 2-6 **Refine a Thumbnail into a comprehensive page**

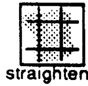
Start with a thumbnail sketch

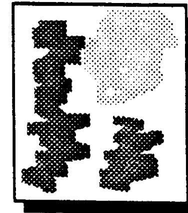
Open the page for editing.



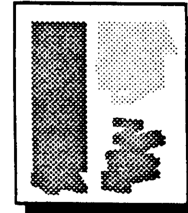
Firm up the sketched regions

Use the Straighten tool to tighten up the region edges against the page's default grid (which could be edited by the user)



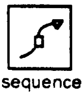


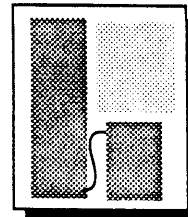
Continue straightening the regions



Define text column sequence

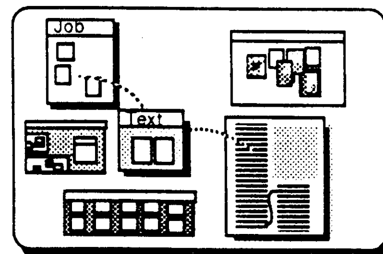
use the Sequential Linking tool to define the flow of text between the columnar regions





Flow text onto the page

Open a text container and select a text file to go onto the page. View the text in the columns.



2.2.5 Edit an image on a page by cropping and color retouching.

An image is selected and copied into the page's image region, and then edited with the Zoom and Contrast tools.

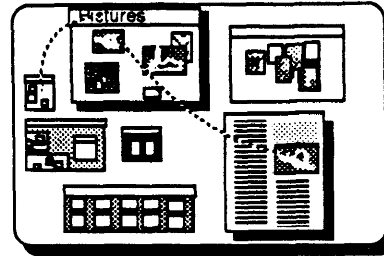
Cropping is performed interactively with the Zoom tool by grabbing the image with the Zoom tool at a location and pulling it. Pulling towards the anchor point (center by default) zooms back, bringing more image into view, whereas pulling away from the point zooms up, stretching the image. Arbitrary anchor points can be defined by changing the tool's anchor link location.

Color and contrast retouching are done with various brush tools. A Contrast tool, for instance, can be opened to adjust its effect, and then be picked up and applied to a pixel image like a brush. As in painting, brushes may completely replace the existing color, or more usually, modify the existing pixels in subtle ways, like watercolor paints or photographic contrast filters.

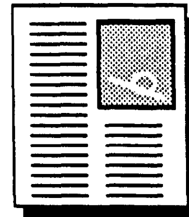
Figure 2-7 Edit an image on a page by cropping and color correcting

Copy a view of an image

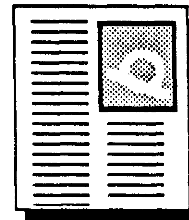
Move pictures container from Job,
open it, and pan through it.
Select an image for the page.



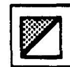
Expand the background window
of the image

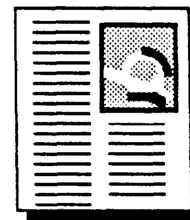


Zoom up on the image with Zoom tool



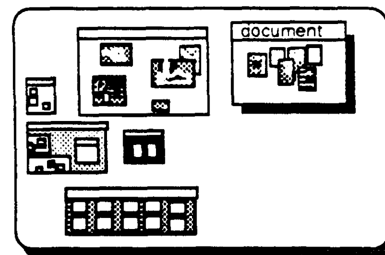
Retouch contrast on the image
using the Contrast brush tool





Put away the page

Close the page, return it to
the Document container.
Continue working by bringing out
another page to work on.



2.3 OBJECTS

2.3.1 Elements are singularly defined units of information with properties and form.

Element types, summarized in Figure 2-8, include graphics, text and images. An element's property values are accessed with an editing tool (discussed below) or by opening the object's property container.

Elements are ordinarily "closed" and treated as a singular unit object. They can be opened so its properties are explicitly edited in it Property Control Panel.

Control Panels are containers attached to a specific object. They contain other elements that control properties of the object acting like meters, knobs, sliders or strings used to constrain or parameterize property values. The properties of an element can be edited by opening it and manipulating the property elements inside its control panel.

Figure 2-9 shows how the color of a polygon is edited using a control panel rather than a coloring tool. The element is opened with an Open tool, causing a property

container to pop up. Properties of a polygon include the list of vertices, the fill style, and the fill color. To edit the color, open the color swash, causing a color space to pop up to choose a color from.

When an element is copied, it can either be Duplicated or Viewed (Figure 2-10). Duplicates are new instances of the original object, with copies of its property values. The two objects are now independent. Views, on the other hand, are instances where property values point directly back to the original object. When one edits one of these objects, the changes are propagated to the other as well.

Hybrid copies can also be made (Figure 2-11). Ordinarily the properties copied with the Duplicate or View tools are all either duplicated or viewed. But, users can edit property containers, or build their own, and determine individually how each property is copied. Taking properties from more than one parent object is multiple inheritance. Defining new types of elements becomes property inheritance programming by direct manipulation.

Figure 2-8. Types of Element Objects and Their Properties

GENERIC ELEMENT
    extent, transformation matrix, filename

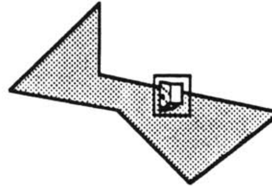
GRAPHICS
- Line
  - vertices, length, angle, color
- Curve
  - control vertices, curve type
- Rectangle
  - width, height, position
- Circle
  - begin arc, end arc, fill, outline
- Polygon
  - vertices, fill, outline

TEXT
- Label
  - string, font, size, char spacing
- Paragraph
  - string, line space, justification

IMAGE
- Lookup
  - buffer, lookup table, resolution, bits
- RGB
  - r buffer, g buffer, b buffer
- Mask
  - operation, transparency

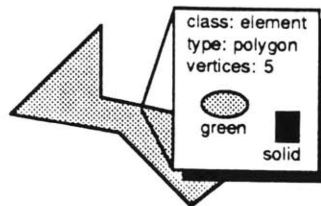
Figure 2-9 **Edit the Properties of a Polygon Element**

Open a Polygon element object



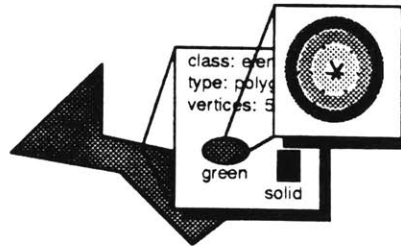
Its property container pops-up

contains displays and controls
for each property of the element



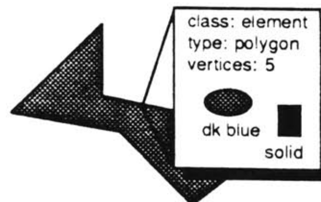
Open color property

pops-up a color space to
select a new color from



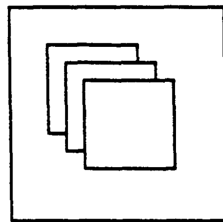
Close

close just the color space
and edit more properties,

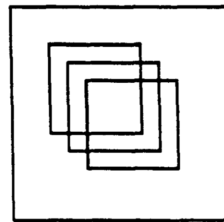


or close the object completely

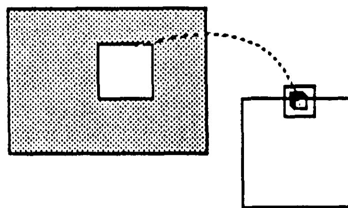
Figure 2-10 **DUPLICATE Copy vs VIEW Copy**

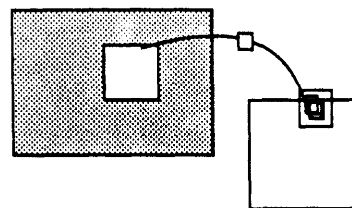


Duplicate



View





Using Duplicate tool,
drag a copy of the object.
Dotted line shows movement
of cursor and object.

Creates a new instance
of the object
with property values
copied from the parent.

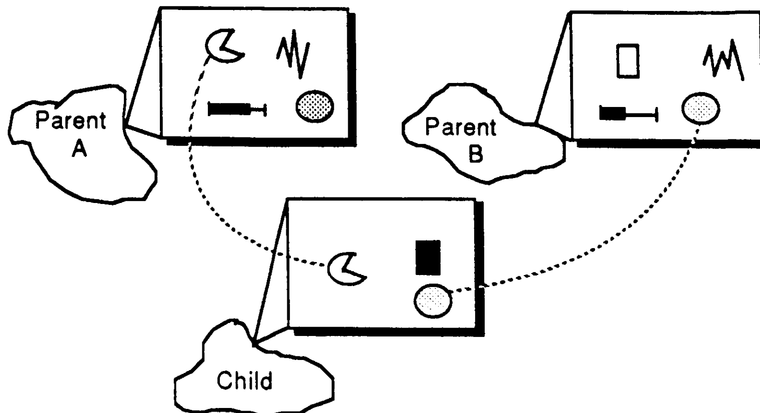
The two objects are
now independent.
Subsequent changes to the
parent's properties
do NOT effect the child's.

Using View tool,
drag a copy of the object.
Link line shows movement
and the view link created.

Creates a view instance
of the object
with property values
directly linked to the parent.

The two objects are
inter-dependent.
Subsequent changes to the
parent's properties
DO effect the child's,
and vice versa.

Figure 2-11 **Property Inheritance Programming Through Direct Manipulation**



Child object inherits some properties from Parent A object, some from Parent B object, and others not inherited at all.

Hybrid copies between Duplicate and View are defined by Duplicating some properties: instance inheritance and Viewing other properties: continuous inheritance

2.3.2 Containers are objects that contain and constrain other objects.

Containment is a natural metaphor for groups and hierarchy (chunking and layering). Containers contain other objects, including other containers. Containers also have constraint properties that affect the behavior of objects and the usage of tools within them. Containers include Environments, Cabinets, Toolboxes, Pages and Property containers (Figure 2-12).

An Environment contains all objects a user is currently working with, and controls the primary user-object interaction. A new default environment has a Main Cabinet for browsing and a Main Toolbox for selecting operations. Like any other object, Environments can be saved, restored and arranged in dataland.

Cabinets are gateways into datalands with spatial and hierarchical organization maps. The Cabinet container acts as a browser or window on the objects currently available in the system. The contents of a cabinet can be changed, as in changing directories in a hierarchical file system. However, not limited to hierarchical organization, containers can be linked in any arbitrary network configuration.

Each object is linked to a "home" container, ordinarily where the object was created. If the object is moved from this container to another, it becomes "contained-by" the new one, but its "home" link remains the same. With the Home tool, a user can transport to an object's home container and jump around the dataland this way (Figure 2-13).

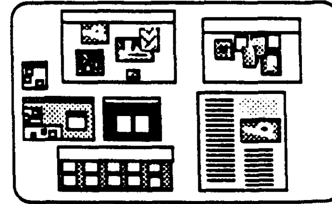
Other containment links can be drawn based on ranges of any number of properties. Containers can be made by searching on logical conditions of properties, such as all images with the word "Boston" in its description. This becomes a direct manipulation interface to a relational database.

Each container has its own coordinate system. In Figure 2-14, the visual consequences of this are illustrated. Objects of a given size will appear larger or smaller with respect to the window boundaries depending on the coordinate system of its container. When a container is placed inside another container, it and all its contents are scaled relative to the new coordinate system.

Figure 2-12 **Types of Containers**

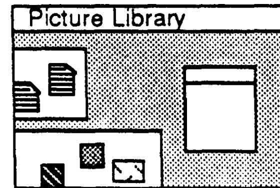
Environment

contains all objects user is currently working with and arranged into workspaces.



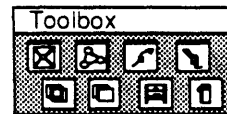
Cabinet

a portal to datalands, contains collections of objects and other containers



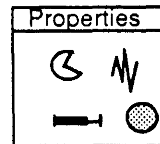
Toolbox

contains tools for user access,; variations include menu box and button box



Property Container

contains displays and controls for properties of objects.



Page

contains text, image and graphics regions, plus grid, format and typographic constraints.

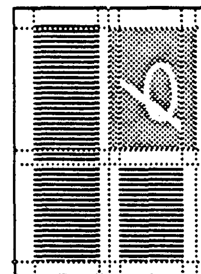
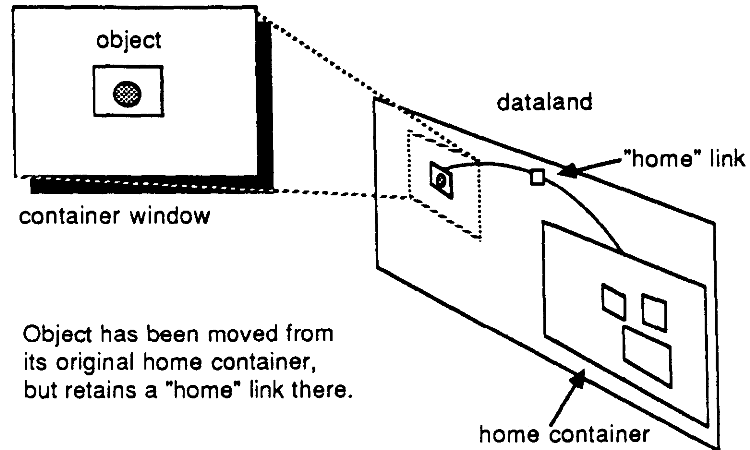
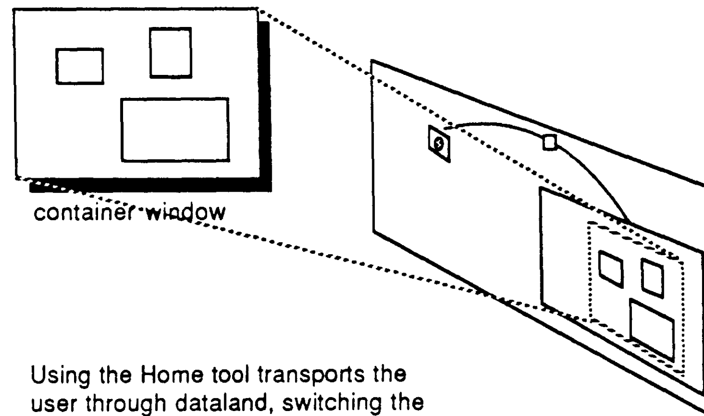


Figure 2-13 Using the Home Tool

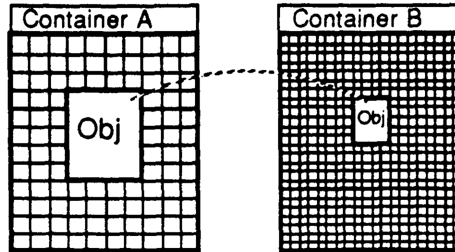




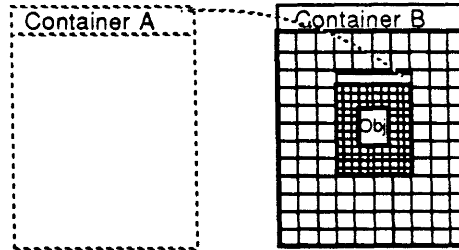
Using the Home tool transports the
user through dataland, switching the
current container to the object's home container.
Use Undo to return.

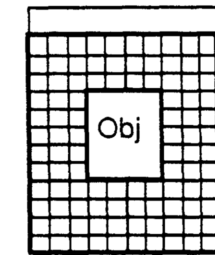
Figure 2-14 Manipulation of Container Objects

Object moved from Container A to Container B is scaled according to the coordinate system of the destination container.

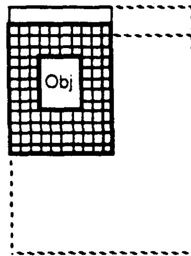


Container A moved into Container B is also scaled to B's coordinates.

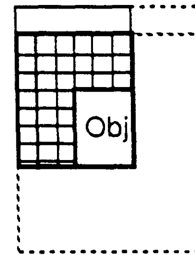




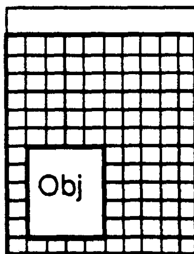
Original Container



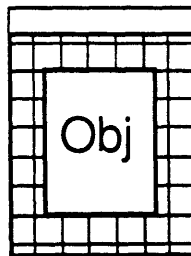
Scale



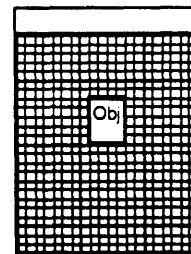
Window



Pan



Zoom Up



Zoom Back

2.3.3 Tools are functional objects one uses to edit other objects and peruse the environment.

Tools are used to perform functions on other objects. There are tools for perusing, editing, and control. Perusing tools (Figure 2-15) include pan, zoom, window and home, are for traveling through datalands and modifying the visible coordinates of containers. Editing tools (Figure 2-16) include move, scale, label, and sequence, are for interactively modifying objects' properties and links. Control tools (Figure 2-17) include duplicate, view, put-away, and help, are for object control and maintenance.

There are three ways to select tools: by picking one up, by moving the cursor over an attachment site, and by selecting it from a menu or button box (Figure 2-18).

A tool can be picked up using the PICKUP button on the mouse (Figure 2-19), and begins tracking the mouse. The DO button applies the current tool to the object beneath it. The UNDO button reverses the previous application of the tool to the object.

Instead of picking them up, tools can be attached to an object and used like a "handle" or border function. Once attached, these tools are automatically picked up when the cursor moves over the attachment site. For instance, closed containers ordinarily have an Open tool attached to their title bar. When the cursor moves over the bar, it will temporarily switch to the Open tool until the cursor is moved away. If the user presses the Do button, the container will be opened allowing him to edit its contents.

Suppose a user wants to pan around a dataland container. He picks up the Pan tool and applies it to the opened container, "pulling" the contents with the cursor (Figure 2-20). Or instead, copies of the tool could be attached to the border of the container and constrained to scroll in only one or two directions. Figure 2-22 shows how a MacIntosh MacDraw-like window borders can be constructed from Spatial Contexts attached tools.

Toolboxes, Menu boxes, and Button boxes contain tools organized by the user but constrained in specific ways (Figure 2-23). Toolboxes contain tools that the user can pickup, remove, use, and then drop it anywhere in the environment. Tools selected from a Menu are view copies, automatically "put away" to the menu when dropped. Button

boxes are like menus, where tool positions correspond to physical function keys on the keyboard for picking up the tools.

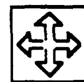
Tools can be applied to other tools. Functional extensions to tools would allow them to be opened, edited and combined as graphical programs. Toolmaking becomes a means for system extensibility by the user, by combining tools to make macro tools.

Figure 2-15 **Perusing Tools**

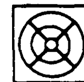
OPEN
open an object, allowing editing its contents and/or properties

CLOSE
close an object, unifying its contents/properties

WINDOW
modify the clipping boundary size of a container and on-screen viewport synchonously.



PAN
modify the window boundary coordinate position of a container

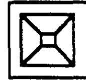


ZOOM
modify the clipping boundary size of a container while the on-screen size remains constant.

HOME
switch to the home container of an object

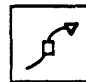
Figure 2-16 **Editing Tools**

LABEL
create/edit an object's label or title.
(string, font, background)

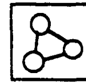


SCALE
edit the size of an object.
(preserve aspect ratio, grid gravity)

MOVE
edit the position of an object
(grid gravity)

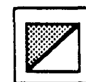


SEQUENCE
define/edit sequential links between objects
(from object, to object, link label, enumeration)



GROUP
define/edit unordered groups or sets of objects
(selection attribute range)

PAINT
mark objects with hand strokes
(color, size, pattern)



CONTRAST
retouch contrast of a bitmap object
(factor)

STRAIGHTEN
allign object boundaries to grid

Figure 2-17 **Control Tools**

DUPLICATE COPY
create a new instance of an object

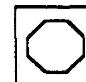
THROW AWAY
remove an instance of an object

VIEW COPY
create a continuous inheritance instance of an object

PUT AWAY
remove a view copy of an object

COLOR PALETTE
color property editing

HELP
inquire object capabilities and description



QUIT
exit current environment and save it for later retrievals

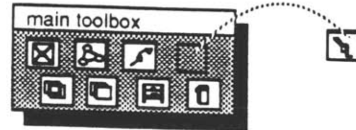
Figure 2-18 Ways of Using Tools

Pickup and Do

Use mouse PICKUP button to grasp a tool; it becomes a cursor tracking the mouse.

Use DO button to apply the tool to object beneath it.

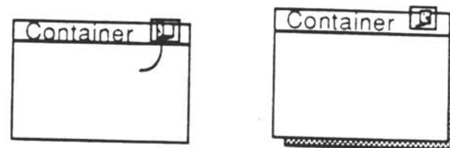
Use UNDO to restore object.



Attachment

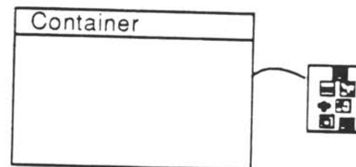
Moving the cursor over an attachment area temporarily picks up the attached tool.

Here, the container's OPEN tool is attached to the title bar. Pressing DO opens the container, a CLOSE tool replaces Open for easy re-closing.



Menus and Panels

When the container is moved, attached toolbox is also moved. Toolboxes can be used like menus and control panels.



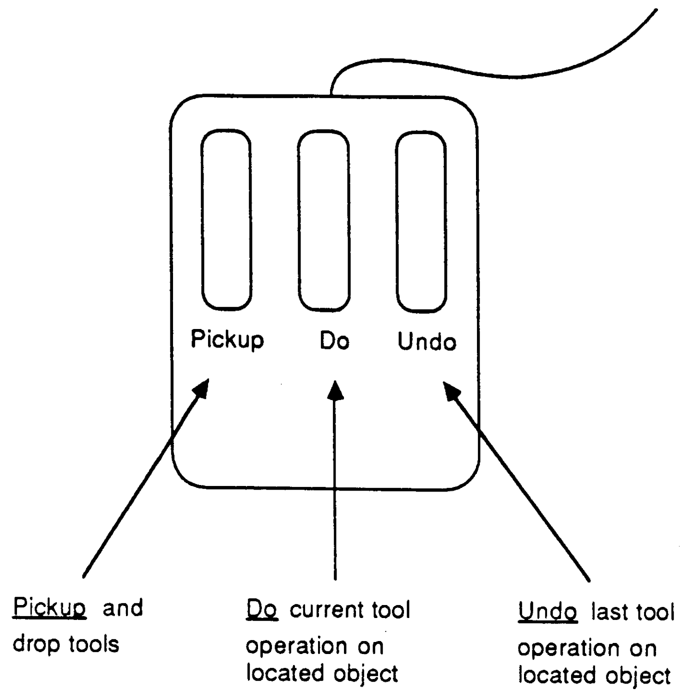
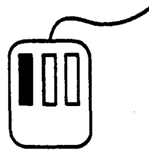


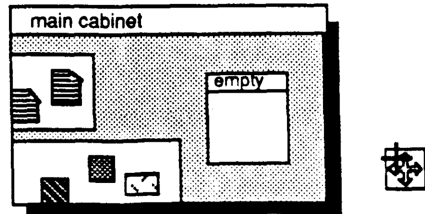
Figure 2-19. Standard Mouse Buttons for Using Tools

Figure 2-20. USING THE PAN TOOL

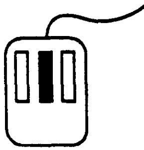
PICKUP Pan tool

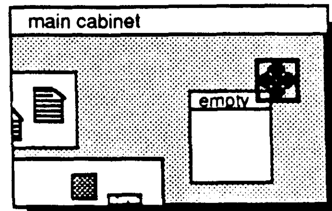
by moving cursor to its icon, and using the PICKUP button on the mouse



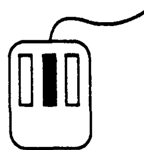


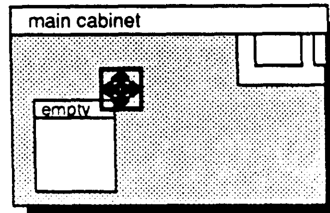
Begin using the tool by pressing DO to grab a location in the container



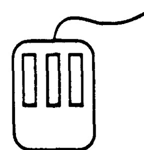


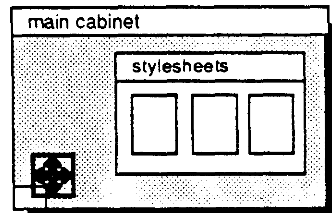
Continue pressing DO pulling with the Pan tool



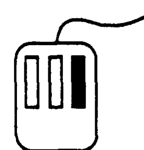


Continue pressing DO pulling with the tool, and then release the button when done.





UNDO will return the container to its previous state.



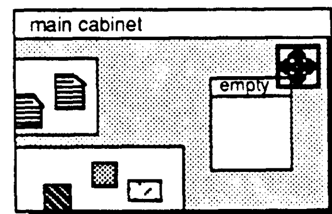
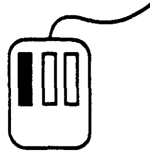
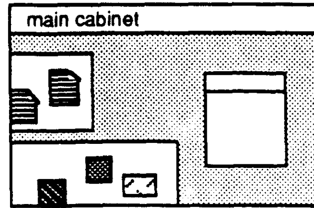


Figure 2-21. USING THE SCALE TOOL

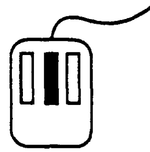
PICKUP Scale tool

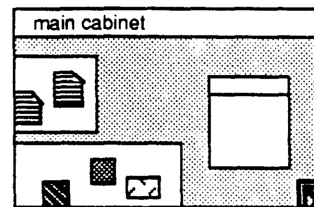
by moving cursor to its icon, and using the PICKUP button on the mouse



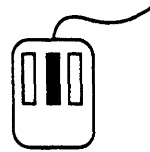


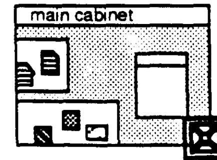
Begin using the tool by pressing DO to grab a location in the container



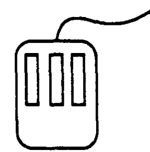


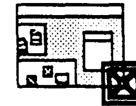
Continue pressing. DO Scale is achored at opposite corner, if not otherwise specified.



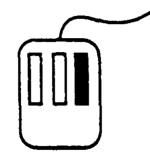


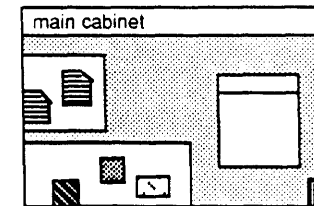
Continue pressing DO pulling with the tool, and then release the button when done.

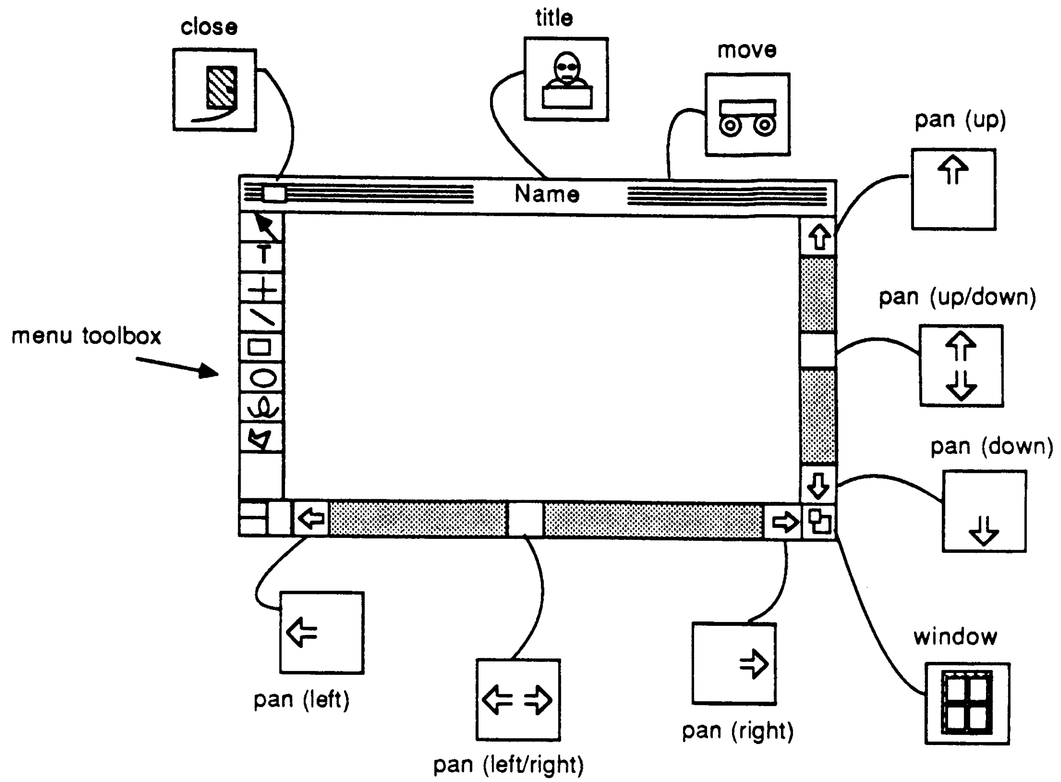




UNDO will return the object to its previous state.







A MacIntosh MacDraw-like window
built from a Spatial Contexts container
with attached tools.

The Pan tool has been constrained
to limited directional movement.

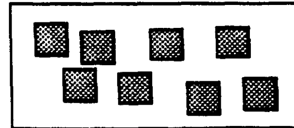
Figure 2-22

Figure 2-23. Types of Tool Containers

Tool Box

Pick up tool out of container,
use it and
drop it anywhere in the environment

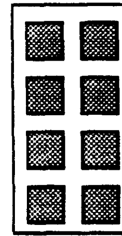
Position in container is unconstrained.



Menu Box

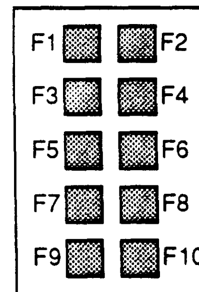
Pickup view copy of tool
use it;
dropping it puts it back directly
to the menu box container.

Position in container is fixed.



Button Box

Can use like Menu Box, plus
tool positions correspond to physical
function keys on the keyboard.
Pressing function key picks up
corresponding tool.



Section 3. ARCHITECTURE

3.1 SYSTEM ORGANIZATION

3.1.1 Spatial Contexts software is organized in a layered architecture with objects.

The Spatial Contexts user interface environment is built on a color graphics computer workstation and an object oriented software architecture. Figure 3-1 shows the components of the system are built on top of each other in a layered organization.

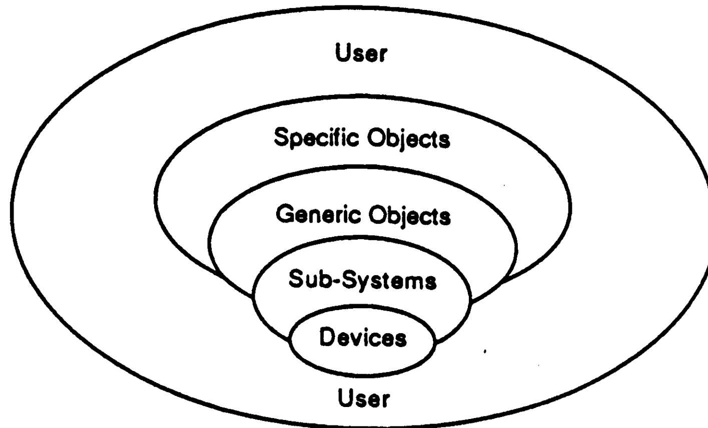
From the bottom-up, there are the physical devices which interface the simulated work environment with the physical world. On these are built the sub-system libraries which create a higher level functional interface to the devices, and assist in system resource management. The sub-systems include an object management library that allow the definition of object classes. Three principal classes are defined: elements, containers, and tools, from which application level objects are defined.

With object-oriented programming, each piece of a system is treated as a separate autonomous object made of

private data (properties) and the operations supported on that data (methods). The system has independence from the objects it contains, such that new objects may be added and existing ones modified without major code modifications. Figure 3-2 summarizes the advantages of this paradigm.

An essential component of the system is the user, who works in the physical domain by interfacing with physical devices, and in the problem domain by interfacing with high level simulated objects.

Figure 3-1 System Architecture



SPECIFIC OBJECTS

Element:	graphic, text, image
Container:	environment, cabinet, toolbox, page
Tool:	peruse, edit, control

GENERIC OBJECTS

Attributes:	class, type, name, form, history
Methods:	maintenance, display, tools
Links:	home, contained-by, next, previous

(Figure 3-1 continued)

SUB-SYSTEM LIBRARIES

Objects:	control, inheritance, select, overlap
Display:	render, 2d, color, font
Input:	locate, cursor, border
Memory:	pixel buffers, tiles, lists
Database:	elements, containers, tools, relations

DEVICE INTERFACES

System:	cpu, memory, ports, Dos
Graphics:	frame buffer, processor, monitor
Interact:	mouse, keyboard

Sound:	voice in, voice out, music synthesis
Soft Store:	video tape, disks
Hard Store:	printer, typesetter, photo
Communication:	network, asynch, modem

Figure 3-2. Features of Object-Oriented Programming

ENCAPSULATION
- Objects are private data and operations on that data

MODULARITY
- Software is malleable, reusable and enhanceable

EVOLUTION
- System independence from the objects it contains

CLASSIFICATION
- Objects know their type explicitly

MESSAGING
- Objects know what operations they can perform

METHODS
- Objects know how to perform their operations

INHERITANCE
- New objects are defined from existing ones

3.1.2 The system is demonstrated on a color graphic personal computer workstation.

Spatial Contexts is implemented in the "C" language on an interactive color graphic workstation configured from an IBM personal computer, a YODA graphics system, and a mouse input device (Figure 3-3).

The system unit is an IBM PC/XT personal computer running under MS-DOS 2.0 operating system. The typical system has up to 640k bytes RAM memory, an 8087 arithmetic co-processor, two 10 megabyte hard disk drives and a floppy disk drive.

The graphics system is an experimental IBM YODA frame buffer and graphics processor, that plugs directly into the PC chassis. [SHOLTZ'85]. The frame buffer has 640 x 480 pixel resolution displayed, plus an additional 320 scan lines of off-screen image buffer. Each pixel is eight bits and indexes into a 256 entry color lookup table with 24 bits output (one byte each red, green, and blue) to a low cost RGB color monitor.

The YODA has a programmable bitslice processor that provides high speed graphics, imaging and anti-aliasing support. Additional custom microcode was developed for the Spatial Contexts system that addresses performance requirements of the project, including the scaling of bitmap images by arbitrary scale factors in interactive real time.

The principal interaction device is a three button mouse. A character keyboard is used primarily to enter text.

Other devices from the Visible Language Workshop laboratory that could be integrated into the workstation configuration include other display systems, interaction devices, sound instruments, hard and soft copy storage, and communications interfaces, as illustrated in Figure 3-4.

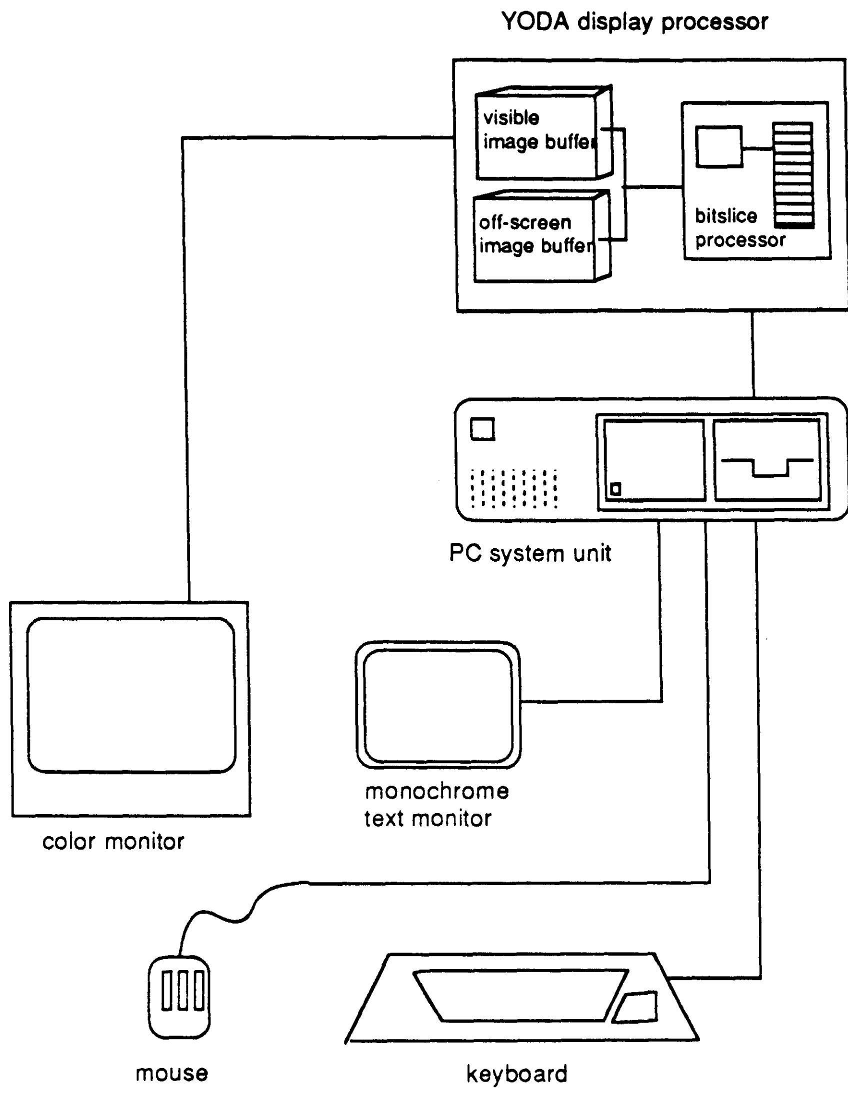


Figure 3-3. IBM PC/YODA based Spatial Contexts workstation configuration

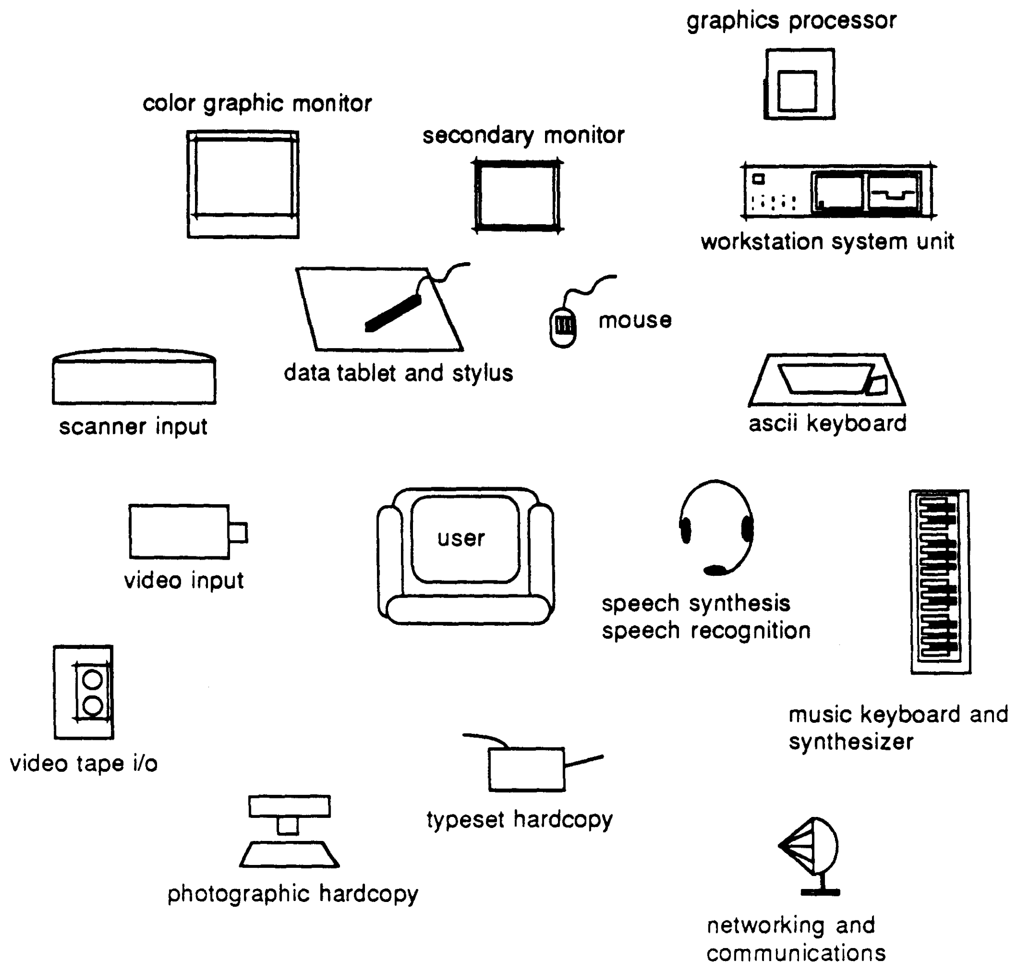


Figure 3-4. Devices in the VLW Lab

3.1.3 Application level objects are built from graphic elements, containers and tools.

There are three principal classes from which all objects are derived. Element class objects have geometric tranformations and property container attributes. Container class objects have additional information about their borders, attached tools, and "contains" links. Tool class objects include its icon property and functional constraints.

Figure 3-5 shows the taxonomic relationship between the generic object of each class and its specific sub-class types. Through this organization, objects lower in the hierarchy can inherit properties and methods from its parent, reducing redundency of code and facilitating new, similar objects to be added to the system.

The procedures in Figure 3-6 are the methods installed in each object's method dictionary. Methods include those for creating new instances, loading from a file, saving to a file, drawing, and each of the tool operations. New methods may be installed by system programmers.

Figure 3-5. Object Taxonomies

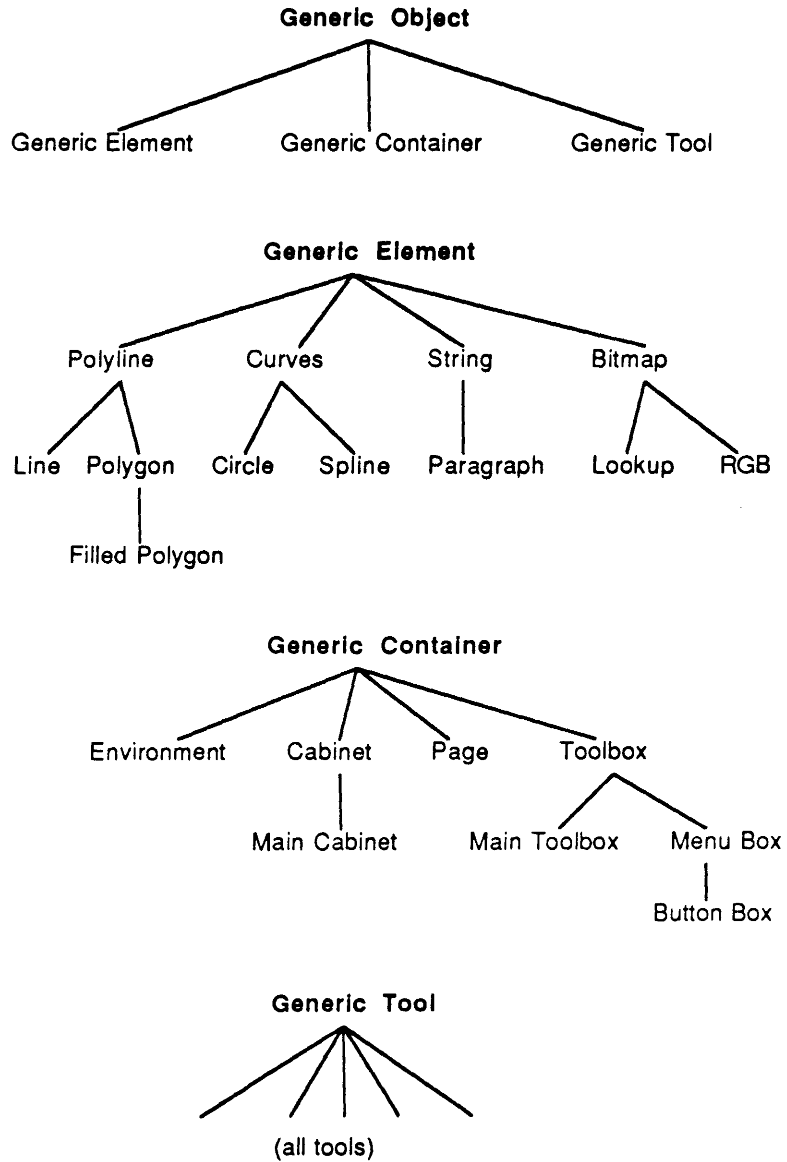


Figure 3-6 SPECIFIC OBJECT METHODS

```
mak_env         create new enviroment instance
mak_cont        "           container
mak_tool        "           tool
mak_line        "           line
mak_rect        "           rectangle
mak_circle      "           circle
mak_pmd         "           bitmap
mak_string      "           character string

init_env        init environment object from file
init_cnt        "    container
init_tool       "    tool
init_ln         "    line
init_rect       "    rectangle
init_cir        "    circle
init_pmd        "    bitmap
init_string     "    string

save_env        save environment object to file
save_cnt        "    container
save_tool       "    tool
save_ln         "    line
save_rect       "    rectangle
save_cir        "    circle
save_pmd        "    bitmap
save_string     "    string
```

i_DrwEnv	draw environment
i_DrwCont	" container
i_DrwTool	" tool
i_DrwLine	" line
i_DrwRect	" rectangle
i_DrwCircl	" circle
i_DrwPmd	" bitmap
i_DrwStr	" string
t_mv_cont	tool move container
t_mv_tool	" tool
t_mv_elem	" element
t_mv_pmd	" pmd
t_cp_cont	tool copy container
t_cp_tool	" tool
t_cp_elem	" element
t_sc_cont	tool scale container
t_sc_tool	" tool
t_sc_elem	" element
t_sc_pmd	" bitmap
t_wd_cont	tool window container
t_wd_pmd	tool window pmd
mak_magnify	create new magnify tool
do_magnify	apply magnify tool

3.2 OBJECT ARCHITECTURE

3.2.1 All objects have the same basic structure, and are classified by their Class and Type.

The high level objects specified by the scenarios in the Section 2 and described in the previous section are built from generic objects. All objects in the system have the same basic internal structure, as in Figures 3-7 and 3-8.

Figure 3-7 Generic Object Properties

ID	- instance identifier
CLASSIFICATION	- class and type of object
CLASS PROPERTIES	- sets of properties for class & type
PROFILE	- name, date, filename
VISUAL FORM	- icon, size, color
METHOD DICTIONARY	- maintenance and editing procedures
LINKAGES	- linkages to other objects
HISTORY STACK	- stack of recent states for undo

When a new object is created, memory is allocated for each of its property values and a unique ID is assigned which corresponds to it internal memory address. An object is always referenced by its ID.

Every object has a CLASS and a subclass, or TYPE identifier. These enumeration values define what kind of object it is. For each class there is a structure of class-specific properties, and for each type within a class there is a structure of type-specific properties. Class and type descriptions therefore act like templates with slots to be filled with values.

The initial state of the system provides generic instances of each object class/type: a prototype object with default values used as a top level parent. New objects can be derived from existing ones, defining a parent-child link between them. The new object can then have its own property values and methods, or inherit them from its parent, recursing to the top level parent if necessary.

Objects can be located by property searches that find all objects meeting certain property value criteria. Complex logical expressions can be used for searching and collecting objects.

Figure 3-8. Data Structures for Generic Objects

```
typedef struct {

     int       Class;            /* class enumeration */
     universal Class_data;       /* class properties */

     int       Type;             /* sub-class enumeration */
     universal Type_data;        /* sub-class properties */

     Profile   Profile_data;     /* maintenance properties */
     Form      Form_data;        /* device level properties */

     proplist  Methods;          /* (message, method) pairs */
     proplist  Links;            /* (linkage,object) pairs */
     queue     History;          /* state history for undo */

} GenericObject;
```

```
("universal" is a pointer to an arbitrary data structure)
("proplist"  is a linked list of (name, value) pairs)
("queue"     is a circular queue)
```

3.2.2 An object's Profile holds maintenance properties.

All objects have a Profile record with internal maintenance properties for names, dates, and addresses (Figure 3-9). Objects can have a character string name or title, and a verbal description of arbitrary length for documentation and semantic identification. Dates include the creation date and last update. Data residency properties include archiving and/or networking information.

Figure 3-9 Profile Properties

- Title/Name
- Verbal Description
- Creation Date, Update
- File Name
- Data Residency

3.2.3 An object's Form defines device level properties for high speed display and interaction.

An object's draw method uses the object's class/type properties to render it on the screen, perhaps mapping from a world coordinate space to pixel coordinates. Then the draw method will update the object's Form properties with the physical device information (Figure 3-10). This data is used when operations must be done in screen coordinates and pixels for high speed interactive performance.

The pixel extent is the enclosing rectangle in screen coordinates, for instance for locating objects with the cursor or block transfers (bitblt'ing) to drag it on the screen. Overlap information includes the object's pixel windows or corner-stitch tiles. The undersave buffer may contain the pixels that lie beneath the object for fast un-drawing.

Figure 3-10 Form Properties

Pixel Extent

Overlap Information

Undersave Buffer

3.2.4 Links define qualified associations between objects, include "isa", "home" and "contained by".

Specific types of links can be drawn between objects. These links are given a symbolic name and verbal description, establishing semantic connections.

The "isa" link points to the parent object through which inheritance occurs. The default isa link is the prototype object of its class. Multiple qualified isa links permit multiple inheritance, with the links specifying what properties or methods are being explicitly linked from that parent.

The "home" link points to the home container of the object. The default home container is where the object was created. This link is used and edited by the Home tool. The "contained-by" link points to the current container of the object and can be edited with the Move tool. The default contained-by link is the object's home container.

Like the method dictionary, the link dictionary is a linked list of link-name/ link-to pairs, where link-name is a character string and the link-to is an object ID.

3.2.5 Methods are operations an object performs in response to messages.

Operations, like setting and inquiring about an object's property values, are affected by signaling the object by sending messages. The corresponding response method is looked up in a message-method dictionary and is performed. Objects have standard maintenance methods (Figure 3-10), plus tool methods for each tool that operates on the object.

Figure 3-10 General Maintenance Methods

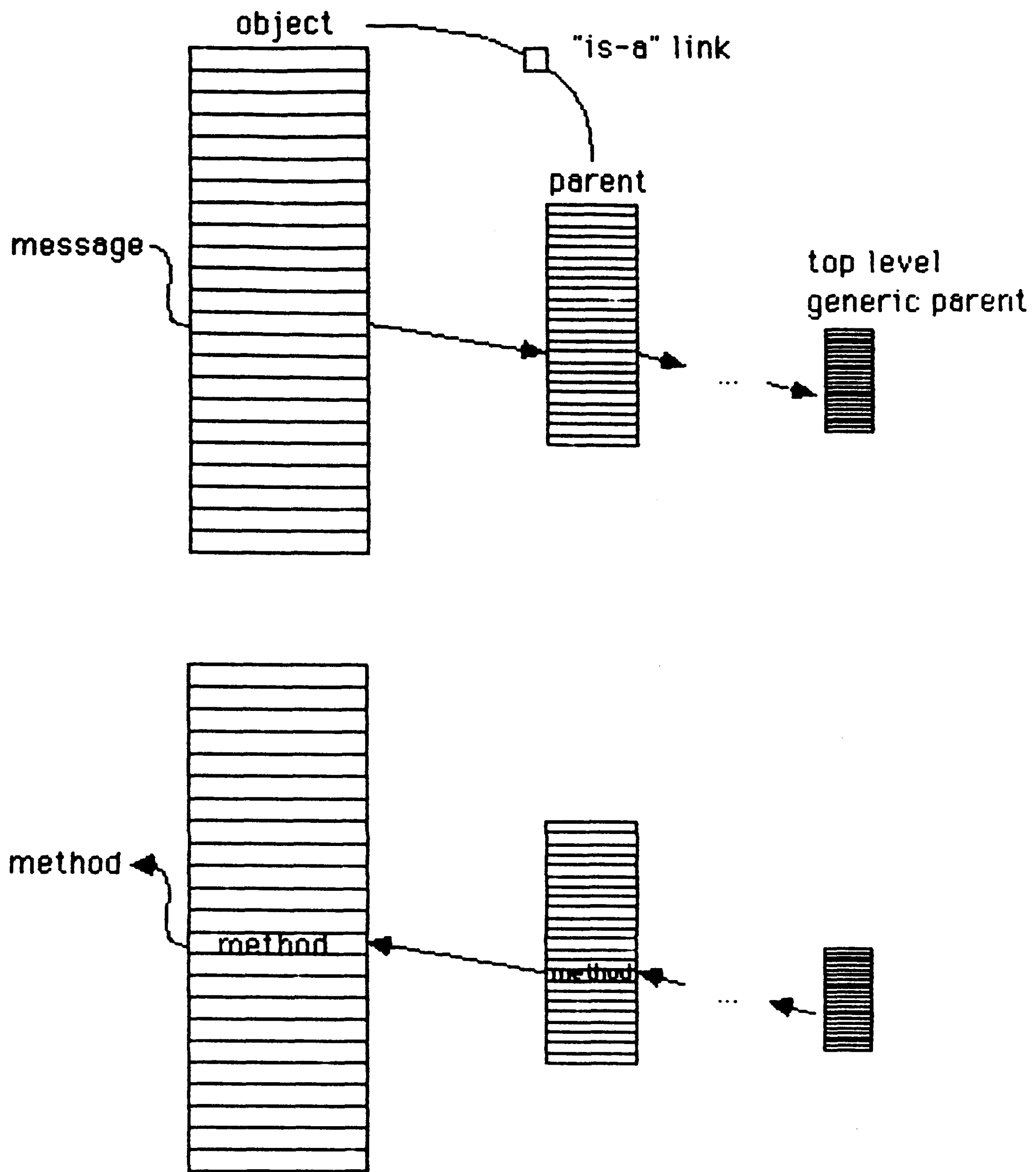
CREATE	- generates a new instance object with default property values
SAVE	- save an instance of an object to disk
LOAD	- retrieve a saved instance object, creating a new instance with previous values intact
DRAW	- draw the graphic representation of the instance object
DELETE	- destroy the instance object

Each object has a method dictionary where method functions can be looked up by message names. If a method is not found, it looks to the object's parent, recursing until one is found. Once a method is found, it is placed in the object's local dictionary and bound (Figure 3-11).

The method dictionary is implemented as a linked property list of message-name/ method-pointer pairs. The message is an enumeration variable. The method pointer is a pointer to a pre-compiled C function.

Each method has up to three operations: Start, Do, and Finish (Figure 3-12). This partitioning of the method allows startup and cleanup operations to be separated from the main function. This way, Do can be repeatedly signaled in the body of a loop or by a clock (Figure 3-13).

For example, MOVE is a method activated by the Move tool to drag an object on the screen. Typically, there is some preparation necessary before interactively moving the object, like allocating temporary bitmap storage. When the user presses the mouse button, Start Move is invoked. Then within the main loop, each time the mouse moves, Do Move is signaled until the user releases the button and Finish Move is called to clean up. Other methods, like Draw, may not need a Start and Finish phase and only Do is defined.



MESSAGE-METHOD Dictionary Lookup
with inheritance and lookup-time binding.

Figure 3-11

Figure 3-12 Method Entry Points

START	Setup Operation for the Given Object
	example: START MOVE TOOL
	signal object to push its state history allocate temporary bitmap and draw the pixels under the object into it
DO	Perform One "Tick" of the Operation
	example: DO MOVE TOOL
	calculate changed areas of screen save newly covered pixels bitblt object to new position restore uncovered pixels
FINISH	Clean Up Operation
	example: FINISH MOVE TOOL
	free allocated buffers signal object to update its position

Figure 3-13 Environment Main Interaction Loop

```
PICKUP tool:
                message = tool.message

Press DO:
                object id = which object under cursor
                method = lookup( tool.message, object id )
                method.START( object id )

while DO still pressed {
                locate( x, y ) from mouse
                method.DO( x, y )
                }

Release DO:
                method.FINISH()
```

3.3 SUB-SYSTEM FUNCTIONS

Sub-systems provide the underlying support for implementing objects. These robust function libraries provide a high level software interface to devices, algorithmic control, and system utilities which include functions for object management, color graphic display, high speed interaction, flexible memory management, and database maintenance.

3.3.1 MAIN SYSTEM DRIVER

The main system driver procedures are the top level functions for starting up, running, and shutting down a Spatial Contexts system. The procedures are listed in Figure 3-14.

Figure 3-14 Main Level Procedures

```
new_environ     generate new default environment
set_tables      setup method dictionary & inheritance tables
obj_init        generic object initialization
display_init    initialize display systems

env_interact    environment interaction control
env_save        save environment to filename
env_setup       setup environment from filename
```

3.3.2 OBJECT MANAGEMENT SUBSYSTEM

3.3.2.1 OBJECT INSTALLATION

The object installation functions are for setting up the system tables, parentage relationships and inheritance mechanisms.

Figure 3-15 Object Installation Procedures

Procedure	Description
InitTables	initialize system tables
Install	install message/method in method dictionary
InheritMethod	inherit method from parent
LookUp	lookup method in dictionary
SetParent	intall object in parentage table
GetParent	lookup parent in parentage table

3.3.2.2 GENERIC OBJECT CONTROL

These functions are for control and manipulation of generic objects.

Figure 3-16 Generic Object Control Procedures

CreateGenObject	create generic object
SetObjPos	move object to new position
GetBndRect	get bounding rectange of object in pixels
EraseObj	erase object
UndoObj	restore object state
SaveObj	save arbitrary object to a file
LoadObj	load arbitrary object from a file
switch_cont	switch the "contained by" link of an object
prep_render	prepare renders for a hierarchy of containers
set_pmd	get bound rectangle size from bufmgr

3.3.2.3 OBJECT PROPERTY SEARCH

These functions search for objects meeting specified property value criteria, such as whether a point is within an object's boundaries.

Figure 3-17 Object Property Search Procedures

is_near	is (x,y) near an object
which_obj	return which object matches
which_tool	return which tool matches
which_cont	return which open container matches

3.3.2.4 TOOL SUPPORT

These are used by the environment interaction loop for handling the user interface with tools.

Figure 3-18 Tool Support Procedures

pickup_tool	pickup tool, make it active
bgn_pickup	initiate tool pickup
drop_tool	release tool, make it inactive
reset_cursor	restore previous cursor
EraseTool	restore pixels from under
SaveUnderTool	buffer pixels under object
GetToolSave	

3.3.3 DISPLAY SUBSYSTEM

3.3.3.1 RENDER DESCRIPTION RECORDS

Render records define the mapping of data from one coordinate space to another, allowing clipping and scaling of graphic primitives. Basically, it is like laying a new coordinate system on top of a pmd (YODA pixel matrix descriptor), and then allowing child renders to be defined relative to existing ones. Conversion between renders and pmd's is simple, so one can work with either (or both) structures as necessary. The render structure is used by the enhanced display primitives (d_). These functions expect float data as opposed to pmd pixel coordinates which are all integer precision.

Rendering is the operation of generating a picture from data. When rendered, graphic primitives are first transformed by the current geometric transformation matrix, then clipped, mapped to pixels, and displayed. A render record is a data structure describing the mapping from one coordinate system (clip bounds) to another (port bounds).

The 'clipping' boundary defines the rectangular boundary of a coordinate space. Any data inside the clip bounds will be drawn, any outside will be clipped. Defines the coordinate system of the area.

The 'port' boundary defines rectangular boundary of where a render is mapped. Defines the size and position of the view port in its parent's coordinates.

The 'effective' boundaries are truncated boundaries of a render. If the clip bounds, port bounds, and/or pmd for a render extends beyond the clipping bounds of its parent, the boundaries will be truncated to only the visible portion.

Figure 3-19 Render Data Structure

```
typedef struct _render {

/* family links */
struct _render *mapto;   /* parent render */

int        mapfmcount;   /* # children */
struct _render *mapfm[RENDMAX]; /* list of children */

/* geometric transformation */
bool       isidentity;   /* T=xform is reset to identity */
tmat       xform;        /* current transformation matrix */

/* boundaries */
rectangle clip;          /* clipping boundary */
rectangle port;          /* port boundary on mapto render */
rectangle bounds;        /* pixel boundary on initial pmd */

/* effective boundaries, from propagation of renders */
rectangle Eclip;         /* effective clipping bounds */
rectangle Eport;         /* effective port */
rectangle Ebounds;       /* effective pixel boundary */

/* transformations from here */
tmat       tomapmat;     /* transform to "mapto" */
tmat       topixmat;     /* transform to initial pixels */
tmat       topmdmat;     /* transform to "rpm" pmd */
tmat       dismat;       /* display xform = xform*topmd */

/* transformations to here */
tmat       fmmapmat;     /* transform from mapto to here */
tmat       fmpixmat;     /* transform from pixels to here */
tmat       fmpmdmat;     /* transform from rpm pmd to here*/

/* pmd's */
pmd        initpm;       /* initial pixel matrix */
byte       startbit;     /* relative bit planes */
byte       depth;
bool       visible;      /* F= ports completely truncated */
pmd        rpm;          /* render pixel matrix */

}               render;
```

Figure 3-20 Render Control Procedures

r_init	initialize a render from a pmd
r_create	initialize a child render from existing one
r_inside	ask if a point is inside a render
r_fromscreen	map a point from pixels to world coordinates
r_toscreen	map a point to screen coordinates
r_setmat	change instance transformation
r_setclip	change window clipping boundaries
r_setport	change port bounds in render coordinates
r_setplanes	change bit planes
r_pmdport	change port bounds in pixel coordinates
r_pmdwindow	change window clipping in pixel coordinates
r_switch	change parent of a render
r_update	propogate render map transformations
r_askmat	ask current instance transformation
r_askclip	ask current clip bounds
r_askport	ask current port bounds
r_askbounds	ask current pixel bounds
r_askEclip	ask effective clip bounds
r_askEport	ask effective port bounds
r_askEbounds	ask effective pixel bounds
r_askpmd	ask render pmd
r_clear	erase render pmd

3.3.3.2 EXTENDED GRAPHIC PRIMITIVE DISPLAY FUNCTIONS

These extended display primitives use renders rather than YODA pixel matrix descriptors (pmd's). All primitives are specified in the world coordinate space defined for the render. All coordinates are floating point numbers. Each function transforms the primitive by the render instance transformation, then clips the points, and maps from the window to screen coordinates.

Figure 3-21 Extended Display Primitive Procedures

d_line	display line
d_rect	display rectangle outline
d_frect	display filled rectangle
d_pline	display polyline
d_polygon	display polygon
d_fpolygon	display filled polygon
d_circle	display circle outline
d_fcircle	display filled circle
d_string	display short text string
d_image	display bitmap image data
d_bitblt	bitblt from one render to another
d_setaa	set anti-aliasing parameters
d_setline	set line attributes
d_setfill	set fill attributes
d_setimage	set image dimension attributes
d_setscan	set image scan attributes
d_setchar	set character attributes
d_settext	set text paragraph attributes
d_setfont	set font selection attributes

3.3.3.3 GEOMETRIC TRANSFORMATION FUNCTIONS

These functions manipulate geometric transformation matrices for two-dimensional (x,y) coordinate data. The matrices are 2x3 where column 1xN transforms the x coordinate, and 2xN transforms the y coordinate (the third column is assumed to be 0,0,1). For an explanation of the principles governing these transformations, see the basic computer graphics texts by Newman and Sproull, or Foley and VanDam.

Figure 3-22 Geometric Transformation Functions

m2_identity	reset transformation to identity matrix
m2_move	move by (dx,dy)
m2_scale	scale by (sx,sy) about origin (ox,oy)
m2_rotate	rotate by angle about origin (ox,oy)
m2_mult	concatenate two transformations
m2_copy	copy a transformation
m2_apply	transform point (oldx,oldy) to (newx,newy)
m2_port	generate mapping transform between windows

3.3.3.4 COLOR UTILITY FUNCTIONS

These functions support color handling, including determining pixel values for particular colors in the cone.vlt color space, color map file i/o, and compensation table.

The CONE.VLT represents the skin of the double hex-cone color model, where each pixel uses 3 bits to indicate color (1 bit each red, green, blue), and 5 bits for shading (4 bits gradient, 1 bit direction), as follows:

```
msb | 7   6   5 | 4 | 3   2   1   0 | lsb
    +---+---+---+---+---+---+---+---+
    | r   g   b |b/w|    gradient   |
    +---+---+---+---+---+---+---+---+
```

When the gradient half-byte is 0, the full primary (rgb) is visible. As the gradient increments to 15, the primary is shaded towards black or white depending on the b/w bit. This allows the gradient to be logically treated as an overlay and support mixture of anti-aliased text and images.

Figure 3-23 Color Utility Procedures

getprime	get pixel value for a primary color
getrange	get pixel range between two primary colors
getcolor	get pixel value between two primary colors
colormenu	offer a color selection menu to user
conevlt	generate a standard "cone" color table
loadvltf	load a color table from a file
savevltf	save a color table to a file
makctbl	make a color correction compensation table

3.3.3.5 ANTI-ALIASED FONT SUPPORT FUNCTIONS

The font support functions simplify the interface to the high quality fonts on the yoda display by allowing multiple fonts to be open at a time. The program simply requests a font by name and all the low level initialization and interfacing are taken care of.

Figure 3-24 Anti-Aliased Font Support Procedures

fontinit	initialize high level font support
fontselect	select and open font to use
fontclose	close a font
fontmenu	offer a font selection menu to user
ftodbl	convert from float to fixed double
dbltof	convert from fixed point to float

3.3.4 INPUT SUBSYSTEM

The locate functions provide a high level interface to locator (pointing) input devices and cursor. The main function "inp_locate" waits for the user to perform some action, and returns an action code. It reads the device, moves the cursor, checks the keyboard for any input there, and returns an action code, the locator device coordinates, and a button value depending on the action code. Programs using these functions instead of lower level ones will retain a level of device independence, because these functions are sensitive to the configuration of the workstation, using DOS environment variables.

The keyboard functions provide a direct interface with the IBM PC keyboard (through the ROM BIOS), rather than standard C getchar, etc. This allows non-ascii input and special keys to be read, such as the function keys. The function "keyarrows" allows the keyboard to simulate the mouse.

The mouse functions simplify the interface to the equivalent Mouse Systems, Inc. optical mouse interface library. The "mouse" function is smarter, remembering the button state.

Figure 3-25 Locate Actions

MOVED	user simply moved the cursor, no button change (button returned as UNDEFINED (-1))
OUTSIDE	like MOVED, except cursor went outside locate boundary (see set_locbounds)
PRESSED	a locator button was pressed, and its number is returned in button
RELEASED	a locator button was released, and its number is returned in button
ASCIIKEY	an printable ascii key was pressed on the keyboard, returned in button
FTNKEY	a function key was pressed on the keyboard, returned in button (see below)
ABORT	the abort key was pressed (esc, break or ctrl-c)

Figure 3-26 Input Library Functions

inp_locate	signal user events, track cursor
inp_dblclick	look for double click on mouse
inp_device	get input= device configuration from DOS
ask_locposn	get current locate position
set_locposn	set current locate position coordinate
set_locdevice	set current locate input device
set_locbounds	set current cursor boundary limits
set_locspeed	set device speed sensitivity
set_cursor	set cursor pattern from a bitmap
mak_cursor	make a standard cursor pattern

plus keyboard and mouse interface functions

3.3.5 MEMORY AND DATABASE SUBSYSTEMS

Numerous memory management and database management procedures were implemented to support (a) string manipulation, (b) large buffer and file io, (c) linked list and property list handling, (d) offscreen pixel buffer allocation and management, (e) rectangular border control, (f) overlapped layering and tiles, (g) raster image processing and color lookup table management, and (h) raster image database management.

CONCLUSION

The Spatial Contexts system demonstrates the integration of personal workspaces into computer-based environments. Key characteristics of personal design workspaces are identified, and a set of important computer metaphors that can support these requirements are described. A sequence of scenarios and detailed functional definition of objects in the system illustrate how these metaphors can be integrated into a comprehensive user interface environment. The system architecture specifies how a demonstration of this system has been designed and implemented on a personal workstation.

Programmers can expand the system by adding new objects, object properties and methods. However, the current software program is not a complete implementation of the ideas defined in this thesis. Furthermore, the "C" language has no built-in support for objects and limitations in software tools impacted the system development. In an object-oriented language and an interactive programming environment, the implementation of these specifications could be improved.

Future work on these ideas obviously includes completing the implementation and testing the effectiveness of the concepts with real people involved in real design projects.

BIBLIOGRAPHY

Subject Categories:

- Personal Workspaces
- Design Methodologies
- Computer Metaphors
- Object-Oriented Programming

PERSONAL WORKSPACES

Ned Block, Imagery, MIT Press, 1981.

Ronald J. Brachman, "What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks", IEEE Computer, 16:10, October, 1983.

Ronald Brachman, Richard Fikes, Hector Levesque, "Krypton: A Functional Approach to Knowledge Representation", IEEE Computer, 16:10, October, 1983.

Samuel Fillenbaum, Amnon Rapoport, Structures in the Subjective Lexicon, Academic Press, 1971.

Howard Gardner, Frames of Mind: The Theory of Multiple Intelligences, Basic Books, 1983.

Charles Hampden-Turner, Maps of the Mind: Charts and Concepts of the Mind and its Labyrinths, Collier Books, 1981.

John Haugeland, ed, Mind Design: Philosophy, Psychology, Artificial Intelligence, MIT Press, 1981.

Douglas R, Hofstadter, ed, The Mind's I: Fantasies and Reflections on Self and Soul, Bantom Books, 1981.

Paul Kellam, "The Future of Computing: Personal with a Capital P", Personal Computing Magazine, May, 1983.

Thomas Malone, "How Do People Organize Their Desks? Implications for the Design of Office Information Systems", ACM Transactions on Office Information Systems, Vol.1, No.1, January, 1983.

George Miller, P.N. Johnson-Laird, Language and Perception, Belknap Press, 1976.

Seymour Papert, Mindstorms: Children, Computers, and Powerful Ideas, Basic Books, 1980.

Lenhart Schubert, Mary Papalaskaris, Jay Taugher, Determining Type, Part, Color, and Time Relationships, IEEE Computer, 16:10, October, 1983.

Robert Sommer, Personal Space: The Behavioral Basis of Design, Prentice-Hall Inc, 1969.

M. Wertheimer, "Laws of Organization in Perceptual Forms", in Ellis, W.D., trans, A Source Book of Gestalt Psychology, Routledge & Kegan Paul, London, 1938.

William A. Woods, "What's Important About Knowledge Representation?", IEEE Computer, 16:10, October, 1983.

DESIGN METHODOLOGIES

Rudolf Arnheim, The Power of the Center: A Study of Composition in the Visual Arts, Univ of California Press, Berkeley, CA, 1982.

Gregg Berryman, Notes on Graphic Design and Visual Communication, experimental edition, William Kauffmann Inc, 1979.

Alexander Christopher, Notes on a Synthesis of Form,

E.W. Dijkstra, "Go To Statements Considered Harmful", CACM 25th Anniversary Issue, v.26, n.1, January, 1983, p.74. Also see John Rice, "The Go To Statement Reconsidered", and Dijkstra's reply on the same page.

Donis Dondis, A Primer of Visual Literacy, MIT Press, 1973.

Charles Eastman, "Explorations of the Cognitive Processes in Design", CMU Technical Report, February, 1968.

Karl Gerstner, Designing Programmes, Arthur Niggli Ltd, 1964.

Mark Gross, "A Laboratory for Exploring Design Constraints", MIT Dept of Architecture, Design Theory and Methods Group memo, May, 1985, to appear in AISB Conference on AI and Education.

Armin Hoffmann, Graphic Design Manual: Principles and Practice, Reinhold Publ, 1965.

Allen Hurburt, The Grid, A Modular System for the Design and Production of Newspapers, Magazines, and Books, Van Nostrand Reinhold Co, 1978.

Wassily Kandinsky, Point and Line to Plane, Dover

Publications, 1979 (original 1926).

Paul Klee, Pedagogical Sketchbook, Frederick Praeger, 1953.

Peter Molzberger, "Aesthetics and Programming", Human Factors in Computing Systems, SIGCH '83 Proceedings, 1983.

Jack Mostow, "Toward Better Models of the Design Process", The AI Magazine, Spring, 1985.

Alan Newell, Herbert Simon, Human Problem Solving, Prentice-Hall Inc, 1972.

Kimon Nicolaides, The Natural Way to Draw, Houghton Mifflin Co, 1941.

Herbert Simon, The Sciences of the Artificial, Second Edition, MIT Press, 1981.

Jan White, Editing by Design, A Guide to Effective Word and Picture Communication for Editors and Designers, R.R. Bowker Co., 1982.

Hans Wingler, The Bauhaus, MIT Press, 1978.

Niklaus Wirth, "Data Structures and Algorithms", Scientific American, Vol 251, No.3, September, 1984.

Niklaus Wirth, "Program Development by Stepwise Refinement", Communications of the ACM, vol.14, no.4, April, 1974.

COMPUTER METAPHORS

L. Blomberg, et al, "A New Approach to Text and Image Processing", IEEE Computer Graphics and Applications, vol.4, #7, July, 1984.

Richard Bolt, "'Put-That-There': Voice and Gesture at the Graphics Interface", SIGGRAPH '80, ACM Computer Graphics, vol.14, no.3, July, 1980.

Richard Bolt, "Gaze-Orchestrated Dynamic Windows", SIGGRAPH '81, ACM Computer Graphics, vol.15, no.3, August, 1981.

Richard Bolt, The Human Interface: Where People and Computers Meet, Lifetime Learning Publications, 1984.

Alan Borning, "The Programming Language Aspects of ThingLab", ACM Transactions on Programming Languages and Systems, vol.3, #4, October, 1981.

William Donelson, "Spacial Management of Information", SIGGRAPH '78, ACM Computer Graphics, vol.12, no.3, August, 1978.

"Filevision User Manual," Telos Software Products, Santa Monica, CA.

James Foley, Andreas VanDam, Fundamental of Interactive Computer Graphics, Addison-Wesley, 1982.

Don Hatfield, "A 'Direct Manipulation' Approach to Text Processing", draft, IBM Cambridge Scientific Center, 1984.

Paul Heckel, The Elements of Friendly Software Design, Warner Books, 1984.

Christopher Herot, et al, "A Prototype Spacial Data Management System", SIGGRAPH '80, ACM Computer Graphics,

vol.14, no.3, July, 1980.

Alan Kay, "Microelectronics and the Personal Computer", Scientific American, 1977.

Myron Krueger, Artificial Reality, Addison-Wesley, 1983.

D.S. Lipkie, et al, "Star Graphics: An Object-Oriented Implementation", SIGGRAPH '82, Computer Graphics, vol.16, #3, July, 1982.

David McKeown, Jr, "Graphical Tools for Interactive Image Interpretation", SIGGRAPH '82, ACM Computer Graphics, vol.16, no.3, July, 1982.

William Newman, Robert Sproull, Principles of Interactive Computer Graphics, 2nd Edition, McGraw-Hill, 1979.

Craig Reynolds, "Computer Animation with Scripts and Actors", SIGGRAPH '82, ACM Computer Graphics, vol.16, no.3, July, 1982.

Robertson, McCracken, Newell, "The ZOG Approach to Man-Machine Communication", CMU Technical Report, October, 1979.

Ben Shneiderman, "Direct Manipulation: A Step Beyond Programming Languages", IEEE Computer, August, 1983.

David C. Smith, et al, "Designing the Star User Interface", BYTE, vol.7, no.2, April, 1982.

Ivan Sutherland, "SKETCHPAD: A Man-Machine Graphical Communication System", SJCC 1963, Spartan Books, Baltimore, MD. (MIT Lincoln Lab Technical Report #296, May, 1965).

OBJECT ORIENTED PROGRAMMING

Elizabeth Allen, "YAPS: A Production Rule System Meets Objects", AAAI, 1983, 5-7.

Daniel Carnese, "Multiple Inheritance in Contemporary Programming Languages," Master's Thesis, Dept EE and CS, MIT, August, 1984.

Brad J. Cox, "Message/Object Programming: An Evolutionary Change in Programming Technology", IEEE Software, vol.1, no.1, January, 1984.

O.J. Dahl, "Simula 67 Common Base Language", Publication S-22, Norwegian Computing Center, Oslo, 1968.

Adele Goldberg, et al, Smalltalk-80: The Language and Its Implementation, Addison-Wesley Publ Co, 1983.

Carl Hewitt, "Viewing Control Structures as Patterns of Passing Messages", MIT technical report, no.410, December, 1976.

Alan Kay, "Computer Software", Scientific American, Vol 251, No.3, September, 1984.

Lamar Ledbetter, Brad Cox, "Software-ICs: A Plan for Building Reuseable Software Components", Byte, June, 1985.

J. Shoch, "An Overview of the Programming Language Smalltalk-72", SIGPLAN Notices, ACM, 14(9), September, 1979.

Paul Sholtz, et al, "YODA 0.2 Programmers Guide", internal document, IBM Yorktown Heights, February, 1985.

David C. Smith, Pygmalion: A Computer Program to Model and Simulate Creative Thought, Birkhauser Verlag Basel, 1977.

Larry Tesler, "The Smalltalk Environment", BYTE, vol.8, no.8, August, 1981.

Greg Williams, "The Lisa Computer System", BYTE, February, 1983.