

VIEWPOINT  
TOWARD A COMPUTER FOR VISUAL THINKERS

A DISSERTATION  
SUBMITTED TO THE COMMITTEE ON GRADUATE STUDIES  
OF STANFORD UNIVERSITY  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF  
DOCTOR OF PHILOSOPHY  
IN  
COMPUTERS AND GRAPHIC DESIGN

By  
Scott Edward Kim

## Updates, August 3, 2013

See the video demo of Viewpoint at: <http://youtu.be/9G0r7JL3xl8>

Check my web site for other updates: [scottkim.com](http://scottkim.com)

Twenty five years after it was written, the ideas in this dissertation remain provocative and ripe for followup. My goal was to do fundamental theoretical research in interaction design and the nature of programming — fields that remain bodies of practice with very little theoretical scrutiny.

I originally wanted to create a visual programming language by applying graphic user interface design to programming. In the end I laid the foundation for a visual programming language, but didn't get to programming itself. Visual programming remains an interesting unachieved goal. My dissertation stalled for lack of an application domain; I now think game design (my profession) would be an ideal domain.

Many people reading my dissertation think that bitmap parsing is the point. Parsing high-level structure from low-level data is definitely on the rise in our digitized world — consider face recognition, hashtags and QR codes. But my main point is that we should align the visual representation that the user sees and the data structures that the program “sees”, and that this can be accomplished either by bringing the mountain to Mohammed (bitmap parsing), bringing Mohammed to the mountain (conventional programming), or something inbetween (the interesting practical compromise).

Other projects that build on my work:

Editing scanned document images using simple interpretations, by Steven Bagley. <http://www.google.com/patents/US5734761>

Editing Images of Text, by Steven Bagley and Gary E. Kopec  
<ftp://ftp.cfar.umd.edu/pub/documents/contrib/oldpapers/Bagley93.ps.gz>

Triggers: Guiding Automation with Pixels to Achieve Data Access, by Richard Potter. Chapter (page 361) in the book Watch what I DO: Programming by Demonstration, edited by Allen Cypher and Daniel Conrad Talbert. [books.google.com/books?isbn=0262032139](http://books.google.com/books?isbn=0262032139)

Thanks to my long-time colleagues Fred Lakin and Warren Robinett, and honorary VizDin member Henry Lieberman. Our conversations over the years helped launch these ideas.

*Scott Kim*

## Followup research challenges

**Text editors deconstructed.** Analyze the fundamental elements that make up the model of text in modern text editors, such as the insertion cursor, word wrap, invisible characters, and character styles. What are the strengths and weaknesses in our commonly accepted modes, judged by interaction design criteria? E.g. invisible structure can be confusing. What are radically different solutions to these problems? E.g. Jef Raskin (grandfather of the Macintosh), proposed positioning the insertion cursor by search, not by spatial position, and showed that this is much faster and simpler than positioning the cursor by mouse or trackpad. In other words, question the status quo in text editors.

**Pixel-oriented text editor.** Build a text editor / page layout program that uses a bitmapped version of a document as its primary data structure. What can such a program do naturally that would be difficult or impossible in Microsoft Word or InDesign?

**Computers for visual thinkers.** If the first computers had been designed by visual thinkers instead of by numerical thinkers, how would programming look today? What if they had been designed by musicians? Writers? Architects?

**Visual Lisp.** Design a visual programming language with the reflexive power of Lisp — instead of a program being able to manipulate its own list structure, it can manipulate its own graphical structure. Fred Lakin has already done this ([pgc.com](http://pgc.com)); take it further.

**TPU.** Design the architecture for a text-based CPU, in which the primary interpretation of bytes is as characters, and the primary CPU operations are text string operations.

**PPU.** Design the architecture for a pixel-based CPU, in which the primary interpretation of bytes is as pixel arrays, and the primary CPU operations are graphic operations. GPUs already do much of this; take it further to include ultraparallel bitmap parsing operations.

**Visual programming.** Redesign a modern programming environment to be more graphical. In particular, make all the hidden structure visible, such as the contents of the data structures, and the syntax of the language. Scratch already does much of this ([scratch.mit.edu](http://scratch.mit.edu)); it can be taken much further and applied to conventional programming languages. See Bret Victor ([worrydream.com](http://worrydream.com)) for some promising starts in this direction.

*Scott Kim*

**TRADEMARK NOTICES**

**Apple** are trademarks of Apple Computer  
**Macintosh** is a trademark licensed to Apple Computer  
**MacPaint, MacWrite, MacDraw** are trademarks of Claris, Inc.  
**LEAP** is a trademark of Information Appliance, Inc.

**© Copyright 1988**  
**by**  
**Scott Kim**



**I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.**

---

**Donald Knuth, Computer Science (principal advisor)**

**I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.**

---

**William Verplank, Mechanical Engineering (lecturer)**

**I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.**

---

**William Paisley, Communications (formerly)**

**I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.**

---

**Charles Bigelow, Art & Computer Science**

**Approved for the University Committee on Graduate Studies**

---

**Dean of Graduate Studies**



## Preface



In the old days, computers worked with words and numbers. Today, computers such as the Apple Macintosh work with pictures as well as words—which makes them easier to understand, especially for visual thinkers.

What's next? Tactile computers with look and feel? Before we explore other sensory channels, we must realize that we haven't yet built a truly visual computer. Despite its graphic facade, the Macintosh is still built on a foundation of numbers and abstract data structures that defy visualization.

The goal of my research is to rethink computers from the ground up in terms of visual thinking. My approach draws on both computer science and graphic design, though not in their most usual forms. Within computer science, I focus more on user interface design than on algorithms. Within graphic design, I focus more on general theoretical issues than on particular images.

My research, chronicled in Chapters 1 and 2, began with the desire to build better tools for graphic designers and ended with the building of a pixel-based text and graphics editor called "Viewpoint". Chapter 3 demonstrates Viewpoint in detail. A videotape of the demonstration is also available.\* Chapters 4 and 5 analyze design techniques and theory, including a new user interface design principle called "visibility". Chapters 6 and 7 summarize conclusions.

You will find that I often write in symmetrical patterns. The idea is to take a familiar relation between two ideas and see what happens when the relation is inverted. Either the analogy will carry through, establishing a pattern, or it will break down, revealing a difference. For instance, inverting computer-aided design produces design-aided computation. The principle of visibility emerged as I pondered symmetries between the user and the processor with respect to the screen. Finally, adopting pixels as the basic representation of all information inverts the usual relation between structured and unstructured graphics.

Since this book is about visual thinking, I have included many pictures. The sketches down the side of each page typify the diagrams I scribble in the margins of books to help me understand ideas. Please feel free to add your own.

\* For information, write me care of the Stanford Computer Science Department, Stanford University, Stanford CA 94305.

## Acknowledgements

Dissertation committee	Donald Knuth Charles Bigelow William Verplank William Paisley	Thank you Don for taking me under your wing and letting me fly. Your care is appreciated. Thanks all for a safe trip.
Stanford support	Jock Mackinlay David Thornburg Robert McKim Matt Kahn	Jock enthusiastically criticized. David kept school fun. Bob taught visual thinking. Matt taught graphic design.
Conversations	Henry Lieberman Fanya Montalvo David Levy Suzanne West	Thank you for your challenging ideas. Best wishes with your future visual adventures.
Visual Diner's Club	Fred Lakin Warren Robinett	For five years, a great mutual support group.
Xerox PARC	Leo Guibas John Warnock Maureen Stone Michael Plass	Leo invited me to PARC. John and Maureen sponsored my stay. Michael helped with programming.
Visual presentation ideas	Gayle Curtis Cleo Huggins	Pleasant evenings brainstorming video ideas.
Video production	Mark Chow Theron Thompson	Not only excellent work, I got to collaborate fully.
Theory	Jef Raskin Jim Winter Information Appliance	IAI has been my home away from Stanford since 1983. Simplicity lives!
Copy editing	Clare Keaveney	Thanks for coming through.
Typesetting assistance	Stuart Klein Dan Mills	Stuart offered me a hard disk drive. Dan got Lucida.
Cheering section	Robin Samelson Addison Housemates Paul Trachtman Lester & Pearl Kim	Thank you for always being there for me. I couldn't have done it without the support of friends and family.

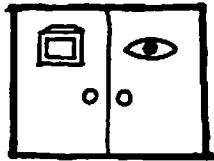


# Contents

Preface v

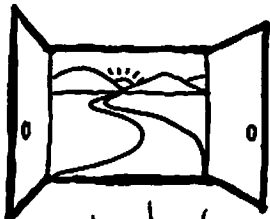
Acknowledgements vi

Table of Contents vii



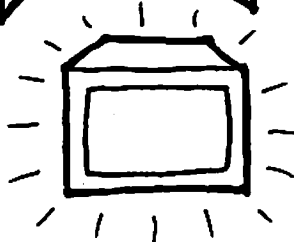
## 1 Introduction 1

The changing face of computers  
Graphic design enters the picture  
Rethinking computers & graphic design  
Viewpoint: a thought experiment  
On being interdisciplinary



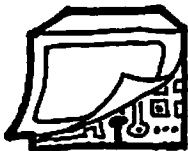
## 2 History 11

Visual thinking / Metafont  
Visual programming / The AI Factory  
Graphics editing / Pixels  
Viewpoint / Timeline



## 3 Demonstration 27

About the programming environment & illustrations  
Drawing / Selecting / Copying  
Puffbox / Typing / The visual boot  
What happens if...?



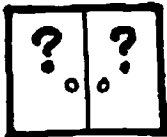
## 4 Implementation 83

Purpose / Behavior / Objects & actions / Encoding  
Cursor / Key highlights and key triggers  
Selection / Puffbox / Font / Text  
Ink color / Interlock / Grid / Inner workings



## 5 Theory 89

The need for formal definitions  
A model of interactive systems  
Visibility / Modeling Viewpoint  
Verifying visibility



## 6 Opportunities 111

Extending Viewpoint / Pixel algorithms and hardware  
Editor-based systems / Computers and graphic design  
Visual programming / User interface design  
Visual thinking

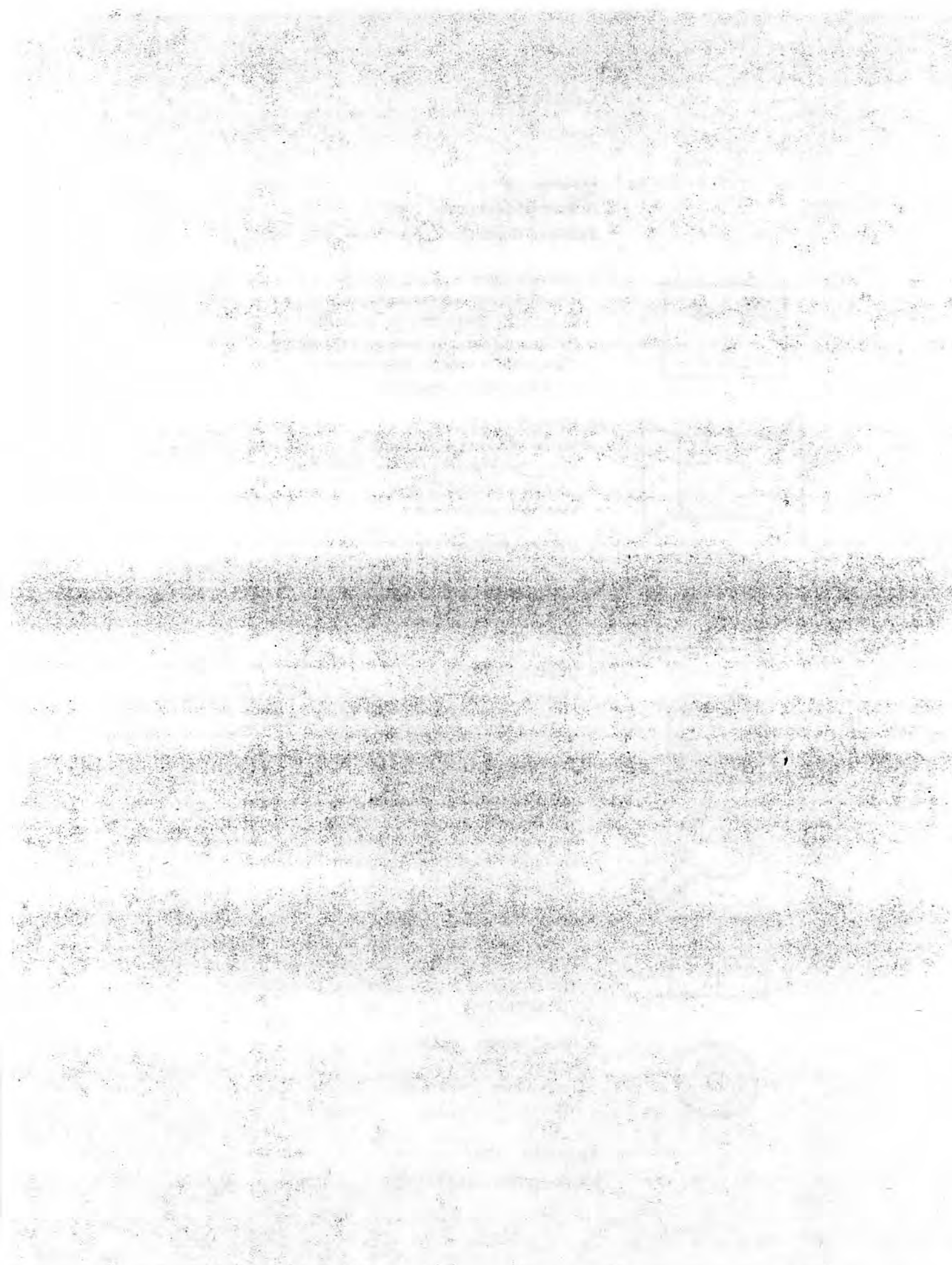


## 7 Conclusions 119

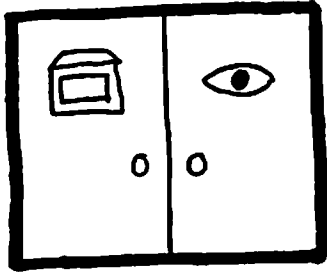
Achievements  
Techniques  
Insights

Appendix 123

Bibliography 127



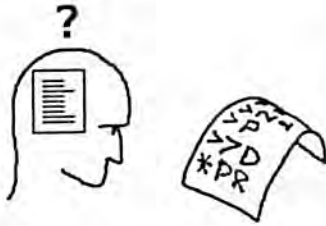
# 1 Introduction



This chapter introduces the main themes of **Viewpoint: computers and graphic design.**

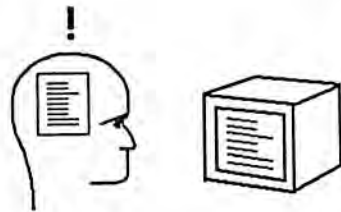
- The changing face of computers
- Graphic design enters the picture
- Integrating computers & graphic design
- Viewpoint: a thought experiment
- On interdisciplinary work

## The changing face of computers



In the old days of computers, the user played detective. Here, the user has just punched 7D <return> to delete seven characters at the beginning of a line of text. Since the resulting text is not directly visible, the user must either imagine the result based on remembered clues or type an additional command to print out the new line.

This style of interaction is called a "command line" interface. Command line interfaces place a heavy burden on the user's memory and are consequently hard to learn and remember. Such interfaces live on today in operating systems such as UNIX and MS-DOS.



Nowadays, computers are becoming more graphic. Bitmapped screens allow users to see live pictures of the changing result. Here, the user has typed <control>-D repeatedly to delete characters at the beginning of a line of text. Since the result of each action shows immediately on the screen, the user doesn't have to plan ahead.

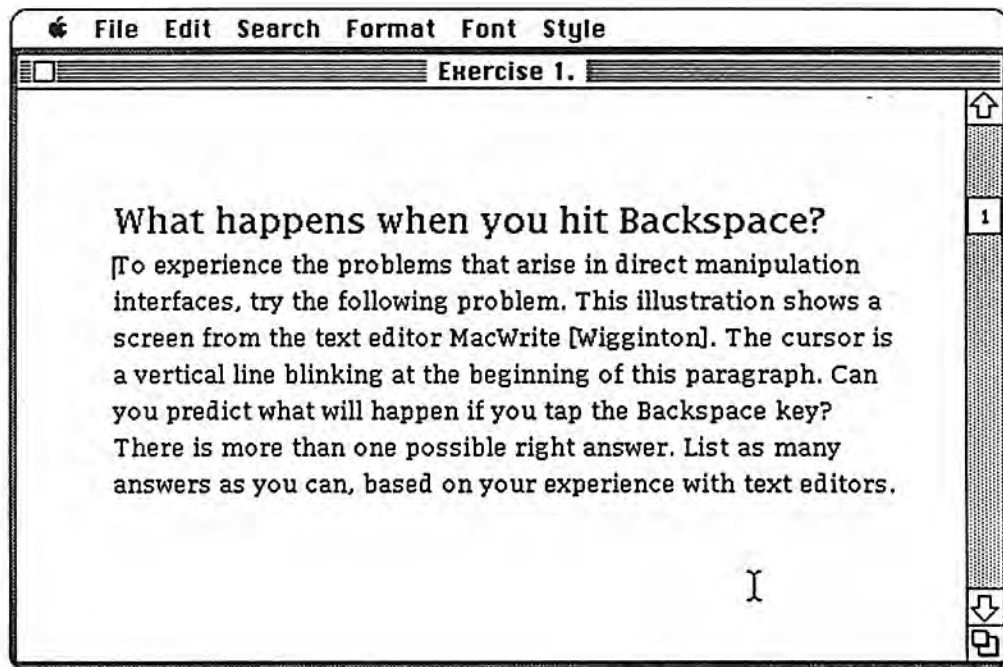
This style of interaction is called "direct manipulation" [Shneiderman]. Direct manipulation interfaces lighten the load on the user's memory and are consequently easy to learn and remember. Examples of such interfaces include video games, spreadsheets, and the desktop metaphor (first introduced in Xerox's Star office system).



The movement toward direct manipulation interfaces results from the push of technological improvements and the pull of user demands. Cheaper memory and faster processors make it practical to devote more computer power to the user interface. A typical bitmapped display of today may use as much storage as an entire mainframe computer of twenty five years ago. Wider and less specialized audiences demand systems that are easier to use. A typical user of today requires programs that can be learned in hours, not weeks.

Direct manipulation represents a fundamental change in the role of the interface. A computer with a bitmapped screen is not a paperless teletype any more than an automobile is a horseless carriage. The difference is that command line interfaces describe invisible objects while direct manipulation interfaces purport to show the objects themselves. David Canfield Smith, one of the creators of Xerox's Star system, the first system to use the desktop metaphor, writes that "a subtle thing happens when everything is visible: the display becomes reality."





In this example, the illusion of direct manipulation breaks down due to the presence of invisible information.

Perhaps Backspace will delete the Return at the end of the previous line. But where is the end of the previous line? There might be spaces or tabs after the question mark; we can't be sure. There might also be tabs embedded in the text after the cursor, which would cause gaps to appear when the text was wrapped.

Perhaps the left margin isn't where it appears to be. There might be spaces before the beginning of the current line—"To" might not be the first word on the line. Margin and tab settings are not visible unless we scroll back to the last ruler. But are rulers currently visible? We can't be sure without opening a menu to check ruler visibility mode.

In text editors with both overtype and insert modes, Backspace sometimes deletes the previous character and sometimes just moves the cursor. In such a system, there is no way to tell what will happen when you hit Backspace unless there is a visible mode indicator.

## Graphic design enters the picture

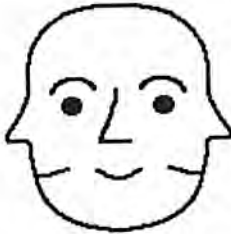
Graphic design is the composition of two-dimensional static images for the purpose of communication. Images may include text, diagrams, photographs, and illustrations. Applications include books, magazines, posters, stationery, packaging, and signs. Related fields include the design of three-dimensional products, exhibitions, architecture, video, and clothing.

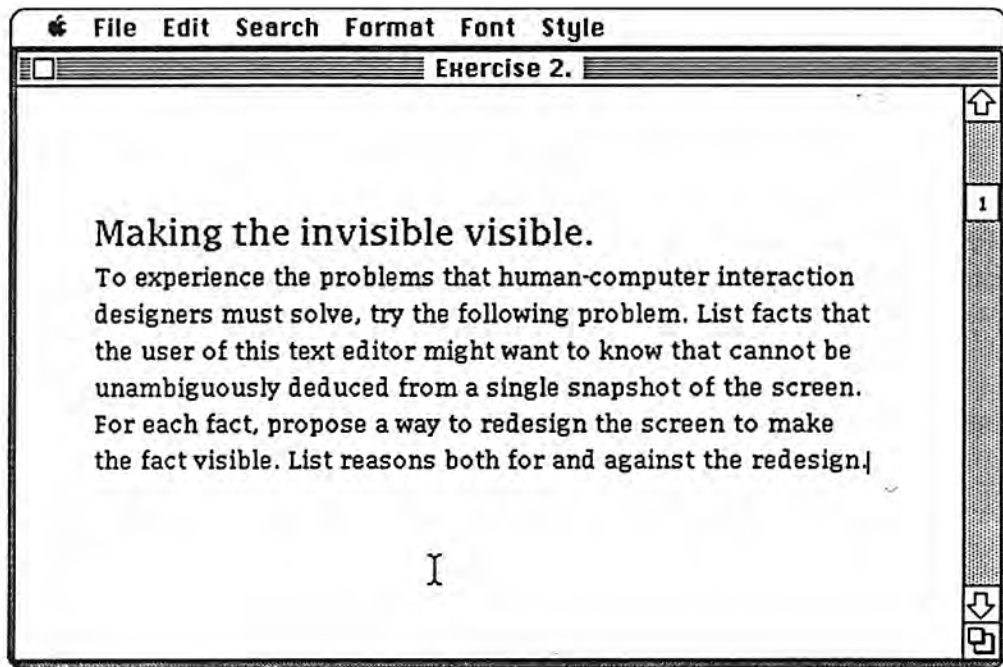
To build direct manipulation interfaces, software engineers must solve problems of visual communication. Since these problems are beyond the scope of conventional computer science, software developers are turning to graphic designers for the missing skills. Graphic design may be applied to computer systems at three levels: the outerface, the interface, and the innerface [Marcus 1983].

The outerface includes images, diagrams, and text produced with the aid of computers but presented through traditional media such as slides or print. The need for graphic design is obvious here, since the computer serves in place of traditional visual communication tools such as pen and ink, type, camera, and pasteup board. The topics remain the same; only the tools differ. The outerface challenges designers to uphold the qualities of previous technologies while taking full advantage of the unique capabilities of new tools.

The interface includes images, text, and animation presented by computer as part of online interactions between users and programs. The need for graphic design here is beginning to be recognized as software publishers begin to compete on the basis of "look and feel". The visual design problems are similar to those of traditional graphic design, but with the added dimensions of animation and interactivity. Typical interface design assignments include icons, cursors, screen layout, and animated transitions. The interface challenges designers to extend their skills to solve new types of problems.

The innerface includes diagrams, text, and displays used by programmers to build software. The need for graphic design here is just starting to be imagined as programmers become accustomed to graphic programming environments. The visual design problems here are related to those in areas outside of traditional graphic design, such as map making, instructional design, and engineering drawing. The innerface challenges designers and programmers to work together to understand the nature of visual representation.



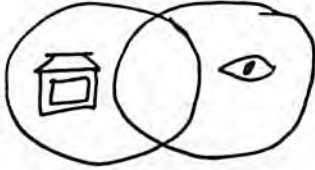


There are many ways to reveal hidden information.

We might display an *I* in one corner of the screen to indicate that we are in insert mode. Just because an indicator is visible, however, doesn't mean it will be seen. A more effective solution, when the number of modes is small, is to change the cursor shape. This puts the information where it is likely to be seen.

Many text editors display nonprinting characters such as tabs and carriage returns as explicit characters. But this means that the screen no longer matches the printed page. To resolve this dilemma, many editors display nonprinting characters only they are selected or when a special "show structure" mode is invoked.

Some information is best kept hidden. For instance, the user normally has no need to see the current value of the program counter. Furthermore, the need to reveal hidden information can sometimes be taken to mean that the system itself should be simplified. Today's text editors often eliminate otype mode in favor of insert mode in order to simplify the choices the user must make.



## Integrating computers & graphic design

Computer science and graphic design differ in topic. Computer science is about computers; graphic design is about graphics. Less obvious but more important is the difference in approach. To understand how computer scientists and graphic designers can work together, we must understand how their approaches differ.

Computer scientists approach computation with the objective question "Does it run?" To answer this question, computer scientists write programs. Since user interfaces to programs are considered secondary to programs themselves, computer scientists rarely test their programs on users.

Graphic designers approach graphics with the subjective question "Does it look right?" To answer this question, designers draw pictures. Since rules for generating good design are considered secondary to designs themselves, graphic designers rarely write down rules.

Approach belongs to the collective unconscious of a discipline. Only when disciplines clash do approaches emerge as arbitrary assumptions.

Designer: What can this machine do?

Programmer: Anything you want.

Designer: Marvelous! Let's see a page of text.

Programmer: Okay.

Designer: But that looks ugly.

Programmer: That's the font that came with the machine.

Designer: I guess that's what computers look like.

Programmer: No, you can change the font, too.

Designer: Okay, let's change it.

Programmer: You can, but it's too much work to bother.

Designer: You really should use better fonts.

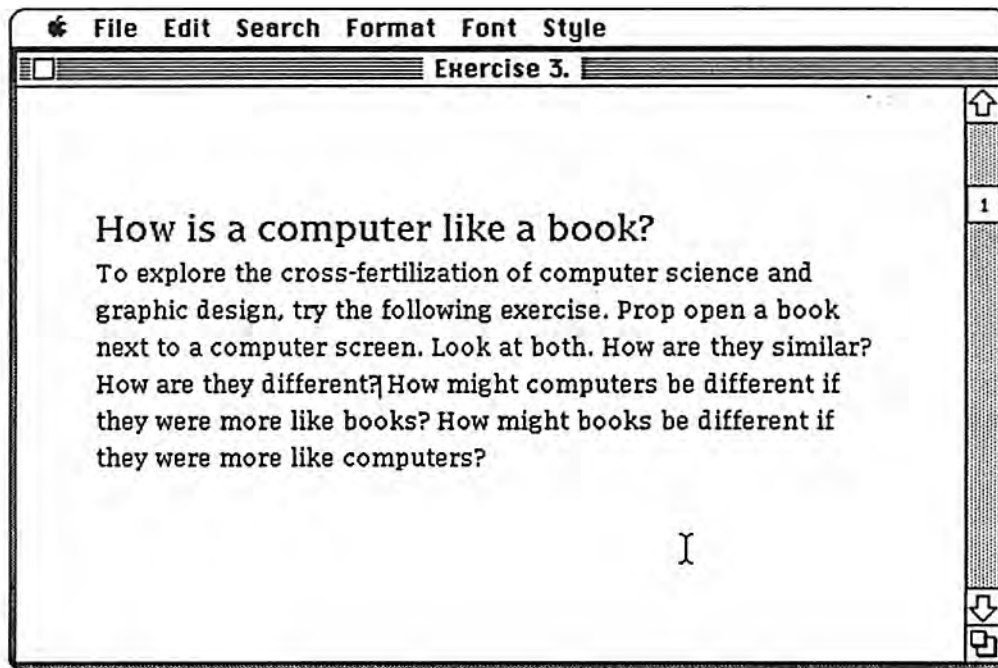
Programmer: What are the rules for designing good fonts?

Designer: Can't you see?

Programmer: Can't you tell me the rules?

Programmers tend to tolerate graphic design as a necessary evil, an art to be admired but never fully comprehended. On the other hand, designers tend to treat computers as immutable, a set of laws to be suffered but never challenged. Both attitudes spring from lack of understanding. To truly integrate computers and graphic design, one must appreciate the insights of both while accepting the blindnesses of neither.



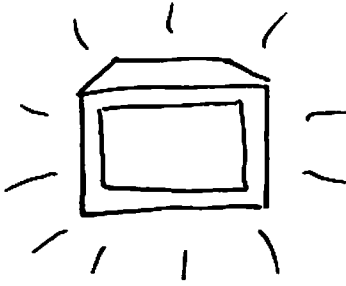


A brainstorming technique I often use is to draw analogies between computer science and graphic design. When I make an observation in one field, I wonder what the corresponding observation would be in the other field.

For example, programmers become so engrossed in the technical aspects of their work that they sometimes forget to make their programs usable. The analogous observation in graphic design is that designers can be so engrossed in the aesthetic aspects of their work that they forget to make their designs communicate.

The inverse of Computer-Aided Design (computers in the service of design) is Design-Aided Computation (design in the service of computers). Just as design can serve the three faces of computers, so computers might serve the three faces of design. What might they be?

When the analogy breaks down, we have discovered a hole in the pattern. Design-Aided Computation is far less common than Computer-Aided Design. Why?



## **Viewpoint: a thought experiment**

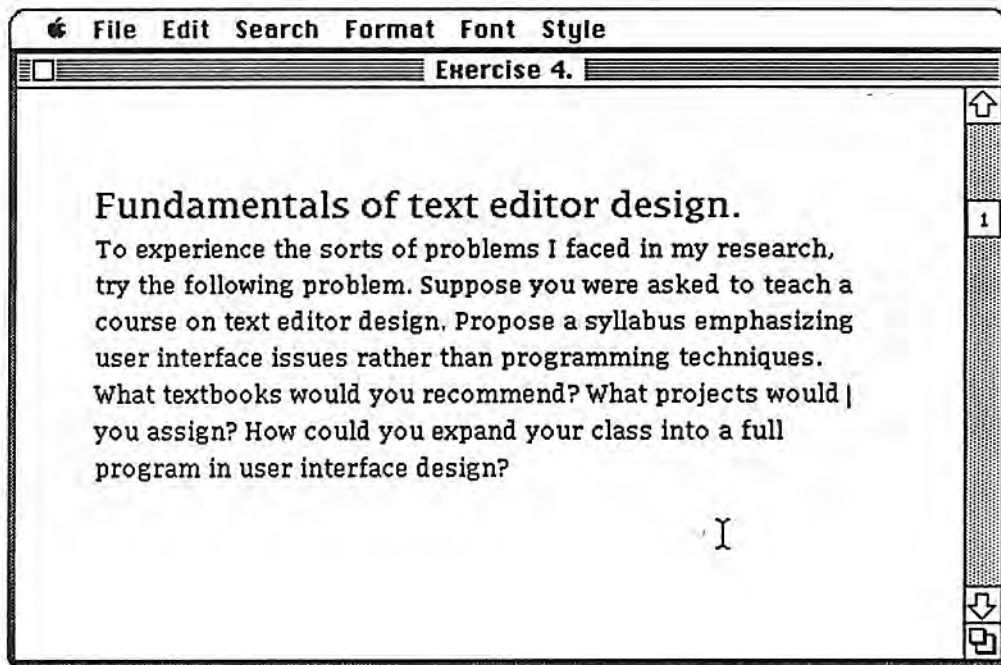
To study fundamental issues in the visual representation of information, I built Viewpoint, an experimental text and graphics editor in which the primary representation of information is pixels.

Viewpoint can be used to draw pictures, design fonts, and edit text. Unlike other painting systems, Viewpoint makes no distinction between text and graphics. The same group of pixels can be treated as either text or graphics, depending on which operation you perform.

The point of Viewpoint, however, is not to edit text and graphics or even to demonstrate better ways to edit text and graphics. As with a Turing machine, its purpose is to test basic theoretical ideas in their purest form. Viewpoint is best thought of as a formal experiment. It is not meant to be immediately practical or efficient.

I chose to pursue an interdisciplinary degree because the issues I wish to study do not fit within a single discipline. Computer science studies computers but ignores users. Human factors includes users but fails to recognize the screen as a primary experience. Graphic design treats images as primary experiences but tends to avoid philosophical issues. Linguistics is willing to discuss philosophical issues but balks at visual language.

Viewpoint is neither conventional computer science nor conventional graphic design, but rather a synthesis of both. If I were to name my area of research, I might call it human-computer communication, foundations of user interface design, philosophy of graphic design, or visual linguistics.



Twenty five years after the invention of text editors, there are still no books or courses on text editor design. The lack is especially striking given that most of the use of personal computers is text editing. Computer science students design compilers to learn principles of programming; why not design text editors to learn principles of user interfaces?

Articles do exist on the psychology of text editor use, on programming techniques for interactive systems, and on the design of complex document formatting systems. Apparently text editor design gains respectability only when cast in the image of psychology or programming.

The lack of research on text editor design shows that the field of human-computer interaction has yet to emerge as an independent discipline. We need to ask basic questions. Before we are entitled to ask whether tiled windows are better than overlapping windows, we must ask, "What is the problem that windows solve?" "What is the range of possible solutions?" and "What language shall we use to talk about the issues?"

## On interdisciplinary work

Curiously, the word *interdisciplinary* exists only as an adjective. There are *disciplines*, but no *interdisciplines*. It is as if interdisciplinary people must forever wander homelessly. Thinking further about the nature of interdisciplinary work, I realized that the word "interdisciplinary" has several shades of meanings.

**Castles.** Disciplines are private, walled kingdoms sitting on neighboring hills. Occasionally, bilingual messengers carry news from one kingdom to another. The walls were originally built to defend territories. Nowadays kingdoms grudgingly accept that they must coexist.

**Cracks.** The world of knowledge is cut up into categories. Categories bring a sense of order and stability to an otherwise chaotic world. Some people don't fit the categories, but instead fall between the cracks. For them we invent a new category: people who can't be categorized.

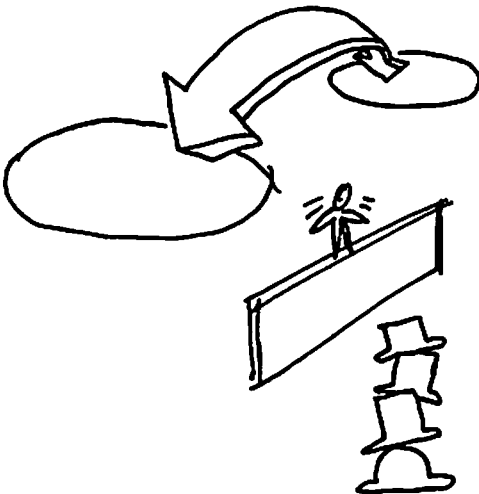
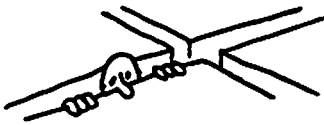
**Bridge-builder.** Disciplines are islands separated by the sea of ignorance. Interdisciplinary people build bridges between islands so that others may cross. Without such bridges, passage between islands is difficult. One day, perhaps, all islands will be connected.

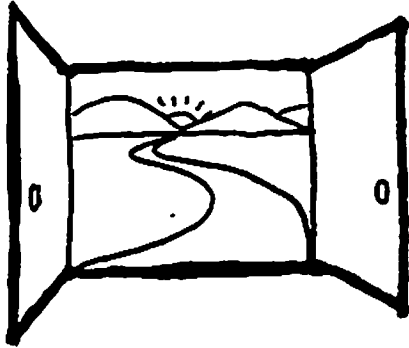
**Fence-sitter.** The boundaries between disciplines are marked by fences. Without such fences, we could never tell who owned what territory. Each person must decide where he or she belongs. Interdisciplinary people sit on the fence, never deciding which side to commit to.

**Hats.** Throughout the day, we all play many different roles: parent, child, teacher, student, worker, friend, creator, performer, viewer. Each role comes with its own hat. Interdisciplinary people wear several hats at once. Too many hats make balancing difficult.

**Viewpoint.** I named my project "Viewpoint" as a reminder of the subjective nature of perception. There is only one world, but many ways to view it. Different frames lead to different interpretations. Interdisciplinary people are able to switch points of view.

Differing viewpoints exist not only between disciplines but within disciplines. In computer science, a digital circuit designer views programming as a way of telling a computer what to do, whereas a programmer views digital circuitry as a way of implementing an algorithm. In graphic design, a production artist views design concept as way of figuring out what to do with tools, whereas a graphic designer views tools and techniques as ways of implementing a design.





## 2 History

This chapter recounts the events that led to Viewpoint—what I did and what I learned.

- Visual thinking
- Metafont
- Visual programming
- The AI Factory
- Graphics editing
- Pixels
- Viewpoint
- Timeline



## Visual thinking

When I first came to Stanford, the Visual Thinking course caught my eye [Stewart] [McKim]. The course was listed in engineering, but seemed to draw equally on art. When I stopped by to visit, the classroom was always filled with energetic drawings and models. Most of all, I liked that the course was about thinking itself—not just the understanding of thinking but the doing.

The Visual Thinking course started because students coming to Stanford didn't know how to draw—an essential skill for engineers. But this was no typical engineering drawing class. Rather than teach precise drafting techniques for recording finished ideas, Visual Thinking taught quick sketching techniques for brainstorming new ideas. The course challenged students to invent, refine, and build imaginative projects with the aid of prolific sketching. Projects tended toward the absurd, to encourage fresh thinking and avoid canned solutions.

A typical project was to build a drawing machine for five people to operate at once. The first step might be to brainstorm possibilities by sketching different types of drawing devices, sketching different ways for five people to interact, then combining each device with each interaction method. The next step might be to develop a particular idea further in tighter sketches. Students were asked to record all their sketches in an “idea log”. The point was not just to make a machine that worked but to learn a style of thinking.

Visual Thinking crystallized an idea I had always valued but for which I had never had a name. In mathematics, I enjoyed the visual and spatial aspects of geometry. But mathematicians tend to suppress the visual aspect of their craft in favor of working with symbols. In psychology and art, I enjoyed creating perceptual illusions. But most of the time, psychologists are too caught up in analysis to create new illusions, and artists are too caught up in creating images to analyze the cognitive aspects of their craft.

The course encouraged me to develop my own visual thinking skills. I studied graphic design and typography. I learned about typeface design and calligraphy. I used more pictures in my note-taking. I helped teach the Visual Thinking course. Gradually, visual thinking became the center of my work.

Visual thinking is the foundation of Viewpoint. Since the idea of visual thinking is not widely understood, I must explain what I mean.

Visual thinking is thinking with the aid of pictures, as opposed to thinking with the aid of numbers, words, sounds, or other vehicles of thought. Thinking includes inventing, refining, and realizing ideas. Pictures include images that are seen, drawn, or imagined. Thinking in pictures is most appropriate when the subject matter is pictorial, as in graphic design, spatial, as in maps, or complex, as in statistical charts.

Since words and numbers have visual notations, visual thinking cannot be completely separated from other types of thinking. Furthermore, visual thinking relies closely on muscular and spatial thinking. Visual thinking is not a tightly defined category but a loose collection of mental techniques drawn from the elements of visual perception: color, shape, orientation, texture, position, etc.

Any discussion of visual thinking must take into account the confusion around the very existence of visual thinking. "People who think in pictures often doubt that others do not. People who lack imagery, on the other hand, are skeptical that anyone has it." [Sommer]

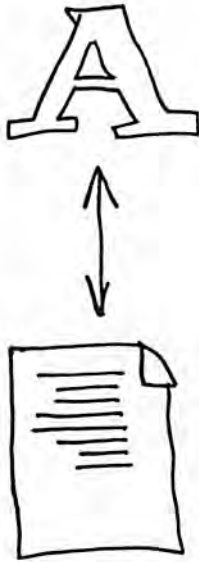
The confusion is largely due to education. Everyone with sight thinks visually. To walk, for instance, is to make visual judgments about the environment. But schools systematically suppress visual thinking in favor of verbal and numerical thinking.

Consider mathematics. In elementary school, math books are filled with pictures and activities. As you grow older, pictures become smaller, fewer, and less colorful. In scholarly journals, pictures all but disappear, the stigma being that if you can't say it in symbols you aren't a "real" mathematician.

Magazines and television teach us to read visual messages. But nowhere do we learn to write visual messages. Visual illiteracy leads to the belief that thinking happens only in words. The belief is trapped by language: Arguments for the existence of visual thinking must be stated in words in order to be taken seriously. Even the term "visual illiteracy" bows to the sovereignty of words.

In his book *Visual Thinking*, Rudolph Arnheim pinpoints the problem: a belief that "...the gathering of perceptual data is unskilled labor, indispensable but inferior." [Arnheim] For visual thinking to flourish, images must be accepted as full citizens in the world of thought.





## Metafont

My interest in computers and graphic design started with Metafont, Donald Knuth's programming language for typeface design [Knuth]. Typeface designers normally draw letters by hand. A typeface designer using Metafont creates letters by writing programs. Each program tells the computer how to construct a letter, much the way a recipe tells a cook how to prepare a dish.

By programming instead of drawing, a designer can describe a class of shapes in general, not just a single shape. For instance, a single Metafont program can describe bold, light, condensed, expanded, serif, and sans serif variations on a letter—and everything in between. For the first time in history a designer can express an entire typeface family in a single description.

I was intrigued by Metafont's blend of technical and aesthetic concerns. For several years, I worked to adapt Euler, Hermann Zapf's typeface for mathematics, to Metafont. Though Zapf designed Euler with Metafont in mind, it was not an idiomatic design. Zapf delivered Euler as a set of outline drawings—one set each for normal and bold weights—without specifying precisely how shapes should vary between weights.

I could have transcribed the Euler outlines without capturing variability. But that would have defeated the point of Metafont as I understood it. In order to restore meta-ness to Euler, I wrote procedures that modeled the structure of a flared pen stroke common to many Euler letters. Unfortunately, the actual letters did not fit my' analysis easily.

Furthermore, Metafont lacked facilities for taking accurate measurements off existing drawings. When Zapf came to visit, he wondered why my transcriptions didn't match his drawings more closely. As a type designer, his natural priority was that the image look right, not that the program run.

To transcribe Euler more accurately, I needed a way to measure drawings. Fortunately, Dave Siegel stepped in to create Metaface, a program that reads a digitizing tablet as input and writes Metafont programs as output [Siegel]. Dave, Carol Twombly, Dan Mills, and Cleo Huggins spent many months tracing letters into the computer. Metaface got the job done, but at the expense of meta-structure.

Most type designers who try Metafont find it too difficult to use [Southall]. The difficulties arise from the conflicting thinking styles of type designers and computer scientists.

**Interdisciplinary education.** Knuth knew that type design and programming are rare skills and so expected that the best work would come from collaborations. To encourage such work, I wrote a Metafont workbook specially for designers, organized around visual examples. In contrast, Knuth's *The Metafont Book* is organized around programming language syntax [Knuth]. A type design book for programmers would be similarly valuable.

**Designing for the medium.** Imitating existing designs in Metafont is useful but doesn't reveal its real strengths. The real challenge is to develop new designs that take specific advantage of the new medium. Metafont itself was molded by Knuth's development of the Computer Modern family of typefaces, with editorial assistance from major type designers. Other original Metafont fonts include Computer Modern Sans Serif by Richard Southall and Donald Knuth, Pandora by Neenie Billawala, and a partial Chinese font by John Hobby and Guoan Gu.

**User interface.** Letters exist to be seen and so must be seen to be judged. The most frequent complaint by type designers about Metafont is that they can't see graphic results immediately. Metaface by Dave Siegel gives Metafont a graphic user interface for capturing coordinates. Paragon by Lynn Ruggles, also built in response to Metafont, offers a graphic interface for editing and composing shapes. Imagen Corporation has developed an interactive graphic type design system with Metafont-like constraints.

**Multiple representations.** Letters have three types of structure: ductal (strokes), glyptal (outlines), and pictal (pixels) [Bigelow]. All three representations are important to designers at various times. Metafont, which includes only strokes and outlines, does not allow switching between representations. The difficulty is mainly philosophical, not technical. Stroke, outline, and pixel representations do not contain equivalent information. Hence transformations from one to another must lose information. Traditional computer science springs from mathematics, which focuses on information-preserving transformations. The question is, Which do you prefer: design flexibility or information preservation?

## Visual programming

For several years, I planned Viewpoint to be a more visual version of Metafont that handled multiple representations. As I considered programming the system, however, something kept bothering me. To build such a visual system, I would have to express my ideas in a conventional, verbal programming language. Why couldn't programming languages themselves be more visual?

Intrigued by the possibility of visual programming, I changed my goal from a type design system to a visual programming language. While visual programming would be of less help to typeface designers, it seemed to address more fundamental issues.

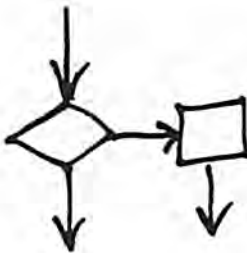
A visual programming language is a programming language in which programs are represented graphically. A system can fail to be a visual programming language in two ways: It can lack the full algorithmic power of a programming language, or it can lack the full visual expression of two-dimensional graphics. For instance, you can't program in MacPaint, and you can't understand a Pascal program from ten feet away.

The seeds of visual programming go back to the first interactive computer graphic system, Sketchpad, which included ways of graphically representing mathematical constraints between graphic objects. Since then, visual programming has blossomed into a whole field [Raeder].

In his taxonomy of visual programming, Brad Myers distinguishes visual programming, programming by example (building programs through visual interaction), and program visualization (visual representation of a conventional textual program) [Myers]. We may also distinguish languages supporting visual interaction (such as turtle graphics in Logo) and visual information processing languages for analyzing images [Chang].

The name "visual programming" suggests a programming language to which visuals have been added. Flowcharts are the most common visualization of programs, so visual programming to many people means fancy flowcharts. In fact, there are other approaches. Warren Robinett created the popular educational game Rocky's Boots by adding programmability to a graphic adventure game environment. Fred Lakin coined the term "executable graphics" to describe the approach of his text-graphic environment PAM: "start with the visual richness of objects provided by graphic design and then find ways to make them executable" [Lakin].

	Textual	Visual
Editing	EMACS	MacPaint
Programming	Pascal	?



For all its promise, visual programming raises more questions than it answers.

First, the notion of a programming language has never been well defined. Language designers are too close to implementation problems to propose general definitions. A definition that begins "a sequence of symbols that..." is useless for visual programming. I found only one taxonomy of programming languages that was not a territorial dispute. Theorists, on the other hand, are too far away from particular implementations to pay attention to visual representations. Formalisms filter out the features that distinguish graphics from text.

Next, we must realize that textual programming languages *are* visual programming languages. Programmers use indentation to visually organize programs. A programmer will fix a misindented statement even though it does not affect execution. That almost no programming languages use indentation to denote block structure reveals that computer science is blind to graphic design issues. One project that does notice the graphic nature of text is Aaron Marcus and Ron Baecker's work on the typography of computer programs [Marcus 1982].

Finally, we must realize that many factors must align for a convincing visual programming language to emerge. We can't just take Pascal and turn up the "visual" knob.

First, the visuals must be well designed. Since graphic design is not a respectable computer science research topic, visual programming languages remain aesthetically crude.

Next, the application must be appropriate. Visual programming languages are often judged by their ability to perform tasks better suited to textual programming, such as computing factorial. The best visual programming systems come from new application domains with strong visual metaphors, such as spreadsheets, video games, publishing, and user interface prototyping.

Visual programming is inconceivable without good input/output hardware. But our displays are still tiny and our input devices clumsy. Jaron Lanier's visual programming environment, tentatively titled "Grasp", is built around a new input device that registers three-dimensional hand and finger position [Lanier].

Finally, programming languages are sustained by programming environments. Lisp without a text editor that understands parentheses would be painful. We can expect more visual programming systems to show up as visual user environments become common.



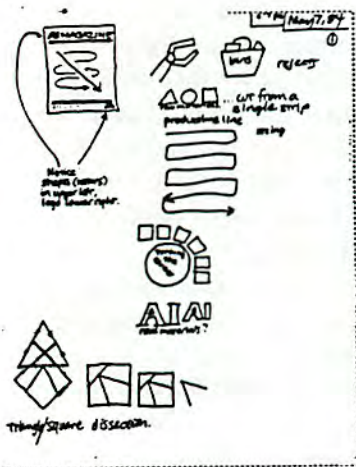
## The AI Factory

After several years of thinking about visual programming languages, I knew what I didn't want. But I didn't have a positive idea of what I did want. To determine my next step, I turned to graphic design. Here are sketches from the development of the "AI Factory", a cover illustration I designed for *AI Magazine* [Kim]. I studied these sketches to remind myself of the nature of visual thinking.

The sketches reminded me that design is a process that begins with chaos and moves towards order. Current computer-aided design systems, which force users to express their ideas in well-formed sentences, address only the later stages of the design process. In the earlier stages, fast feedback (sketching) is more important than precise description (programming) [Negroponte].

I tried imagining a computer-aided design system able to assist the design process in the AI Factory. I concluded that I would want to start with an unstructured painting system and gradually add structure. The key was the image of programs as annotated diagrams.

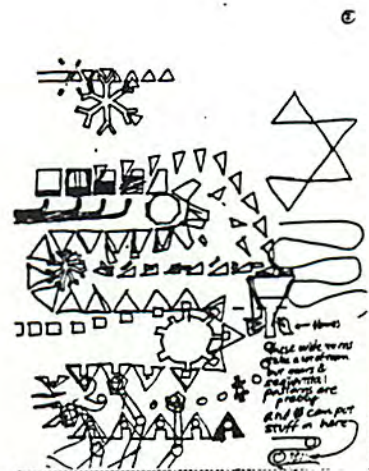
In many ways, the AI Factory is not a typical graphic design job. The images are deliberately mechanical, in keeping with the subject and the means of execution. Most images are far harder to describe in a program. Furthermore, different designers work differently. Nevertheless, the AI Factory captures a practice common to all graphic design: developing ideas by sketching.



Getting ideas. The theme, "AI in Industry", suggested an assembly line.

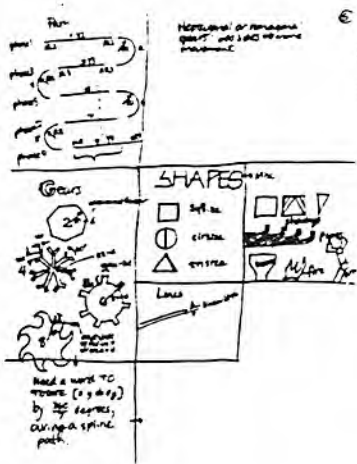
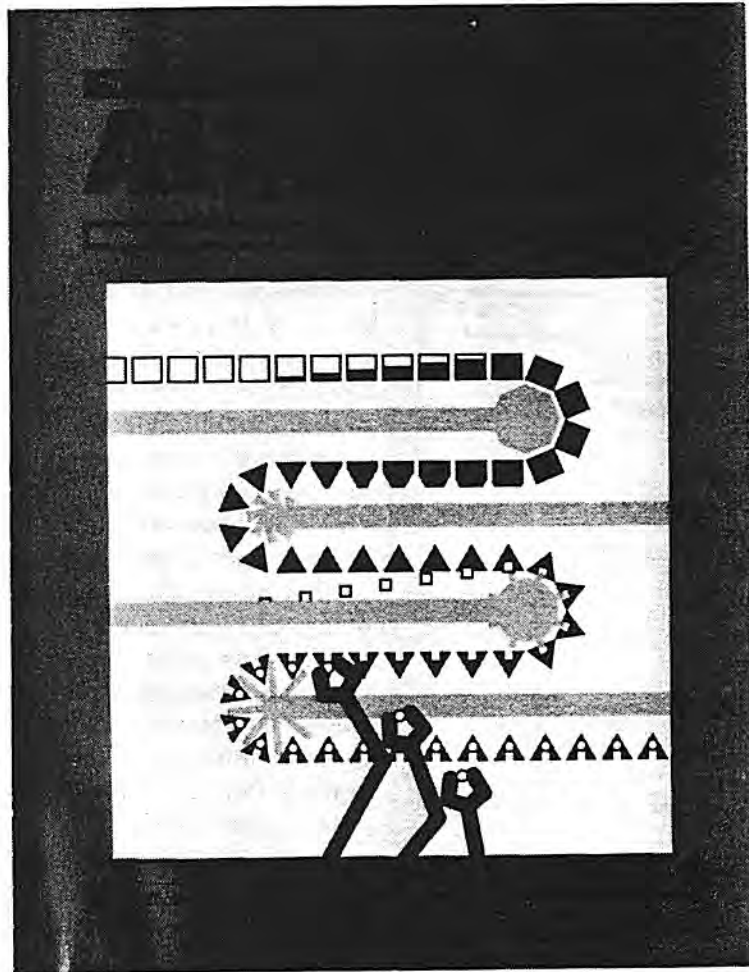


Trying alternatives. I usually work in pen so I can see all thoughts, including mistakes.

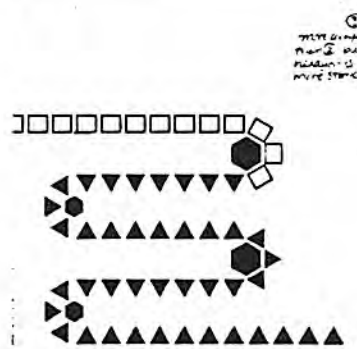


Refining ideas by sketching. From this sketch, I selected elements for the final design.

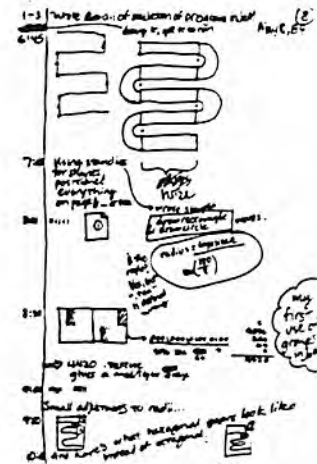
The final cover design,  
based on the AI logo  
and programmed in JaM.



Planning the program. This diagram is the Rosetta stone joining image and program.



Experimenting with variables. This draft showed me that a six-sided gear is too angular.



Recording my progress. A timeline allows me to reflect on how my thinking evolves.

## Graphics editing

The AI factory gave me the image of a unified programming environment built on a graphics editor, much the way current software environments such as the Symbolics Zeta-Lisp programming environment [Weinreb] or the Canon Cat text environment designed by Information Appliance [Alzofon] are built on text editors. Within this environment, I would be able to take a design all the way from rough sketches to finished programs.

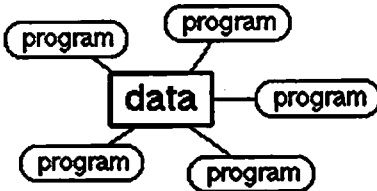
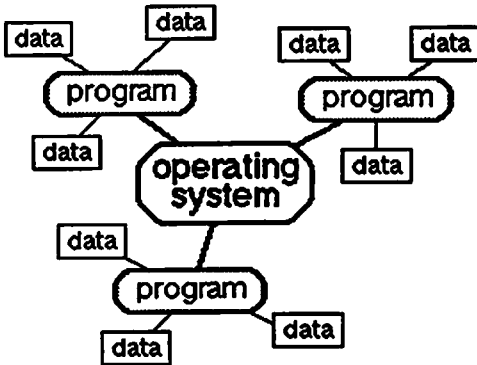
Older operating-system-based environments may have many different forms of data, each with its own editor. Even if programs are compatible, the effort of switching from one program to another interrupts the user's work flow. Users are forced to decide how to proceed before they decide what they want to do. Operating systems are a holdover from older, batch-processing systems.

The advantage of an editor-based environment over an older operating-system based environment is integration: All programs share a single form of data. Not only are programs automatically compatible, they can be used in any order at any time. Users can concentrate on what they want to do, rather than how they have to do it. Editor-based systems follow the noun-verb principle of interaction design [David Smith].

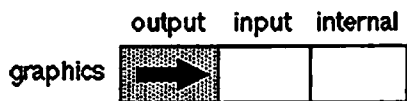
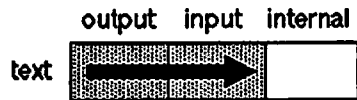
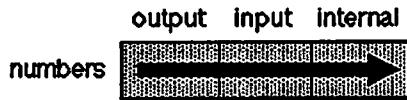
Editor-based environments do more than just make editing convenient—they change the focus of computers from programming to editing. In fact, the most common use of personal computers today is text editing, not computation. The word "computer" has become obsolete.

I now think of editing as part of a spectrum of actions. Most direct is direct action: One action produces one result. Less direct is editing: One action plus a previous state produces a subsequent state. Least direct is programming: One action plus a previous program text produces a subsequent previous program text, which when applied to a previous machine state produces a subsequent machine state. From this point of view, programming is a powerful but cumbersome kind of editing. If you can accomplish your goals without programming, so much the better.

The idea of programming as undesirable crystallized for me in a conversation with Larry Tesler of Apple Computer while he was writing an article on programming languages for *Scientific American* [Tesler 1984]. Larry remarked, "You're going along, getting stuff done, then suddenly you realize, (big sigh) I have to write a program." Tesler's comment convinced me to build an editor before attempting to build a programming language.







The advantage of a graphics over text is expressiveness. Historically, computers have used three forms of data to display results: numbers, text, and graphics. The earliest computers used numbers since numbers are the most efficient form of data for computers to process. Programmers found programs expressed in numbers difficult to read, so they developed textual mnemonics.

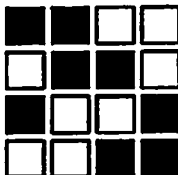
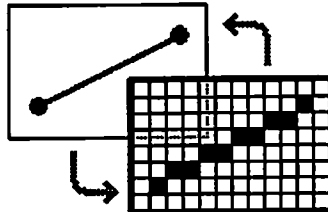
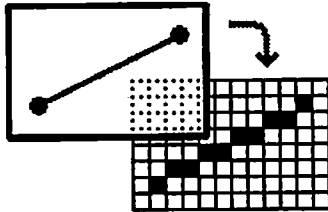
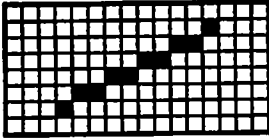
As computers offered more power, and users demanded more clarity, text became the main form of human-computer communication. Text editors and compilers enabled users to use words for both input and output, avoiding numbers entirely. Internally, computers still translated words into numbers for efficient execution.

Nowadays, graphics is becoming the main form of human-computer communication. Graphics is even less efficient for computers than text but even easier for users to understand. Current computers use graphics only for output. Direct manipulation user interfaces are beginning to suggest graphics as input.

Graphics is not just more expressive than text; it is more inclusive. Text is a special kind of graphics, just as numbers are a special kind of text. Thus we do not give up anything as we move from numbers to text to graphics. We only add to our expressive range.

## Pixels

What type of graphics editor could assist the entire design process? There are two types of graphics editors: painting systems and drawing systems. Both have advantages.



T E X T

1 0 0 1

Painting systems, such as MacPaint on the Macintosh [Atkinson], store pictures as arrays of pixels. The advantage of pixels over other representations of graphics is that pixels are what the user sees on the screen. Users can work quickly and responsively since the image on the screen is faithful to the representation in the computer. Painting systems are appropriate for the early chaotic stages of the design process or for projects that are inherently painterly.

In contrast, structured drawing systems, such as MacDraw on the Macintosh [Cutter], store pictures as abstract data structures. The advantage of structured graphics over pixel representations of graphics is that structures exist independent of screen resolution. Users can freely rotate, scale, and combine images without worrying about losing precision. Drawing systems are appropriate for the later, orderly stages of the design process or for projects which are inherently well-structured.

Many attempts have been made to combine the advantages of both painting and drawing systems. Current solutions allow pixel and structured graphics to coexist in overlapping transparent planes. Structured graphics may be converted into pixel graphics, but not vice versa.

The new idea in Viewpoint is to use pixels as the primary carrier of information in a structured graphics environment. In Viewpoint, structures are inferred from pixels and exist only for the duration of an operation. The moment an operation is completed, the structure drops back into pixels. Every structure can always be edited as pixels, no matter what other interpretations it may have.

Pixels are to graphics as characters are to text as bits are to numbers—they are the smallest atoms out of which larger structures are composed. Other decompositions are possible. Graphics may also be expressed in vectors, text in syntax trees, and numbers in exponential form.

Atomic decomposition has the virtue of simplicity. All atoms are alike. In contrast, numbers expressed in exponential form require two types of atoms: bases and exponents. Structures are "flat". In contrast, text expressed in syntax trees builds hierarchies of subtrees. Composition can be uniquely determined from final result. In contrast, graphics expressed in vectors is ambiguous: It is impossible to distinguish a vector from a twice-drawn vector.

Here are answers to common objections.

**Pixels are crude approximations to real graphics.** To a computer scientist writing programs to display pictures, "real" graphics lives in data structures. To a graphic designer using computers to design publications, "real" graphics lives on paper. In both cases, the screen is an intermediate representation of a final product.

In other cases, however, the screen stands for itself. To an interaction designer prototyping computer screens or an artist sketching ideas, the screen itself is the final product and so must be seen as "real".

**Pixels are inefficient and impractical.** In some cases, pixels already are the most efficient representation of graphic information. For instance, window managers scroll windows by copying blocks of pixels in the frame buffer rather than by going back to the original data structures to reconstruct the contents of a window.

As memory becomes cheaper and operations on blocks of pixels migrate into hardware, pixel processing will become more practical. The situation is similar to the early days of text processing. A text editor is difficult to justify on a machine with only 1,000 bytes of memory.

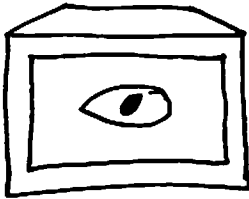
**Pixels, like bits, are too low-level to be meaningful.** In the early stages of the design process, the lack of structure in a low-level representation is desirable. Too much structure forces users to make premature decisions. Pixels allow tentative ideas to preserve their ambiguity.

Pixels do not preclude other structures. High-level structure can be parsed from pixels just as program syntax can be parsed from a string of characters.

**Deriving structure from pixels is a hard AI problem.** Grace Hopper ran into a similar objection when she first proposed high-level programming languages:

That December 1953 report proposed to management that mathematical programs should be written in mathematical notation, data processing programs should be written in English statements, and we would be delighted to supply the two corresponding compilers to translate to machine code. And this time the reason was that computers couldn't understand English words. [Wexelblat]

Viewpoint need not solve the vision problem any more than Fortran need solve the language problem. The challenge is to create an artificial visual language that is natural enough to be understood by users and precise enough to be parsible by computers.



## **Viewpoint**

To explore the consequences of basing a software environment on pixels, I built Viewpoint, an integrated text and graphics editor. I kept the behavior of Viewpoint as simple as possible, to focus on fundamental issues, but rich enough to include interesting issues.

The domain of Viewpoint is limited to a single screenful of pixels. Obviously, a practical editor would handle a larger area. Viewpoint uses four color planes to distinguish different types of information.

The functionality of Viewpoint is limited to editing. Viewpoint can edit three types of structures: text, fonts, and graphics. A single type of structure would not be enough to demonstrate the ability of pixels to handle multiple representations.

The representative form of structured graphics in Viewpoint is text. Originally, I planned to include many other sorts of structured graphics, such as lines and connected regions, but I found that word wrap was sufficient to demonstrate the idea of pixel parsing. Text uses a single, fixed width font, positioned on a fixed grid of square "cells" to simplify character parsing. Text and graphics editing operations both work on cells, blurring the distinction between typing and drawing.

Viewpoint behaves as if the pixels on the screen were the only record of the state of the system. I call this condition "visibility". Formally, a system is visible if the current screen state plus the current user input uniquely determines the next screen state, both for the computer and for the user. For the computer, this means that there must exist an algorithm that maps the current state of the frame buffer (the data structure that stores the pixels on the screen) plus the current user input into the next frame buffer state. For the user, this means that the screen must be easy enough to understand that for any given user input, the next screen is easy to predict.

All current software systems, with the possible exception of some video games, fail to be strictly visible. In practice, strict visibility is usually neither desirable nor practical. The purpose of Viewpoint is to demonstrate the consequences of strict visibility.

The formal definition of visibility proved elusive. I discovered that current formal theories of computation are based entirely on noninteractive batch computing and are therefore of no help for interactive systems. The theory of interactive computational systems, which includes such concepts as modelessness, has yet to be formalized.

I built Viewpoint at the Xerox Palo Alto Research Center in the Cedar programming language [Swinehart] running on a Dorado computer [Pier]. Viewpoint evolved through experimental programming. It would have been impossible to have planned the entire program in advance: So much of what I learned happened during the writing of the program. Because the program was small, I was able to try many revisions. Because it was simple, it worked reliably.

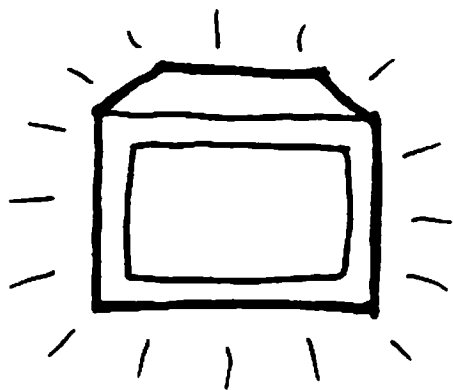
I tested Viewpoint on dozens of people. I wrote a short user's manual (see Appendix) that allowed a new user to learn the system without any prompting from me. By seeing where users got confused, I was able to make the user interface easier to use. I went through a half dozen versions of the cursor until I found one that was satisfying. I was also to discover places where I had failed to adhere to strict visibility. Patterns of use suggested what functionality to add next. Ultimately, the program allowed me to practice a new way of thinking about computer displays.

The primary result of Viewpoint is that a strictly visible text and graphics editor is possible. The program itself provides examples of techniques for implementing strict visibility, such as pixel parsing, the cursor plane, and the visual boot.

In addition, Viewpoint suggests research opportunities. Along the lines of pixel-based systems there are pixel parsing algorithms, hardware support for pixel operations, and artificial visual languages for human-computer interaction. More generally, there are editor-based systems, alternate basis representations other than pixels, theory of interactive systems, and interdisciplinary research in computers and graphic design.

## Timeline

- Undergrad school 1973** Computers, computer music, and computer graphics.  
Visual thinking class with Robert McKim.  
Basic graphic design class with Matt Kahn.  
Met Doug Hofstadter.  
BA in music, with studies in mathematics.  
Wrote article on four-dimensional optical illusions.
- Graduate school 1979** Started as graduate student (Masters in CS) Fall 1979.
- Metafont 1980** Gave Metafont demos.  
Programming AMS-Euler font in Metafont.  
Xerox PARC. Started as consultant.
- Inversions 1981** Work with Richard Weyrauch on computational philosophy.  
Wrote and produced *Inversions* book.  
Viz Din (visual programming language discussion group with Fred Lakin and Warren Robinett) started.  
Started doing graphic design jobs.
- Computer languages 1982** Started interdisciplinary PhD.  
Taught visual thinking.  
Dave Siegel took over Euler.
- Visual programming 1983** July started work at Information Appliance.  
August ATypI conference.
- Thesis proposal 1984** Wrote dissertation proposal.  
Dropped idea of programming. Added idea of pixels.  
Introduction of Apple Macintosh computer.
- Programming 1985** Lecture at Hewlett-Packard: the field of user interface design and why it doesn't yet exist.  
Learned to use Cedar programming environment at Xerox.  
Wrote first draft of Viewpoint program.
- Thesis, draft 1986** Taught "Graphic Invention for User Interfaces" at Stanford with William Verplank.  
Wrote dissertation, draft 1.
- Thesis, final 1987** Rewrote program to match writeup.  
Revised theory.  
Dissertation defense, including a videotaped demonstration of Viewpoint.  
Wrote final version of dissertation.



### **3 Demonstration**

This chapter the Viewpoint program, a text and graphics editor I built as the core of my research.

- About the programming environment
- About the illustrations
- Actions: drawing, selecting, copying, typing
- The split brain
- The visual boot
- What happens if...?



## **About the programming environment**

Viewpoint was programmed in the Cedar programming language, which lives in the Cedar programming environment built primarily for internal research at the Xerox Palo Alto Research Center. I chose Cedar over the other languages at Xerox—Interlisp-D and Smalltalk—because only Cedar supported color graphics at the time.

Viewpoint runs on the Dorado, a powerful personal computer also built for internal research at Xerox. The Dorado comes equipped with a keyboard and three-button mouse for input, a bitmapped black and white display and optional color raster display for output.

Viewpoint uses the keyboard, mouse, and color display, but not the black and white display. The color display is configured to display 640 pixels across by 480 pixels down. Each pixel may be one of 16 possible colors. The palette of 16 colors is chosen from a much larger range of displayable colors.

Pixels are stored in a frame buffer. Viewpoint treats the 640 by 480 frame buffer as 64 across by 48 down ten-pixel by ten-pixel "cells". Cells are a software convention; the physical hardware imposes no such restriction. Viewpoint displays a light blue grid to make cell boundaries apparent to the user. The grid includes the 36 pixels around the edge of each cell. All grid lines except the outermost borders are two pixels thick, one pixel for each of the neighboring cells.

Viewpoint treats the 16 colors as mixtures of four "color planes": black, red, green, and light blue, all against a white background. Like cells, color planes are strictly a software convention. Color mixtures were chosen to give the illusion of transparency while keeping all colors distinct. For instance, truly transparent red superimposed on black would yield black in the subtractive model of color or red in the additive mode; to keep the colors distinct, I've adjusted the lightness of the composite color to pink.

## **About the illustrations**

To be able to follow a demonstration of Viewpoint, you must be able to see each and every pixel. Therefore, I have made the screen illustrations as large as possible. This meant cutting the screen from the full 640 pixels across to only 560 pixels across.

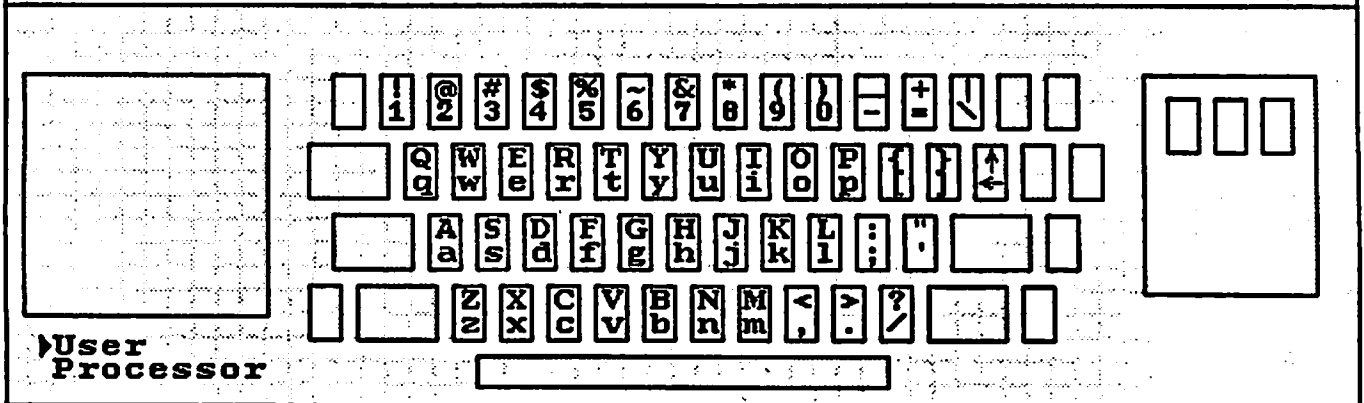
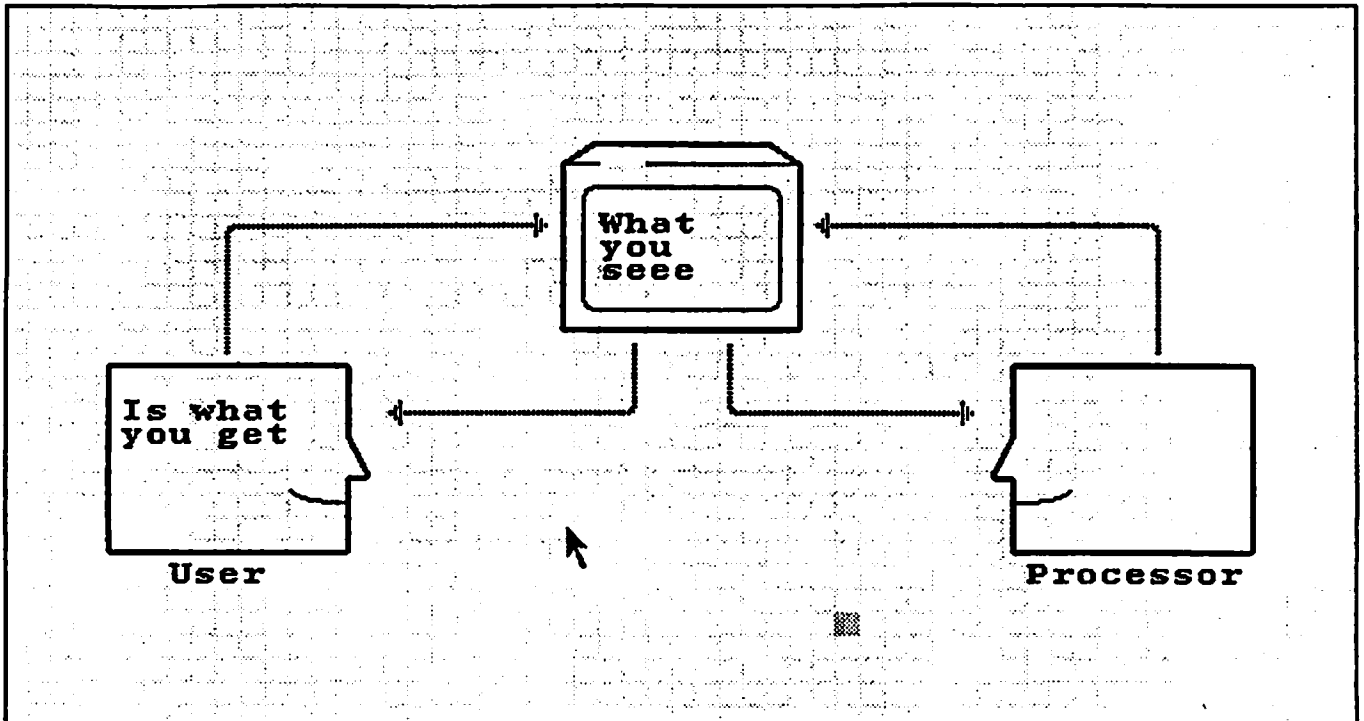
Color is a crucial element of Viewpoint. Unfortunately, color is inconvenient to reproduce in publication. Therefore, I have chosen the following substitutions.

Light blue is shown as very light gray. The only light blue elements are the grid lines.

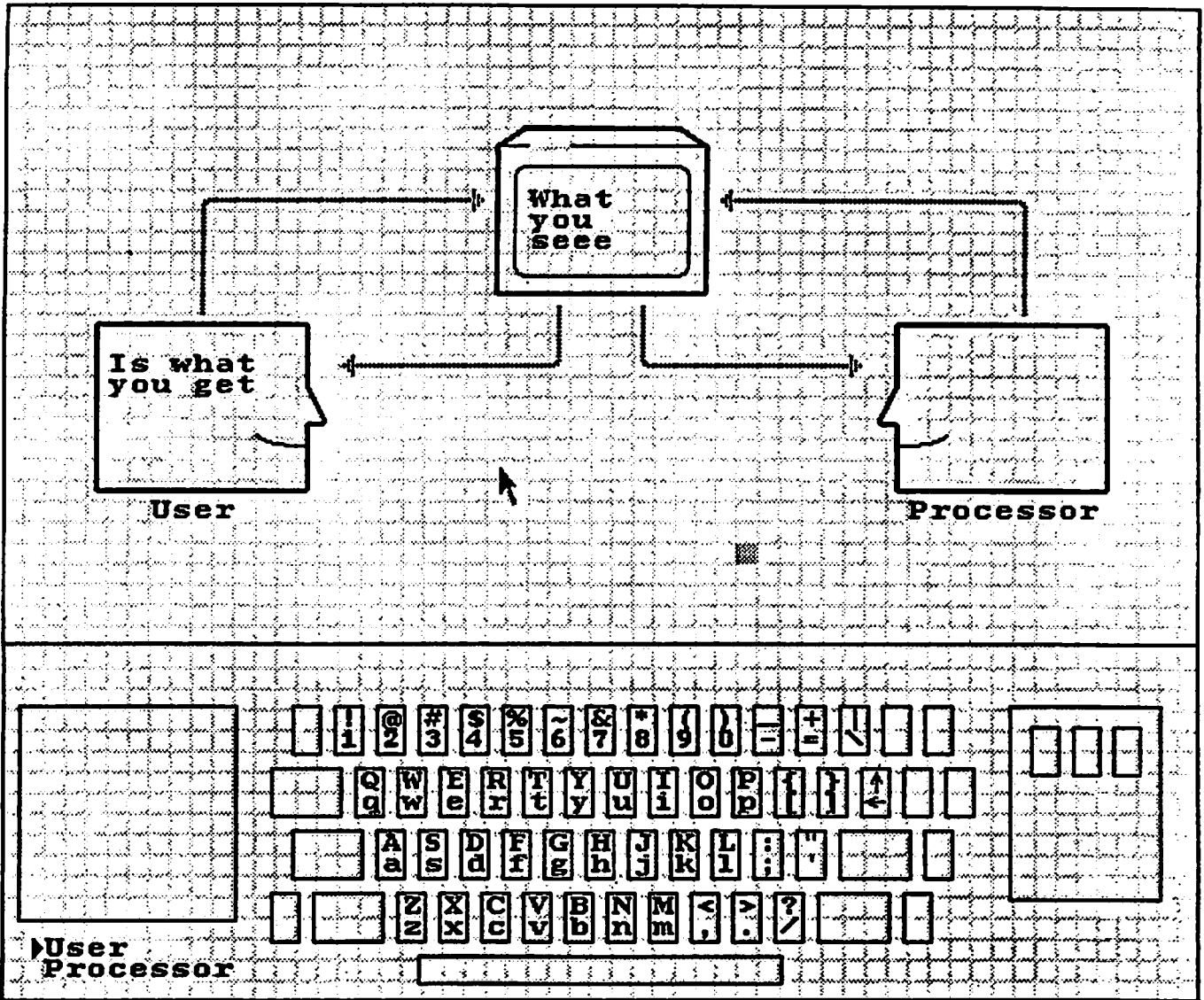
Green is shown as light gray. The only green element is a single shaded cell called the "selection". In the illustrations, green obscures light blue. In the actual system, the colors mix to produce different shades of blue-green.

Black is shown as black. Many elements are black, including both text and graphics. In the illustrations, black obscures green and light blue. In the actual system, the colors mix to produce different shades of blue and green.

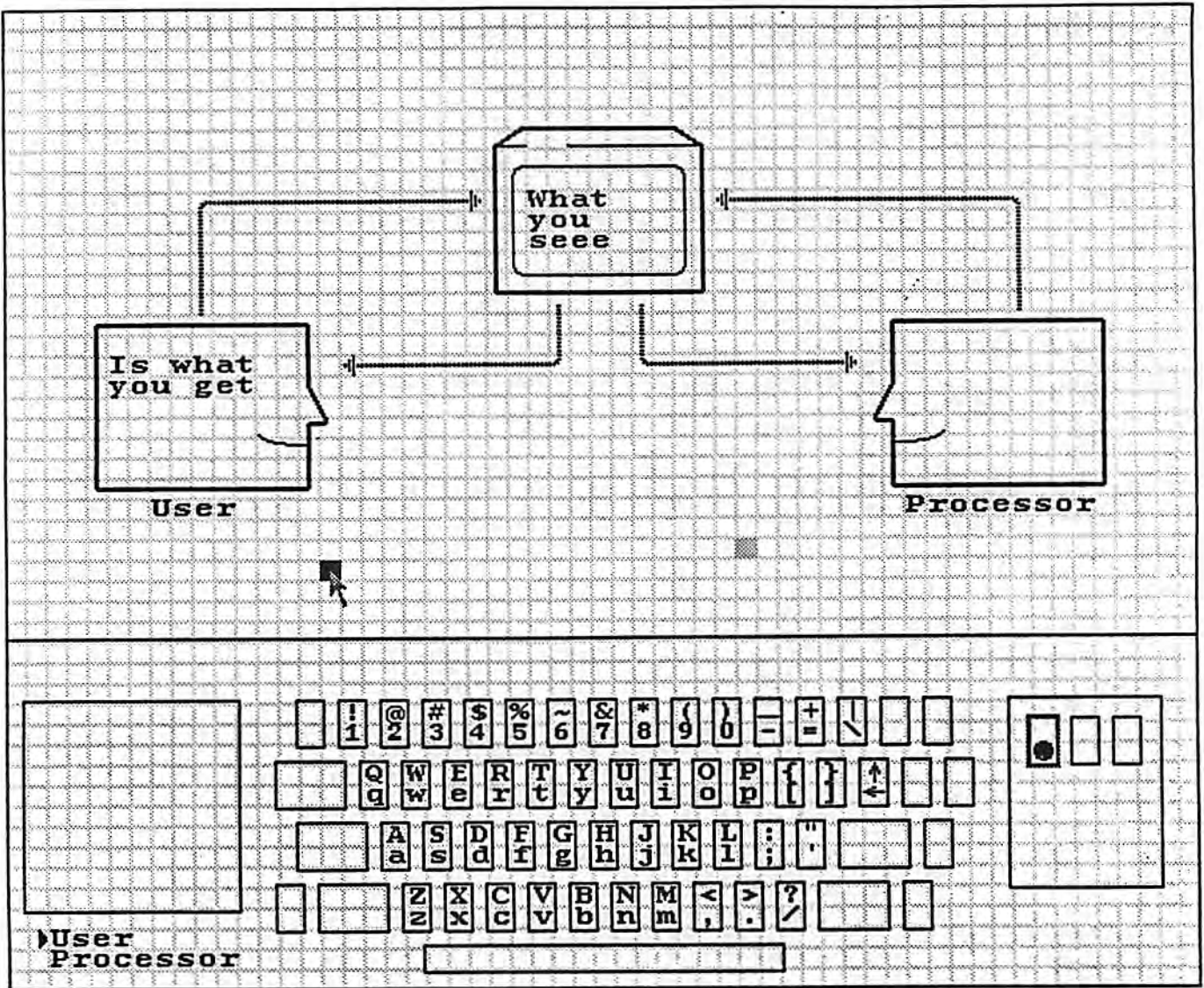
Red is shown as medium gray. The only red elements are the cursor and rectangles called "key highlights" that show which keys are down. In the illustrations, red obscures all other colors. In the actual system, the colors mix to produce different shades of red, purple, and orange.



This is a typical Viewpoint screen. The pictures of the keyboard and mouse in this illustration actually appear on the screen, because they have been previously drawn by the user. In fact, everything black was drawn by the user, except for the triangle in the lower left corner. The arrow is the cursor, which appears red on the screen. The gray square is the selection, which appears green on the screen. The pale gray lines show the boundaries of ten-pixel by ten-pixel cells.



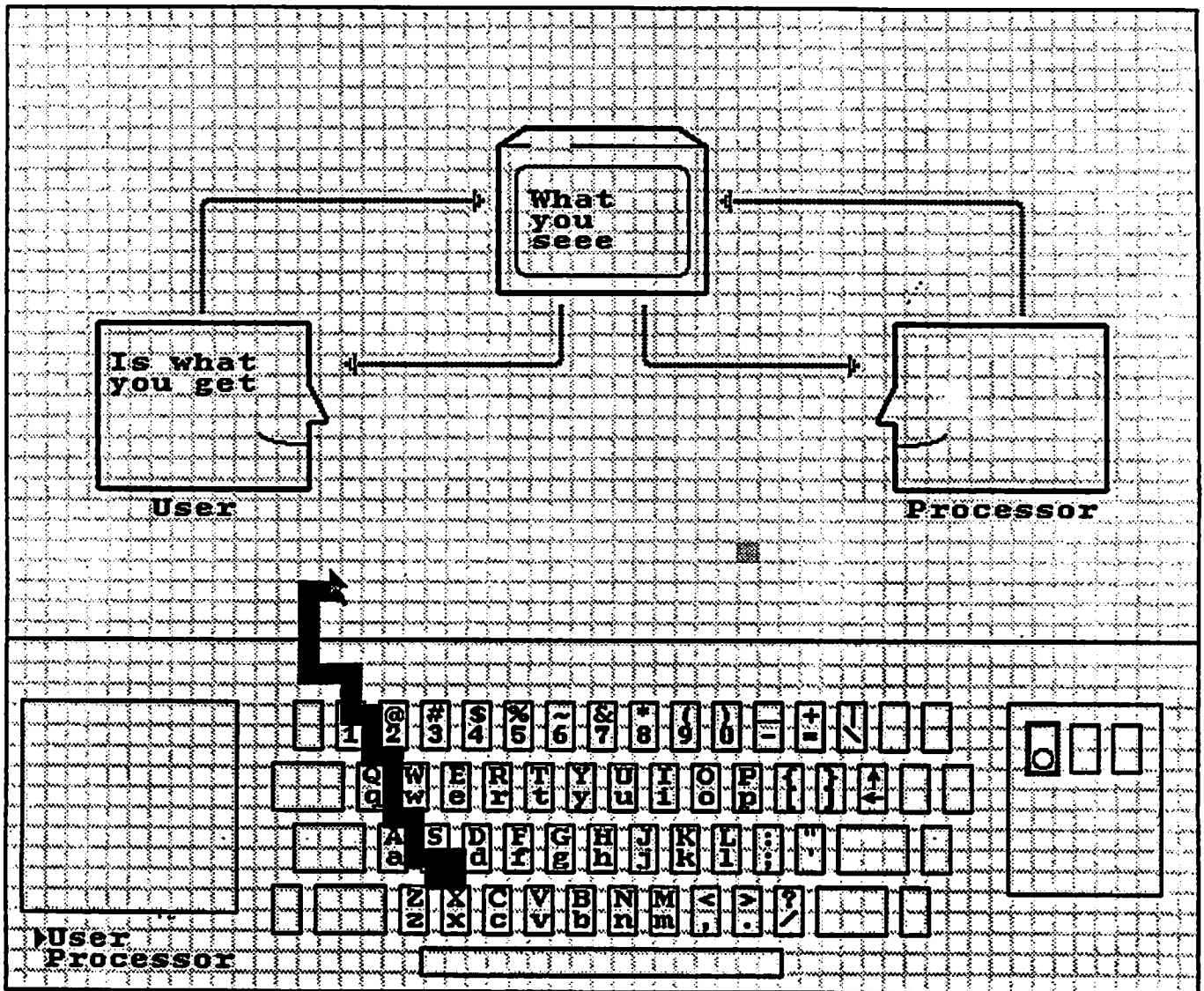
In this illustration, the user has almost finished drawing a diagram of a user and a processor communicating through a bitmapped screen. During this demonstration, we will use Viewpoint to finish the diagram. When we're done, the box on the right will say, "Is what the computer gets". Throughout the demonstration, I will deliberately make mistakes to bring out Viewpoint's more surprising behaviors.



**Drawing.** Moving the mouse moves the cursor. Each of the mouse buttons causes an action. The left mouse button is used for drawing. When the cursor points to a white cell, pressing the left mouse button turns the cell black.

Notice that the screen image of the left mouse button is highlighted in red. Whenever a key is pressed, a two-pixel-wide red border is drawn around the outside of the corresponding key image on the screen in order to make the state of the keyboard visible. The border has two one-pixel-wide layers. The inner layer, called the "key highlight", stays red as long as the key is held down. The outer layer, called the "key trigger", appears for only one cycle when the key has just been pressed or released. Only the key highlight is shown above. We will ignore the key trigger for now.

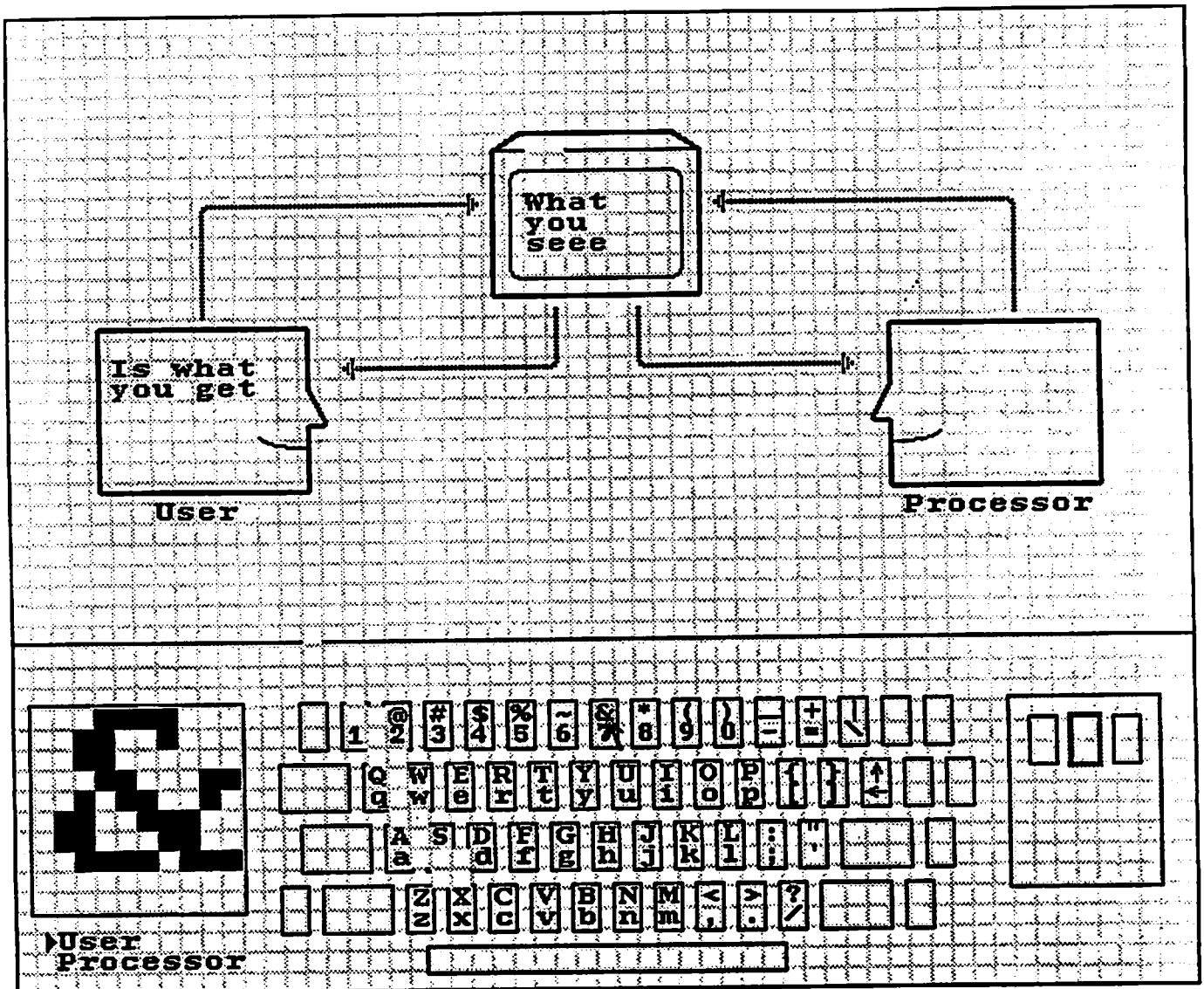




When the cursor points to a black cell, pressing the left mouse button turns the cell white. Notice that a white circle appears inside the image of the left mouse button to show that we are drawing in white, not black. The draw color is not just for the benefit of the user; it is also the way the system itself remembers whether it is still drawing black or drawing white.

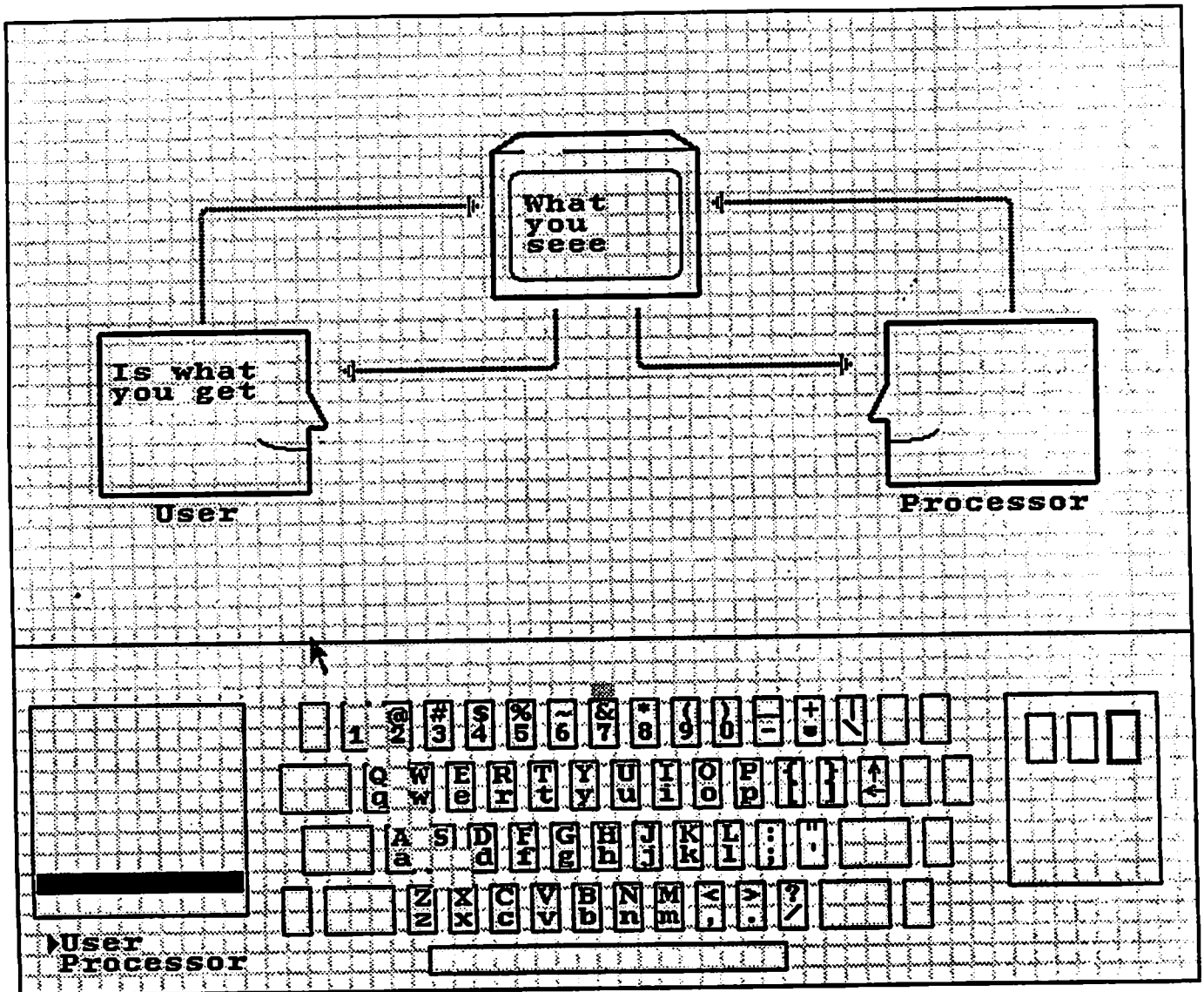




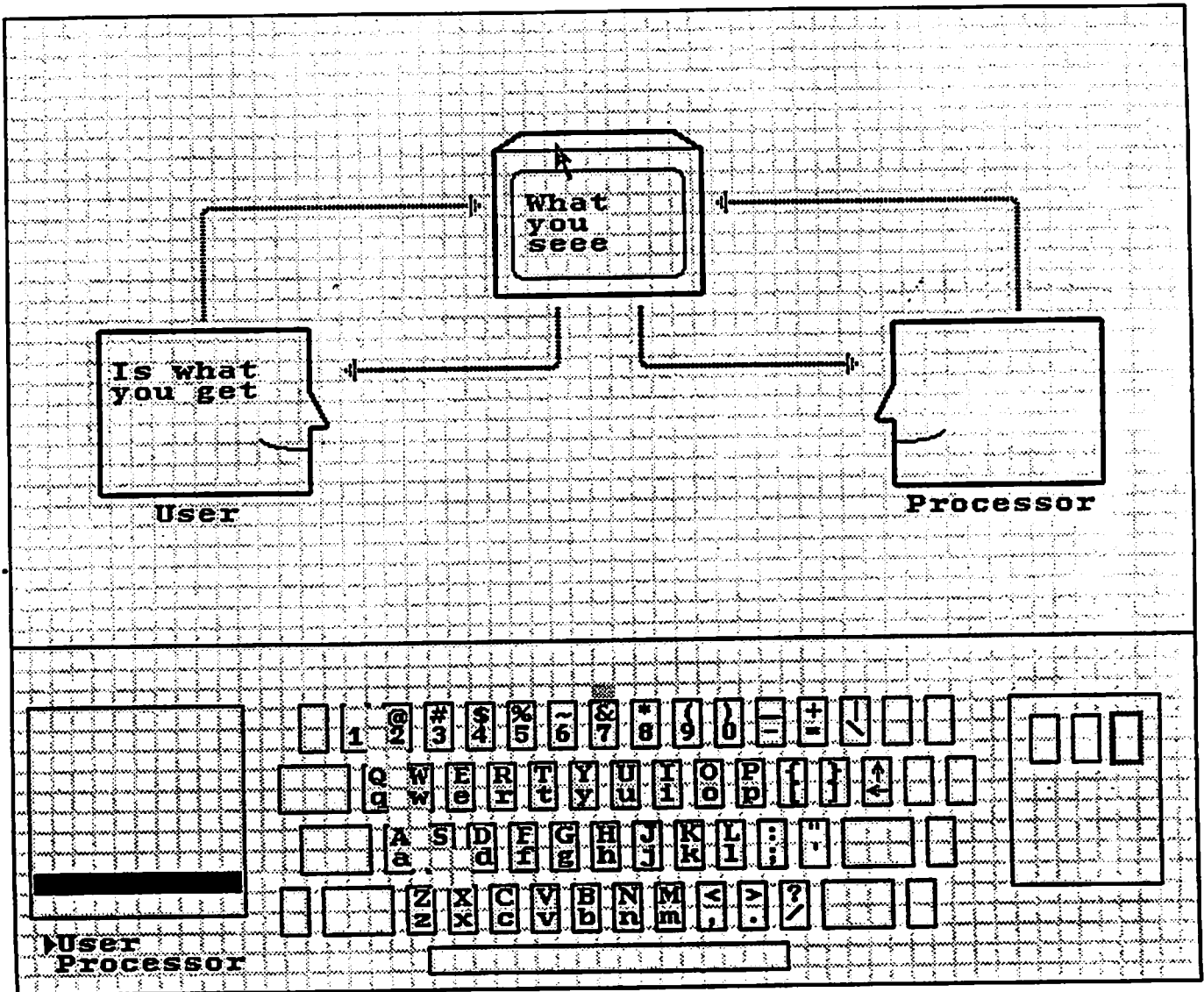


**Selecting.** Pressing the middle mouse button selects the cell at the cursor. The currently selected cell is shaded green. Selecting a cell causes the selected cell to be reproduced ten times larger in a special area at the bottom left of the screen called the "puffbox".

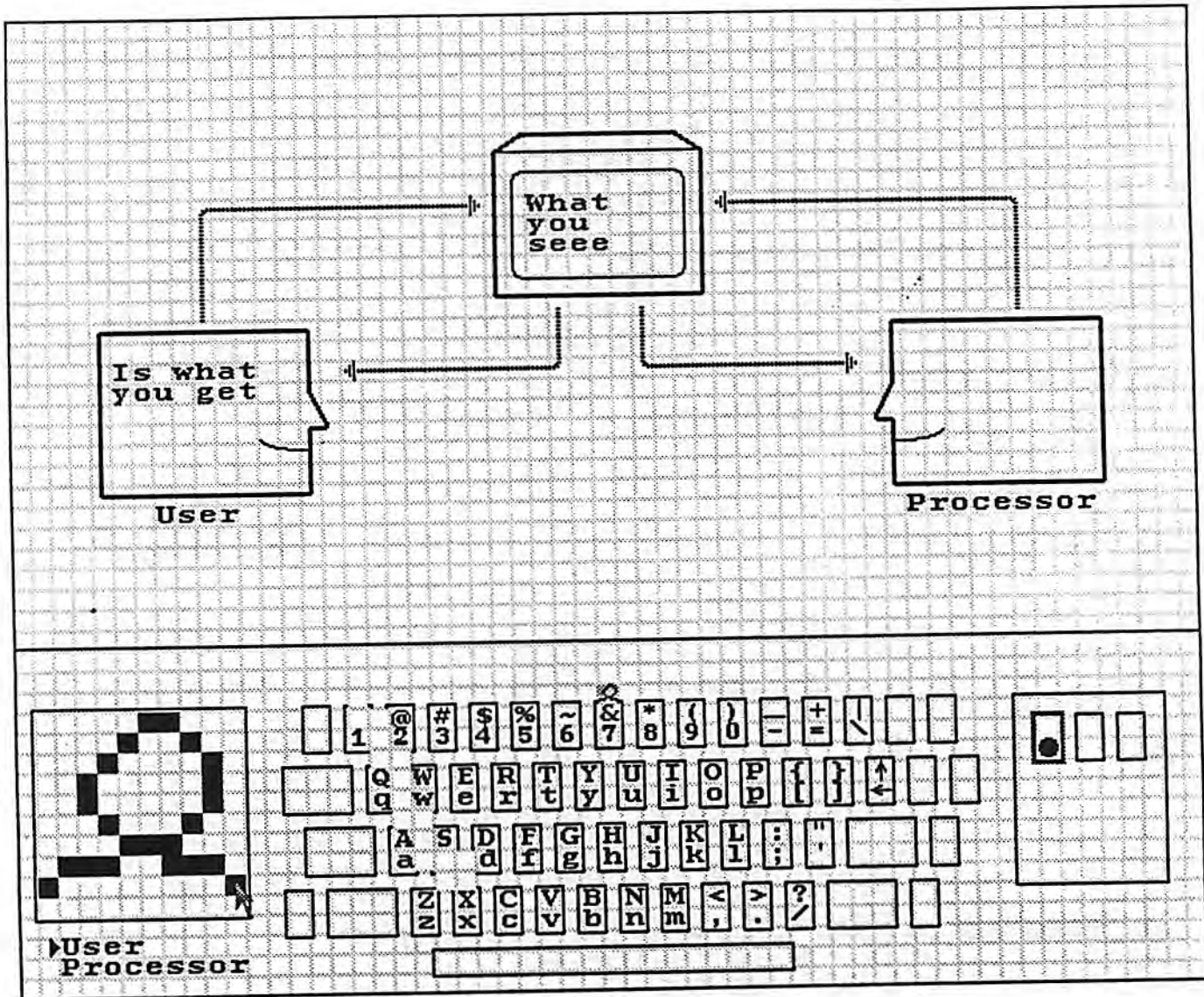




**Copying.** Pressing the right mouse button copies the selected cell to the cell at the cursor. For instance, we can copy the top edge of a key to fix the break in the long horizontal line...

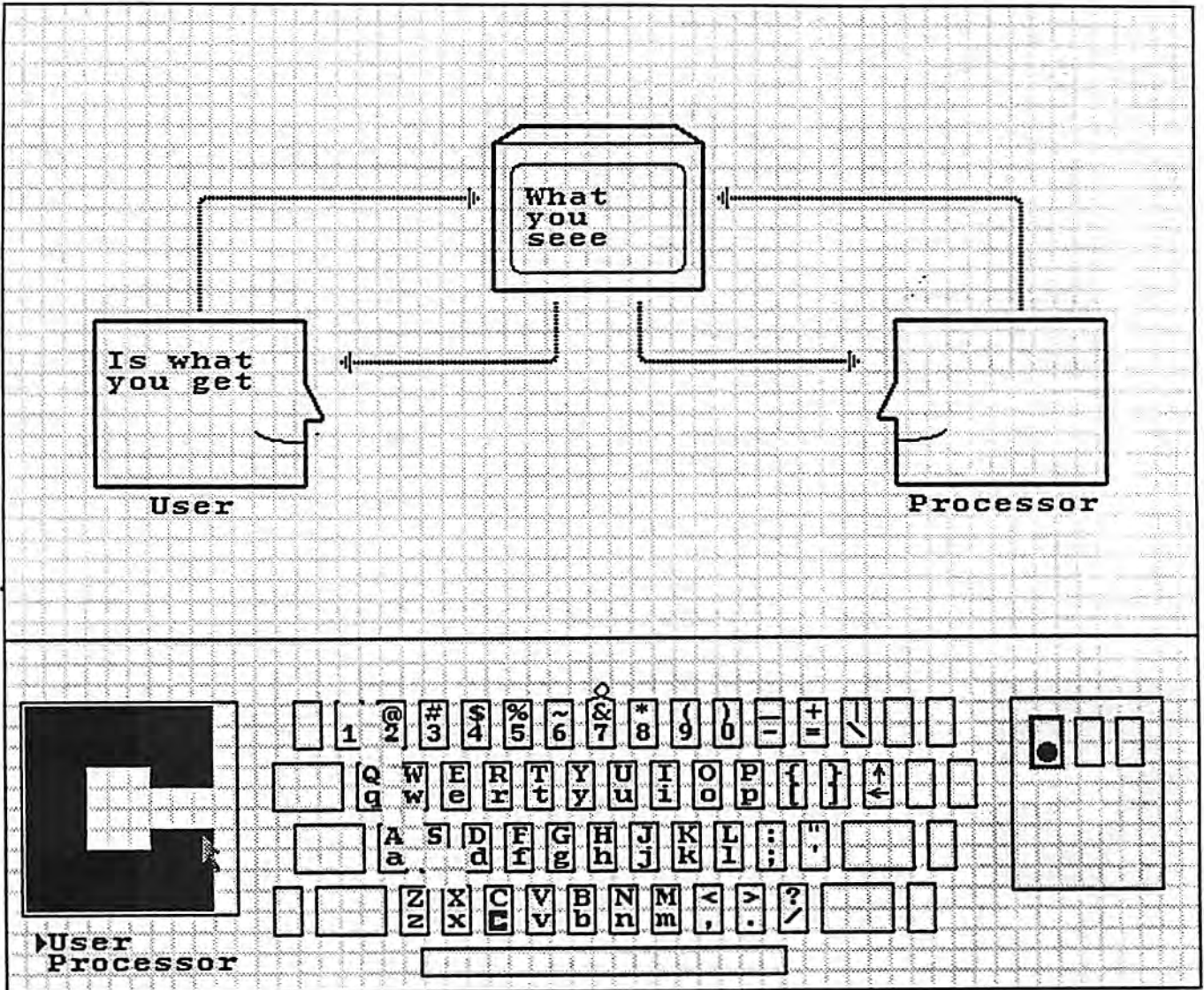


or to fix up the top edge of the monitor.

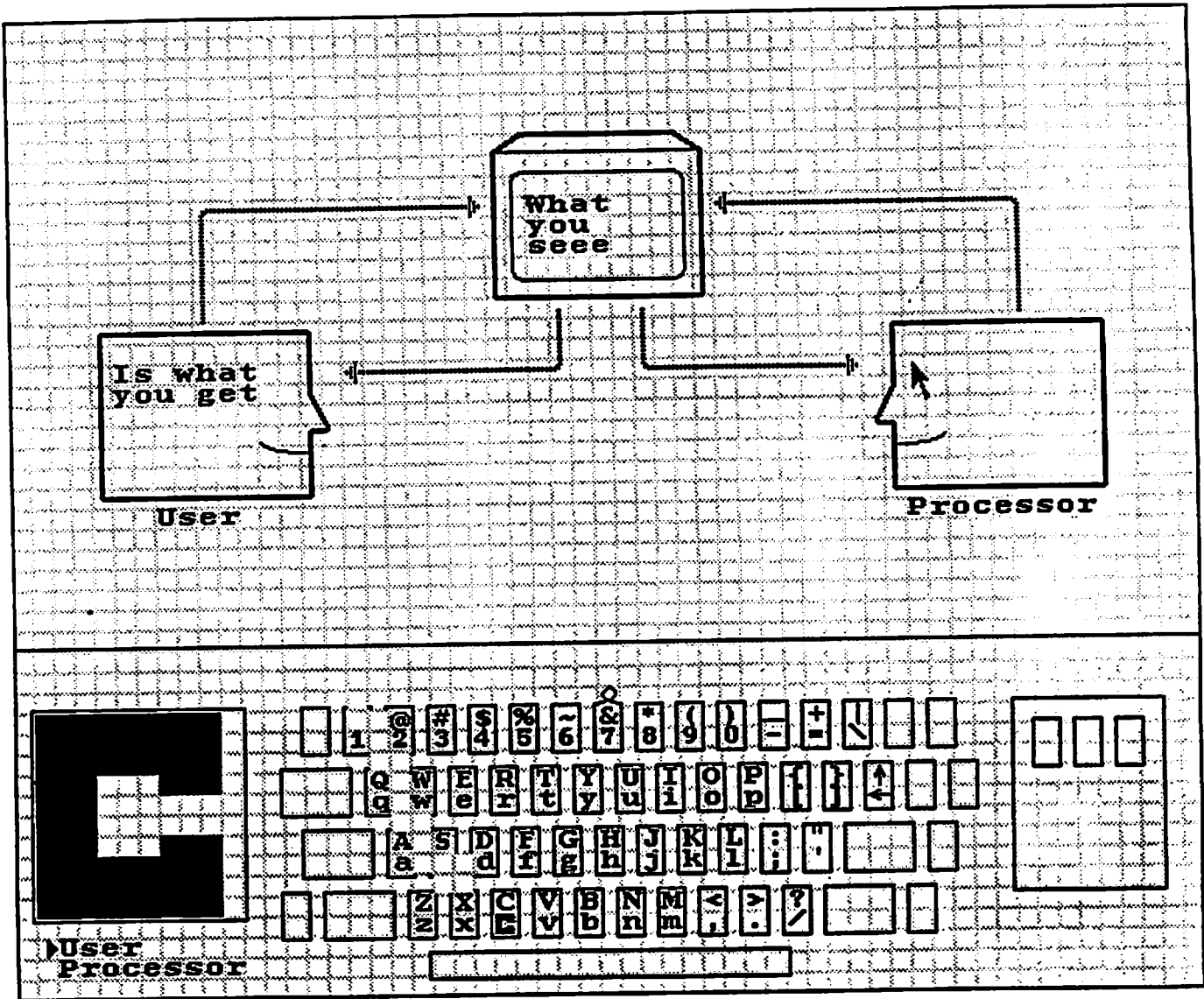


**Puffbox.** Each cell in the puffbox corresponds to a pixel in the selected cell. Drawing inside the puffbox affects the corresponding pixels in the selected cell. We can draw in the puffbox to tie a knot in a key border...



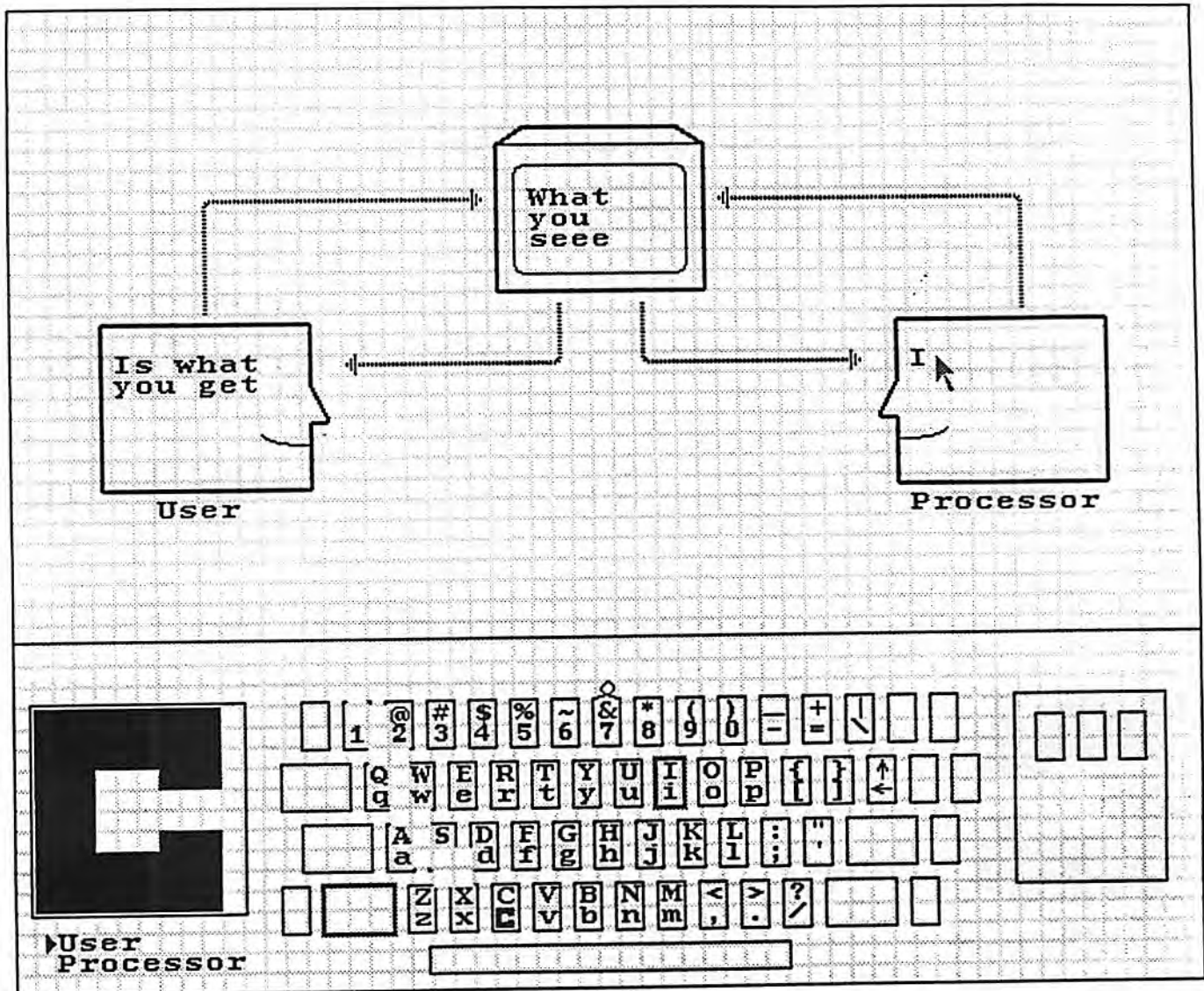


or to redesign the font. Notice that the other c's on the screen are not affected.

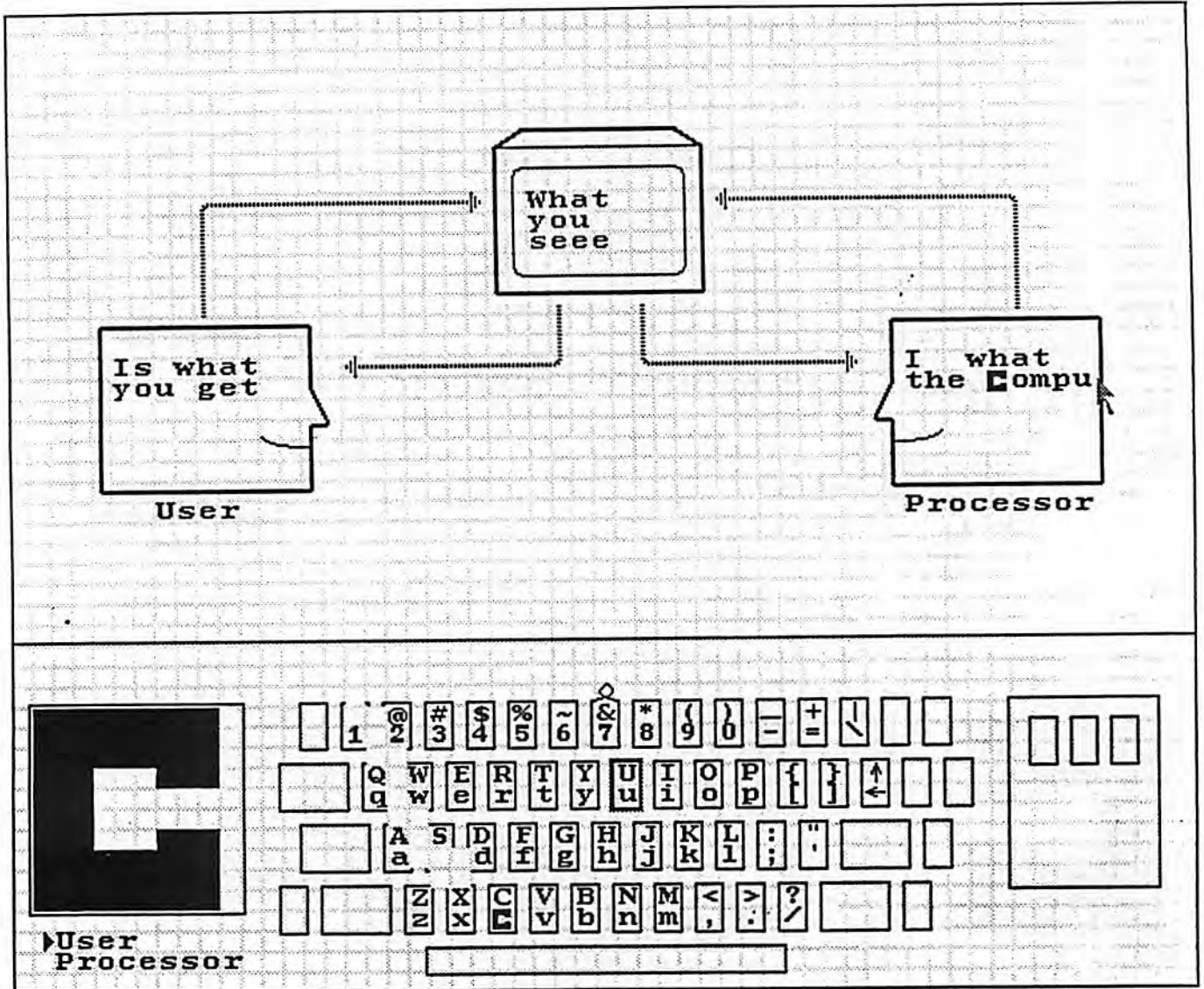


**Typing.** So far, we have seen drawing, selecting, and copying. The final action in Viewpoint is typing. Here, we have positioned the cursor in preparation for typing the phrase "Is what the computers gets."





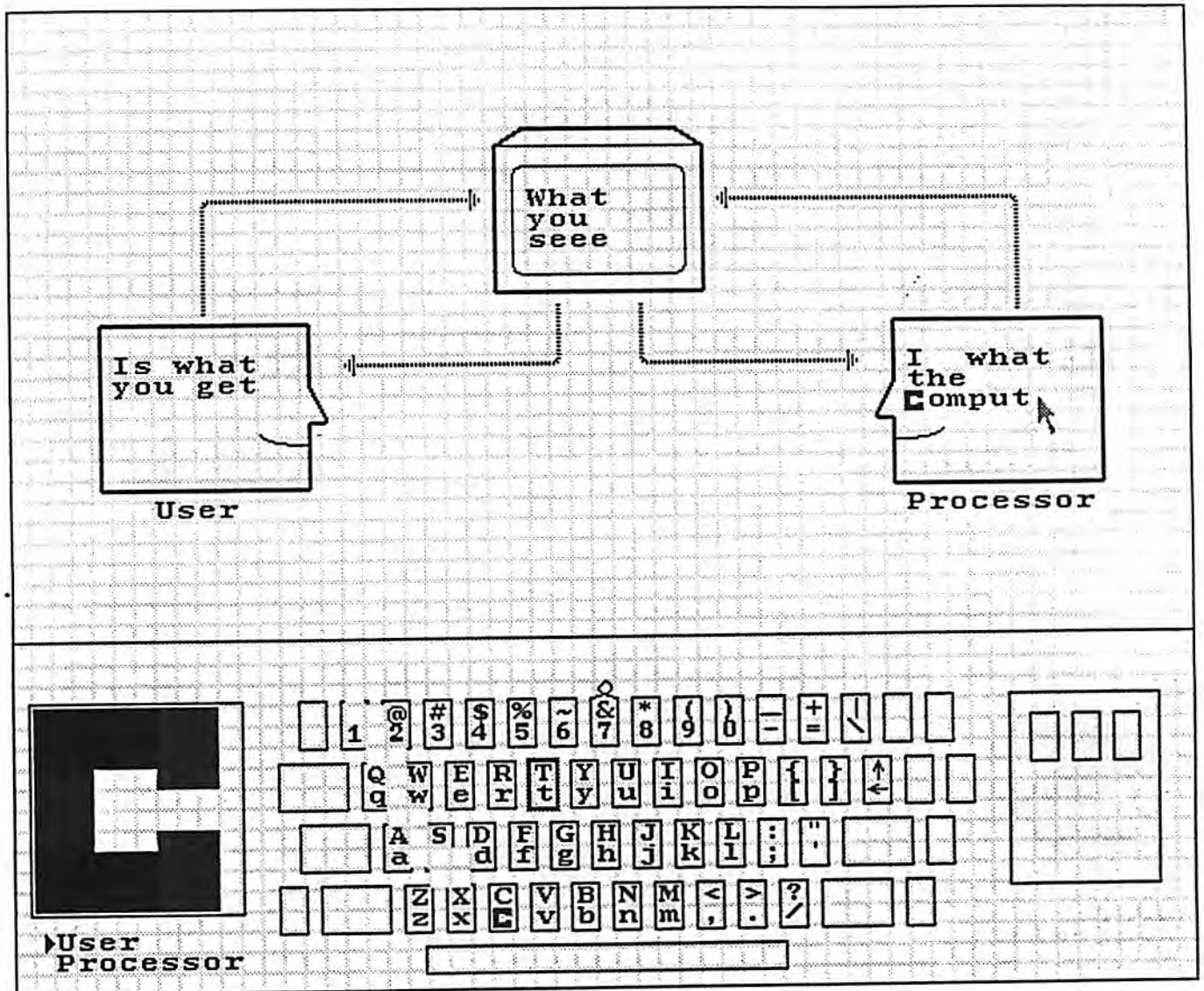
Typing a key copies the corresponding cell in the keyboard image to the cell at the cursor, then advances the cursor one cell to the right. Notice that each key image contains two character images, one for lower case and one for upper case. Holding either Shift key while typing causes the upper image to be copied—in this case an uppercase *I*.



Typing uses characters exactly as they appear in the current keyboard image. Thus the *s* appears here as a blank, and *c* appears in its newly emboldened form.

Word wrap occurs when the cursor encounters a right margin. Here, we have already wrapped once on the word *the* and are about to wrap again on the word *computer*.

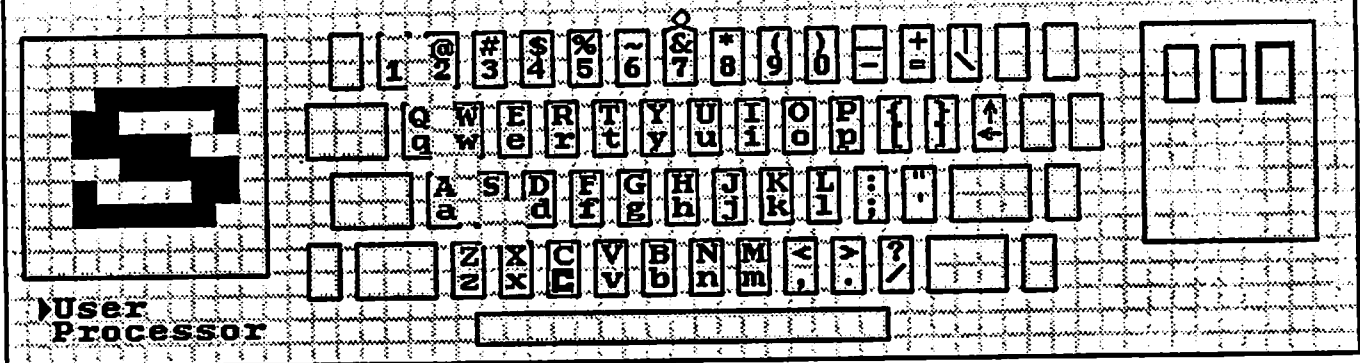
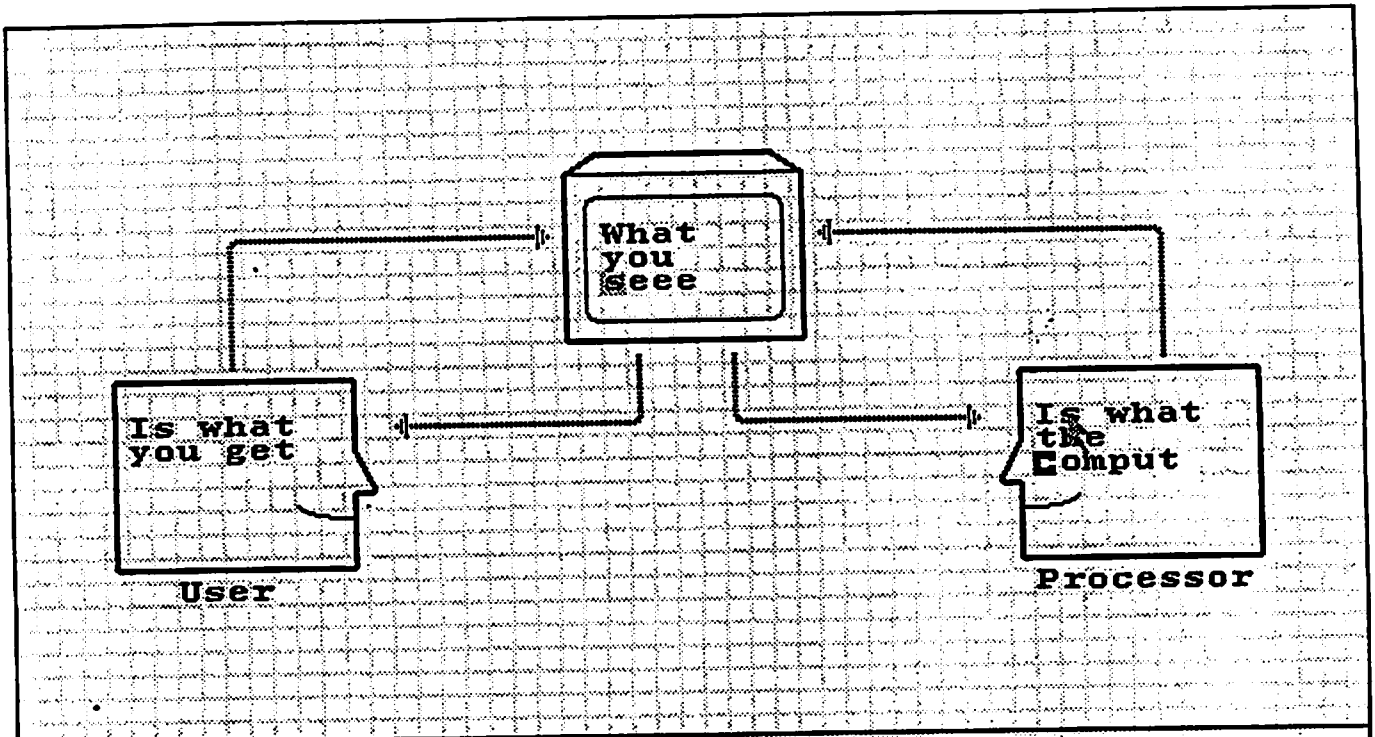
Margins are delimited by "noncharacters"—cells that do not match any character on the current keyboard. Cells that do appear on the keyboard are called "characters". Here the cell at the cursor is a thick vertical bar that doesn't quite match the vertical bar or uppercase *I*.



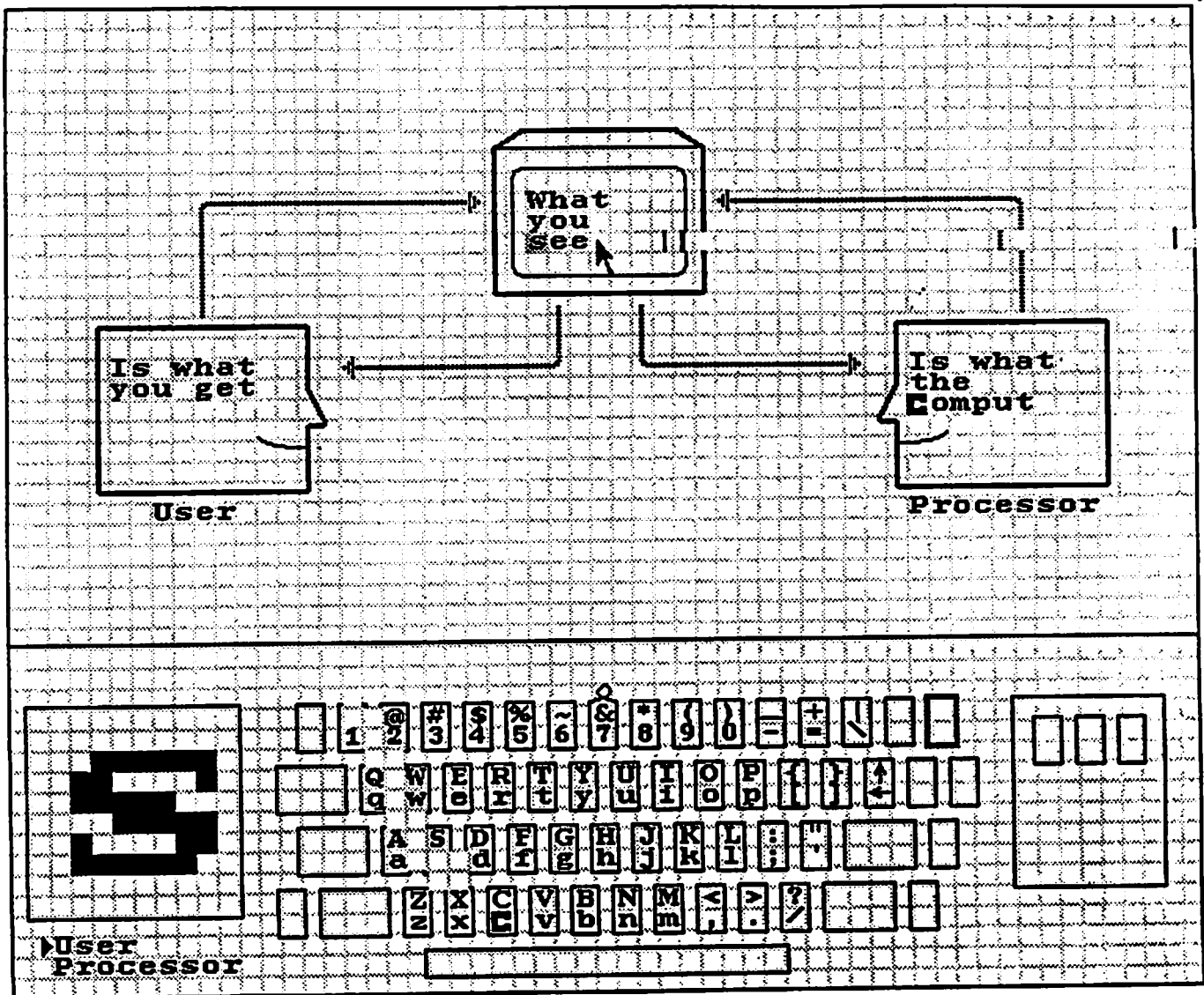
Word wrap erases the last word from the current line and rewrites it at the beginning of the next line, followed by the newly typed character.

To find the start of the last word, word wrap scans left cell by cell, starting from the cell to the left of the cursor until it encounters a space character. The space character itself can be redesigned, just like any other character; it's located at the left end of the Spacebar image.

To find the beginning of the next line, word wrap moves the cursor down to the next row of cells, then searches for the left margin. As before, margins are delimited by noncharacters.



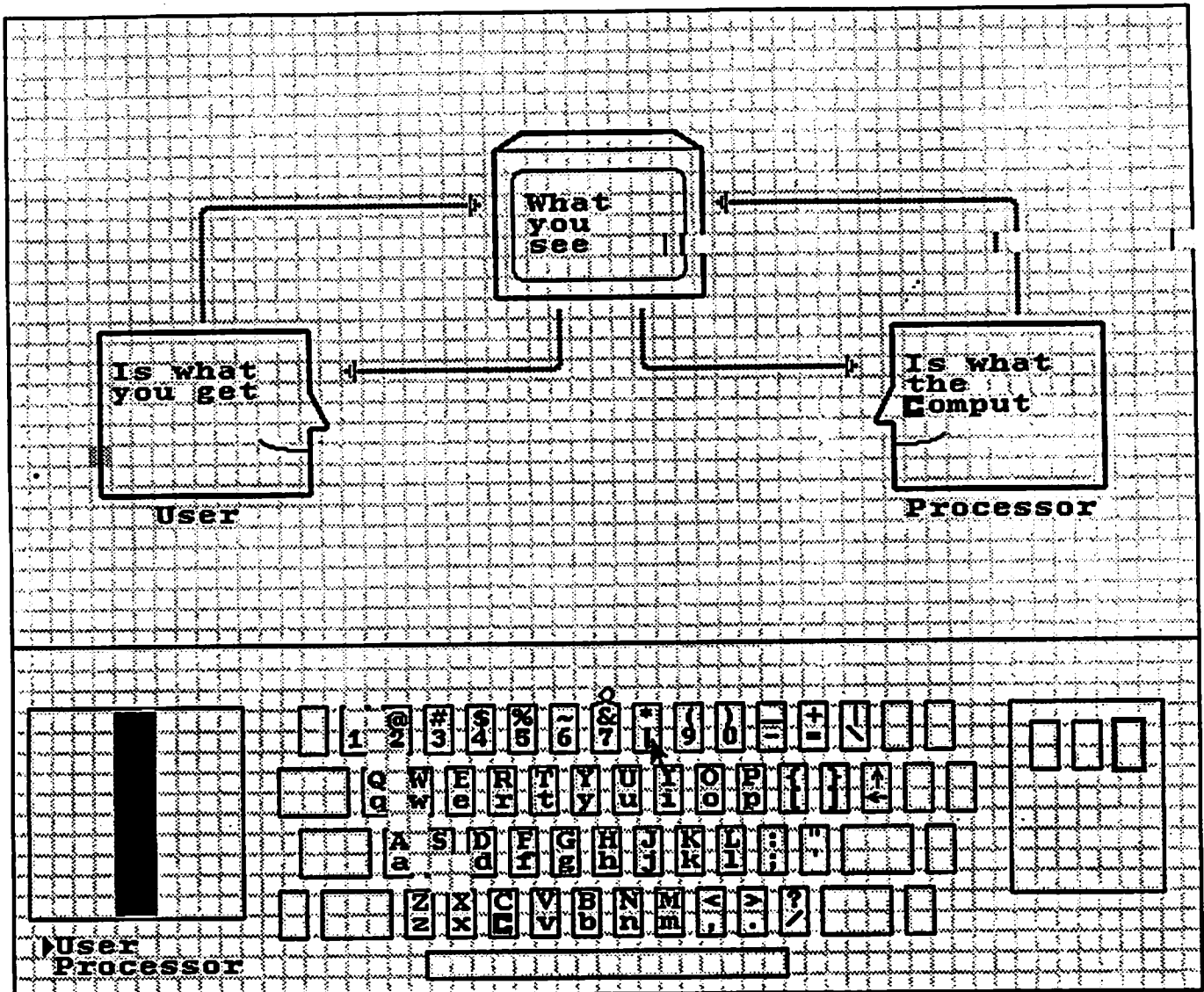
Typing has the same effect as copying. Here, we have filled in the missing *s* by copying from another sentence. We also could have copied the *s* onto the keyboard, then typed the *s* into place.



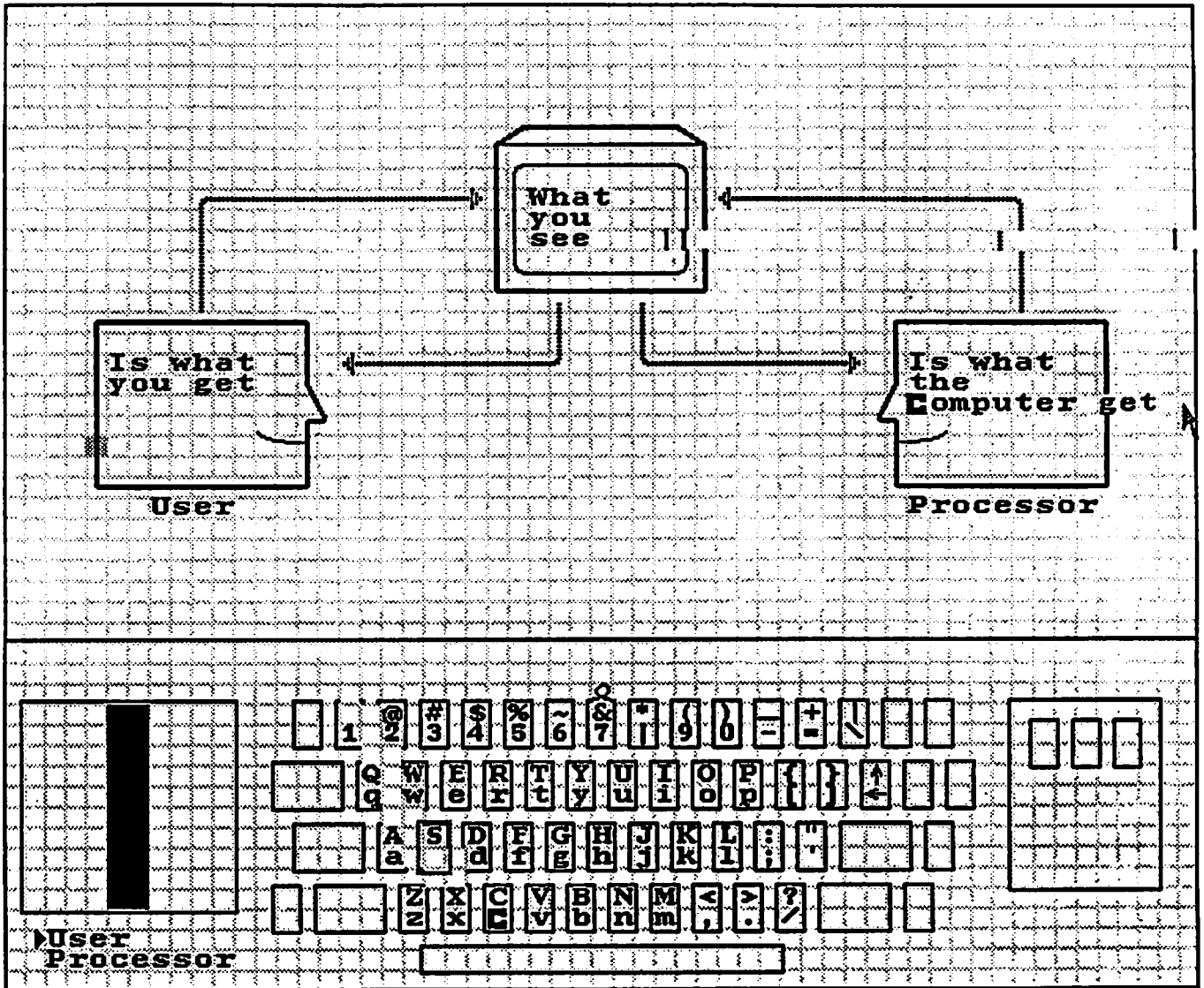
Let's type the Del key to delete the extra *e* in *see*. The Del key, which is unmarked on the screen, is located in the upper right corner of the keyboard.

This action messes up our drawing. Del moves all cells to the right of the cursor left one cell and inserts a blank cell at the end. In a real text editor, this probably would not be a desirable effect. I've included this peculiar operation as a reminder that there is no inherent difference between text and graphics in Viewpoint. Everything can be treated as text, even cells that were never typed.

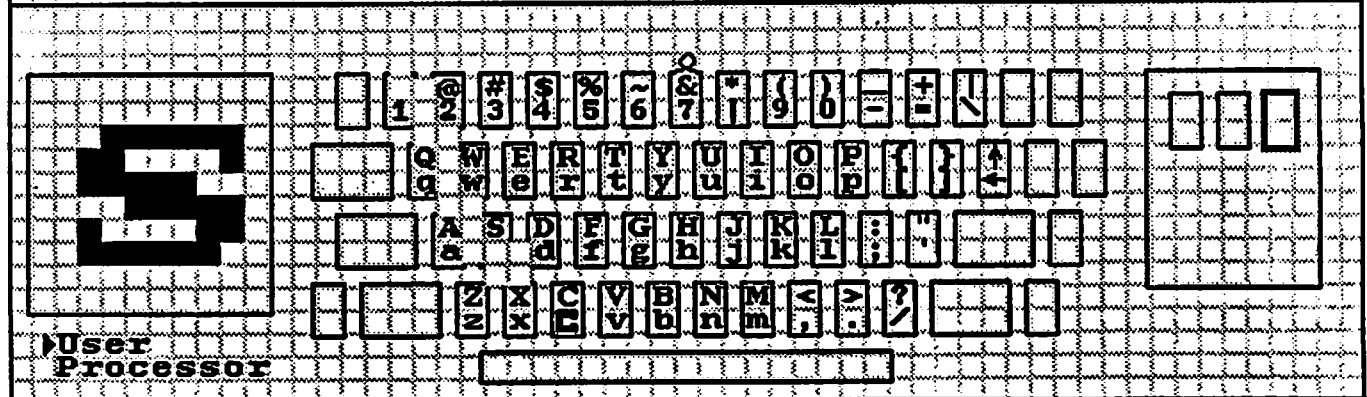
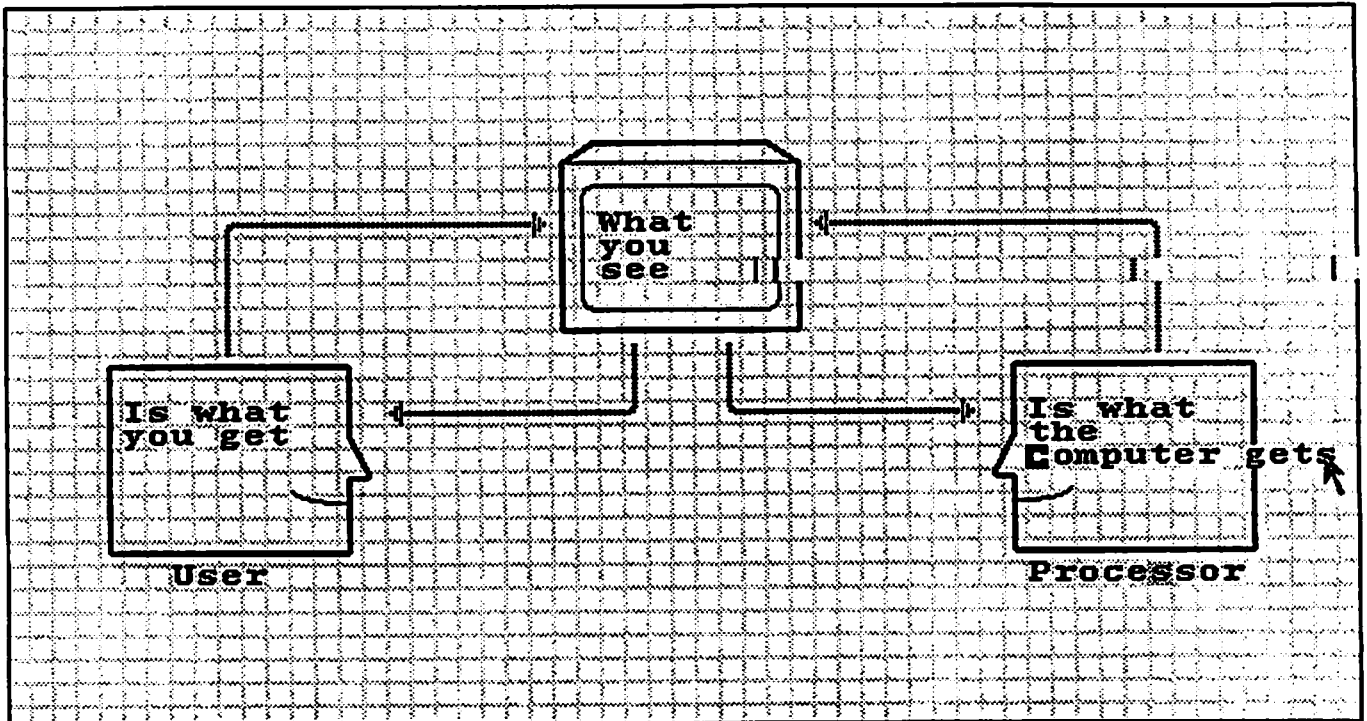




Since word wrap depends on the font, we can change its behavior by editing the font. Here, we've copied a border pattern onto the *b* key.

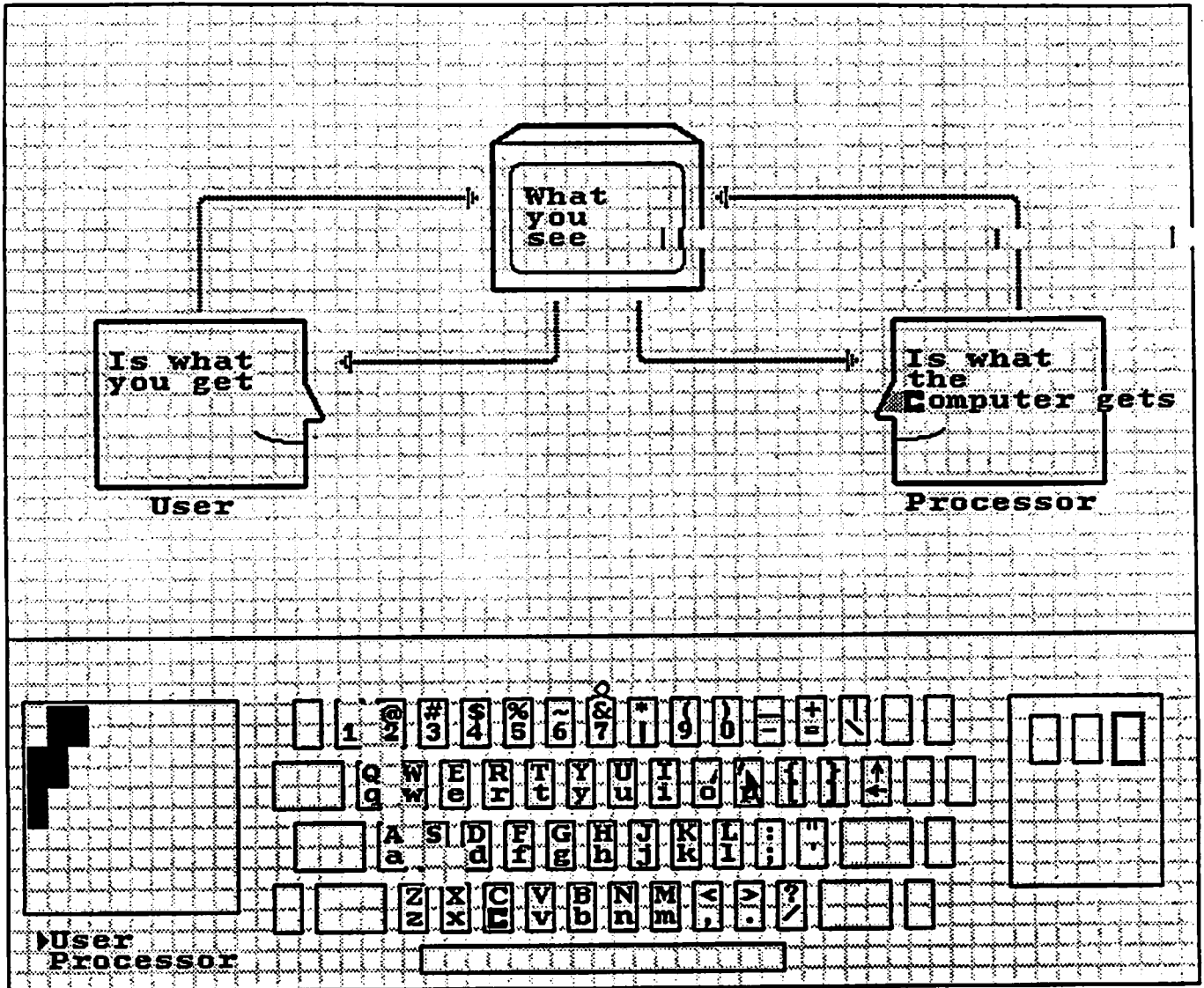


Now, typing no longer causes word wrap at the edge of the box but keeps laying down characters. Notice that the last letter typed was s, which appears as a blank on the keyboard.



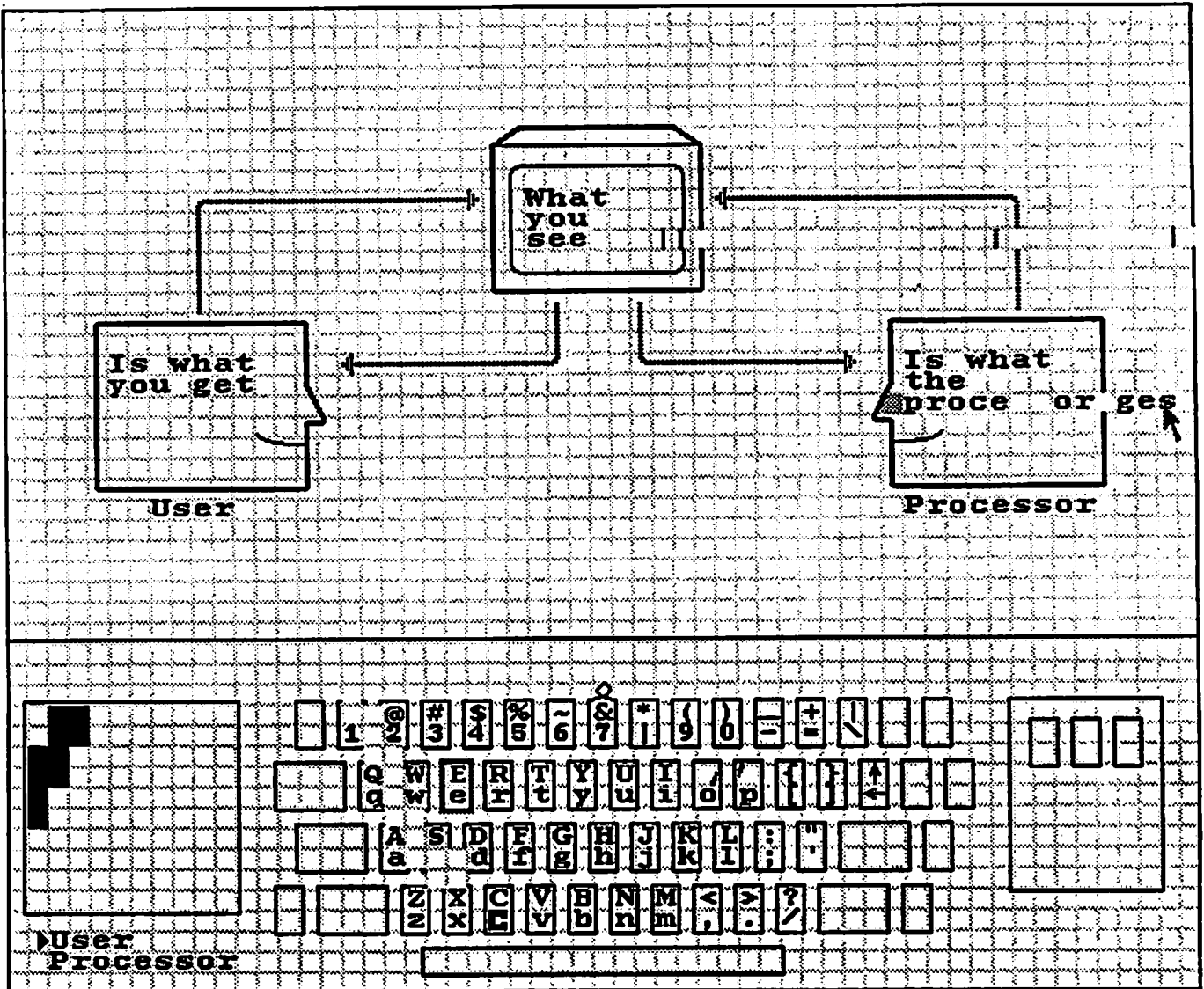
As before, we can fill in the missing s by copying a letter from another sentence.



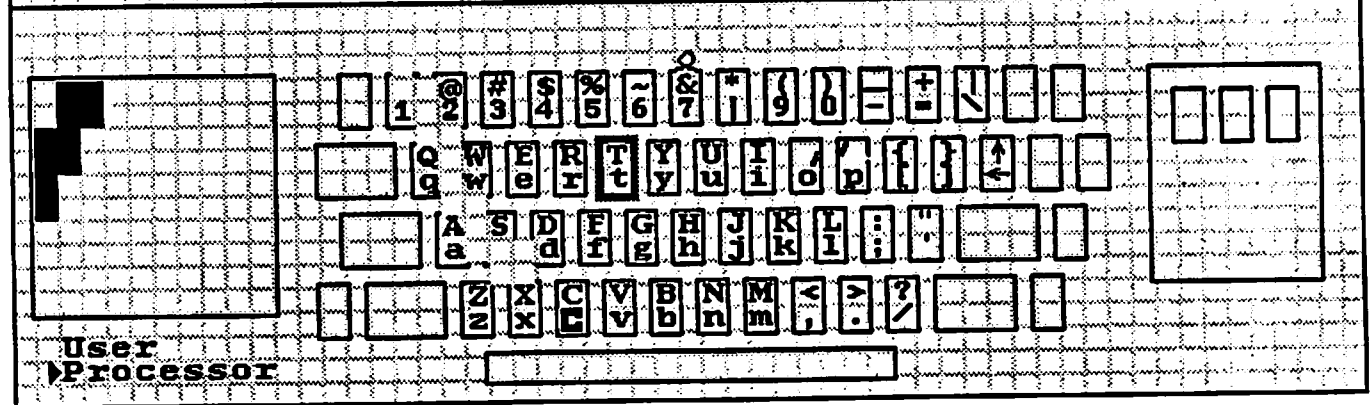
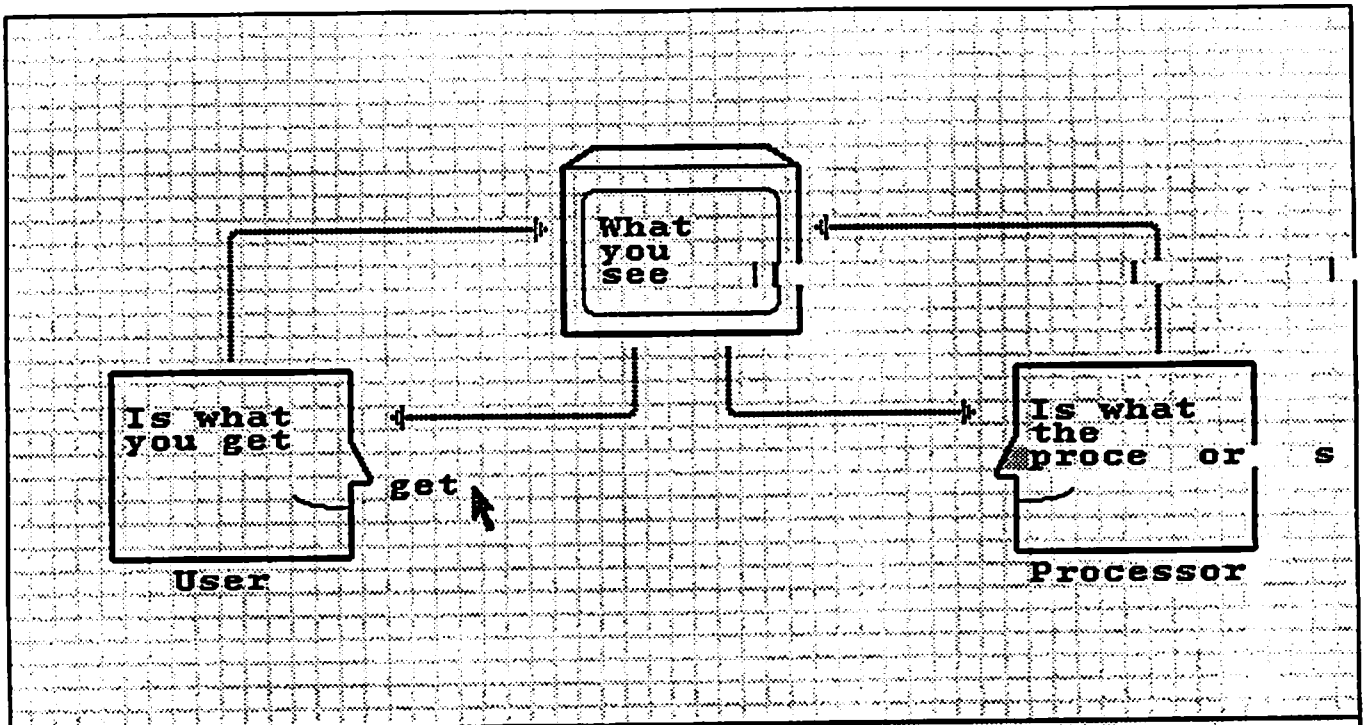


To make the next example more dramatic, I've copied the two cells that make up the bridge of the processor's nose onto the *I* and *O* keys.

Typing works by overtyping rather than by inserting. Let's move the cursor to the beginning of the line and retype *computer gets* as *processor gets*.



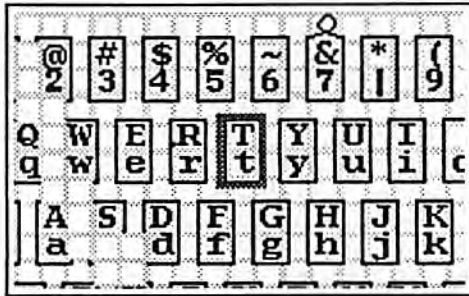
We have just typed *processor ge* on top of *computer ge* and are about to type *t* on top of a previously copied *s*. So far so good.



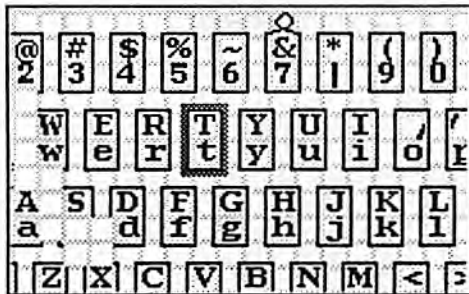
Typing the *t* causes a surprising word wrap.

Since the previously copied *s* is not currently on the keyboard, it is treated as a noncharacter and therefore causes word wrap.

Since the cells that make up the bridge of the processor's nose are currently on the keyboard, they are treated as characters, not as left margin. The new left margin is at the user's nose, which is not currently on the keyboard.

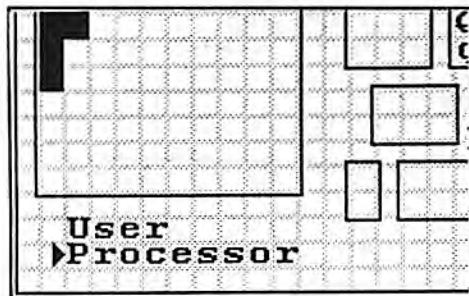


So far we have ignored the details of user/processor dialog. Let's look at the last action in slow motion. First the user presses the *t* key.

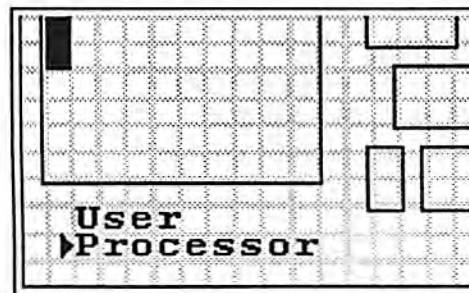


The input processor recognizes the action and responds by drawing a key highlight and key trigger around the image of the *t* key.

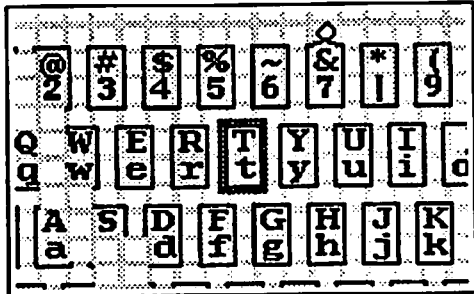
The input processor is traditionally considered part of the computer. For our purposes, the input processor is considered a part of a "user agent", a human/computer entity that includes the user, the input processor, and the output processor. The input processor is entirely independent of the internal "central" processor.



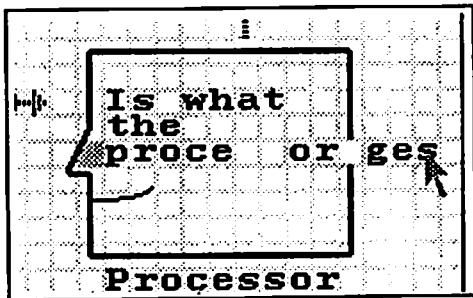
The user agent erases the triangle next to its name and redraws the triangle next to the processor's name. The triangle is called the "interlock". The interlock is the mechanism for synchronizing user and processor actions.



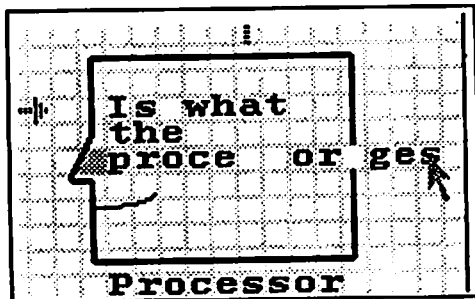
Seeing the interlock next to its name, the processor agent wakes up.



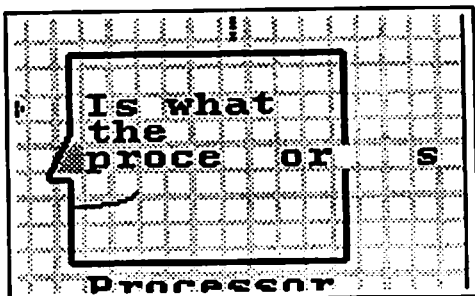
The processor searches the screen for a key trigger and finds a trigger on the image of the *t* key. The processor assumes that the user will draw a trigger around only one key at a time. If the processor finds no trigger, it assumes that the last user action was a mouse move and searches for the new cursor position.



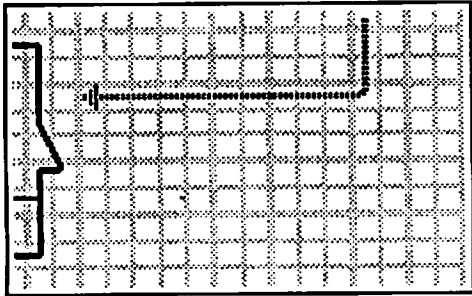
The processor searches the red plane for a shape that matches the cursor. From the cursor, it deduces the location of the current cell, which it checks against the keyboard to see whether it is a character.



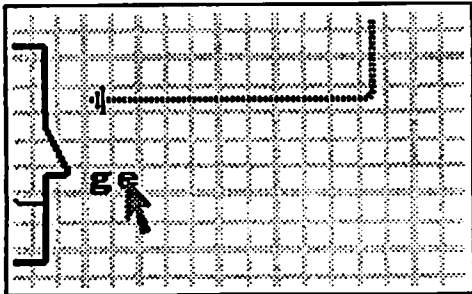
If the current cell does not match any character on the keyboard, then the processor must see if word wrap is required. Word wrap occurs when: (1) the current cell is a noncharacter, (2) the previous character is a character other than a space, and (3) the string of adjacent characters preceding the current cell includes a space character. If condition 2 or 3 is not met, then the newly typed character is placed at the beginning of the next line without wrapping any characters. This is called "character wrap".



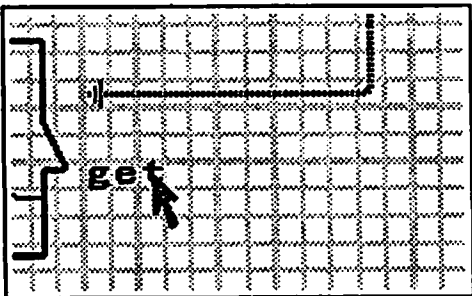
The string of characters before the current cell and after the previous space character is called the "wrapped word". When word wrap occurs, the processor first erases the wrapped word from original position.



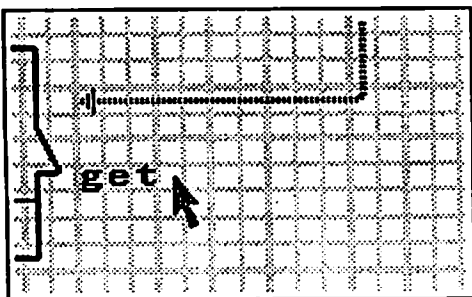
Then, the processor scans for the beginning of the next line. Here is the exact procedure. First, the processor scans left to find the first noncharacter.



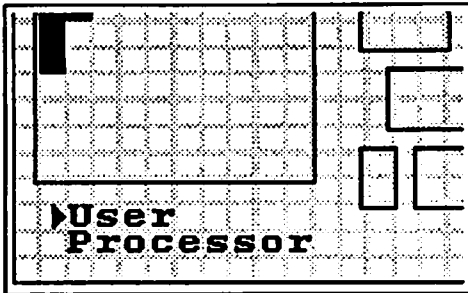
Next, the processor retypes the wrapped word one character at a time starting at the new left margin. Typing proceeds exactly as if the user had done the typing, except that keys are not highlighted. If the new line is not long enough for the wrapped word, character wrap will occur.



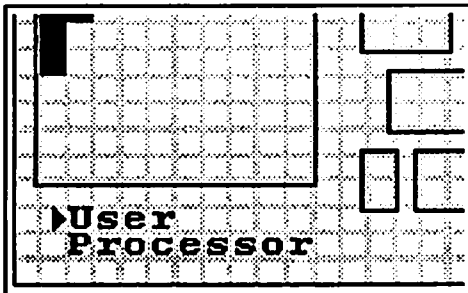
After wrapping the wrapped word, the processor draws the newly typed character.



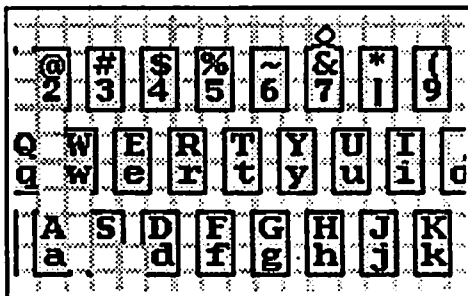
Finally, the processor redraws the cursor at the new position. Notice that both user and processor can redraw the cursor. This strategy makes sense only for a relative positioning device, such as a mouse, since an absolute positioning device, such as a tablet, must maintain a fixed relation between pen position and screen position.



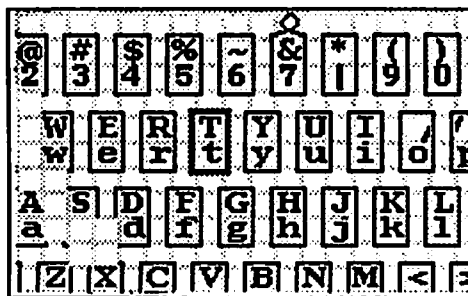
The processor erases the interlock next to its name and redraws the interlock next to the user's name, handing control back to the user agent. Which interlock belongs to which agent is determined solely through position. The labels are strictly for the benefit of the (human) user.



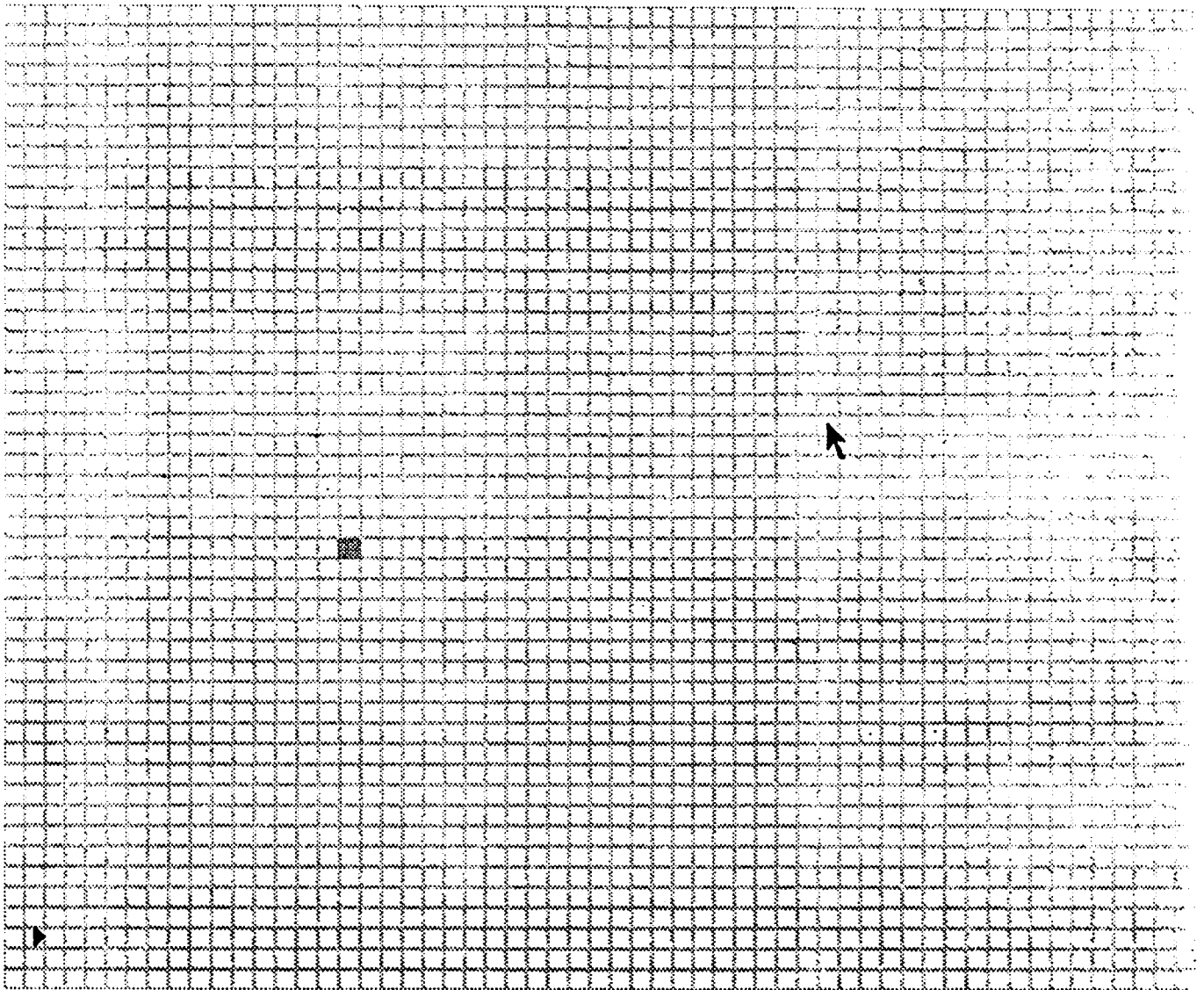
Seeing the interlock next to its name, the user agent wakes up and waits for the user to act.



If the user acts next by releasing the *t* key, then the user agent responds by erasing the key highlight to show that the key is now up, but leaving the the key trigger to show that the key just went up. The key trigger allows the processor to detect that a key just went up.

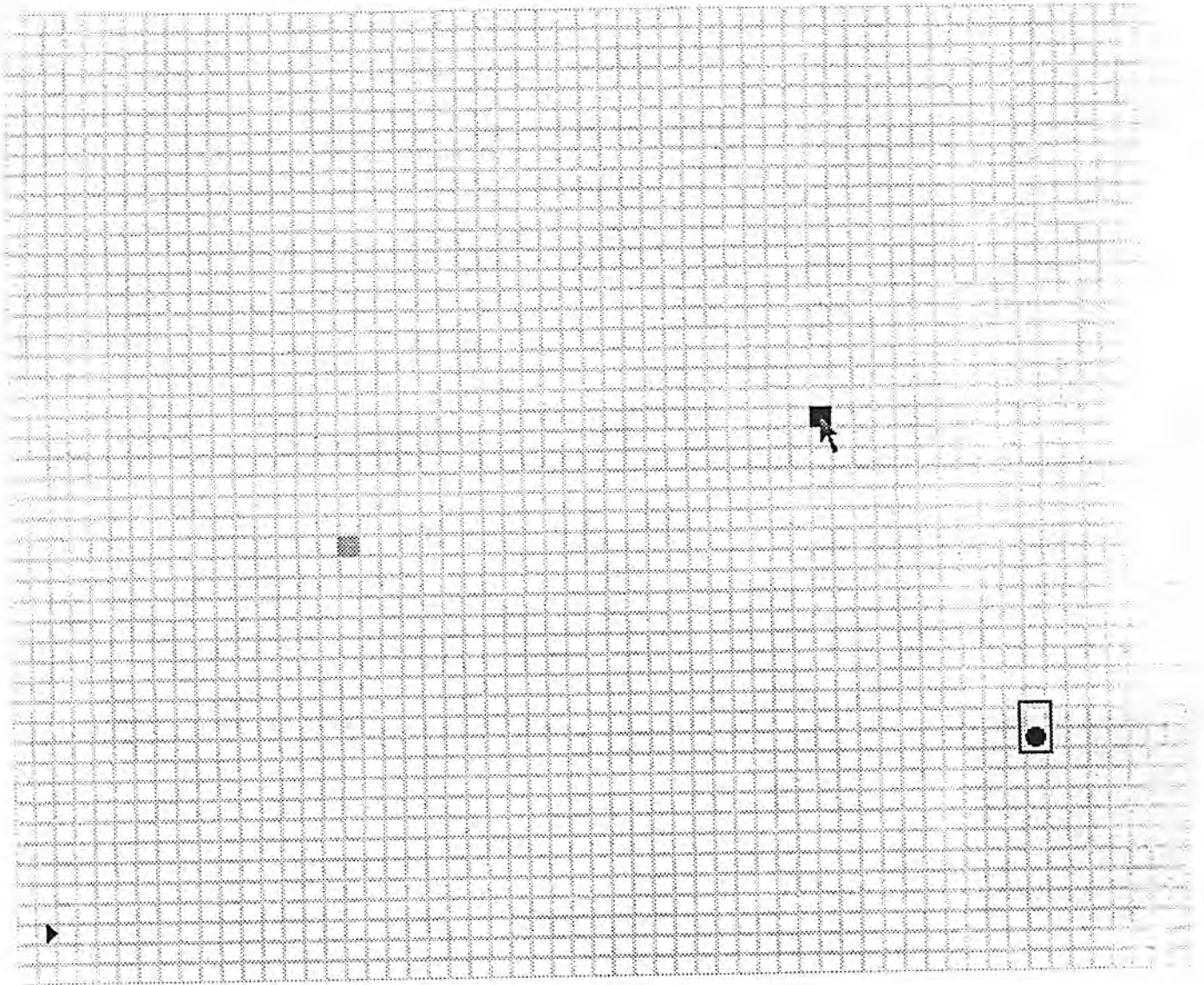


If the user acts next by moving the mouse, then the user agent responds by erasing the key trigger to show that the *t* no longer just went down, erasing the old cursor, and redrawing the new cursor. The absence of a key trigger alerts the processor that the last action was a mouse move.

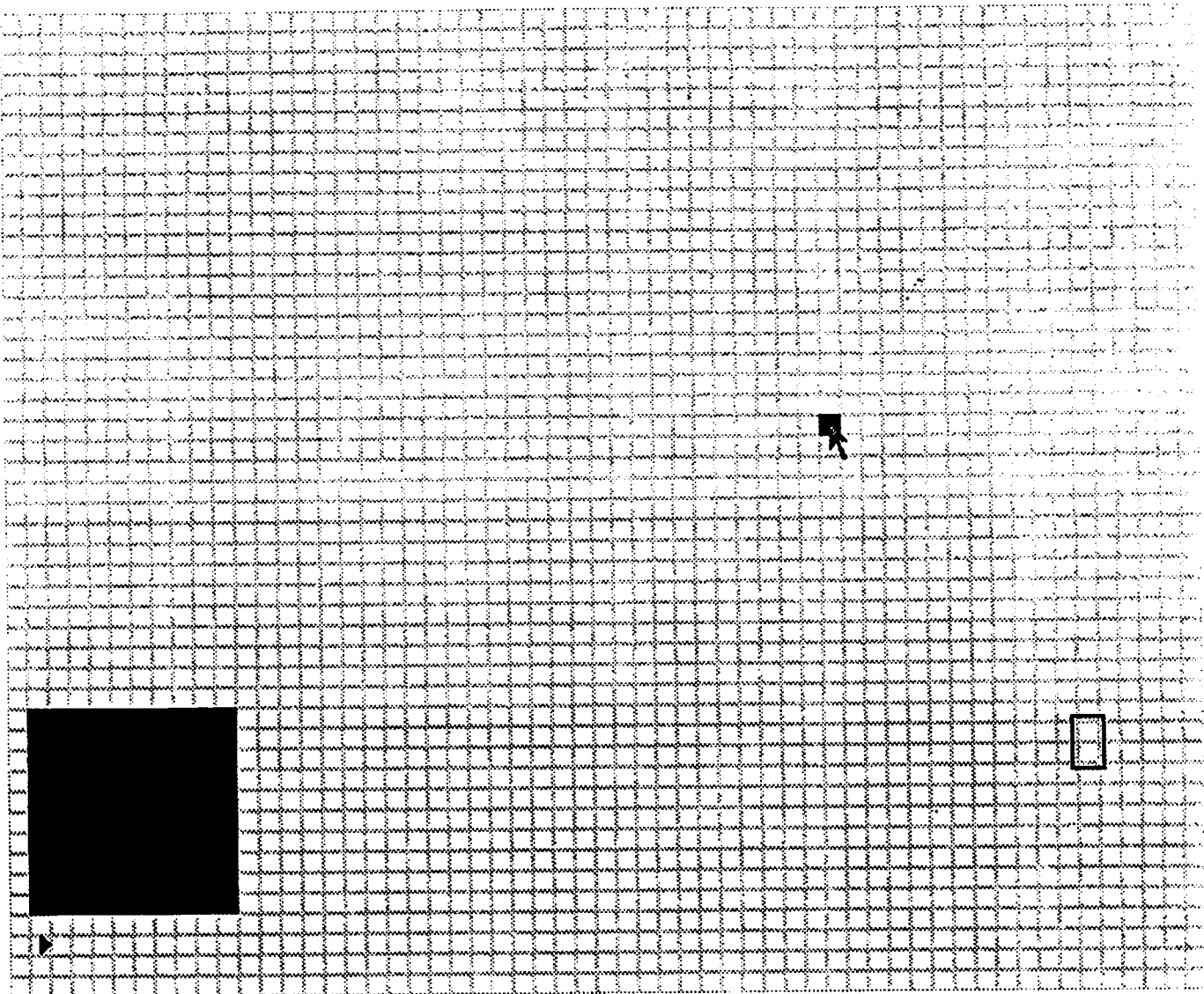


**The visual boot.** When I first built Viewpoint, I started with an empty screen: only a grid, a cursor, a selection, and an interlock. How did I build up the rest of the screen? Since the behavior of Viewpoint depends on the content of the frame buffer, and the content of the frame buffer depends on the user's actions, drawing the first screen is a tangled process similar to bootstrapping. Here is one way to perform a "visual boot".

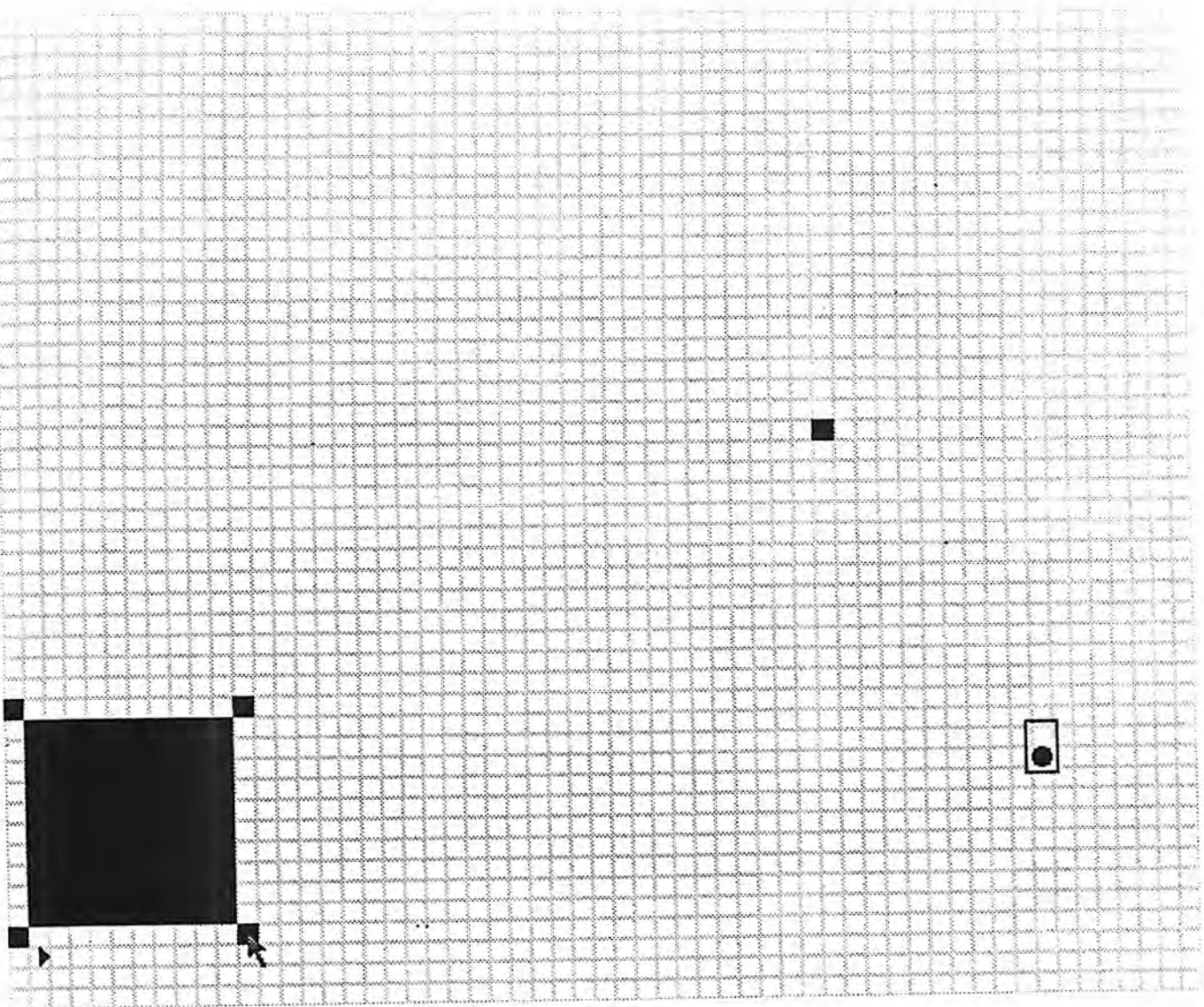




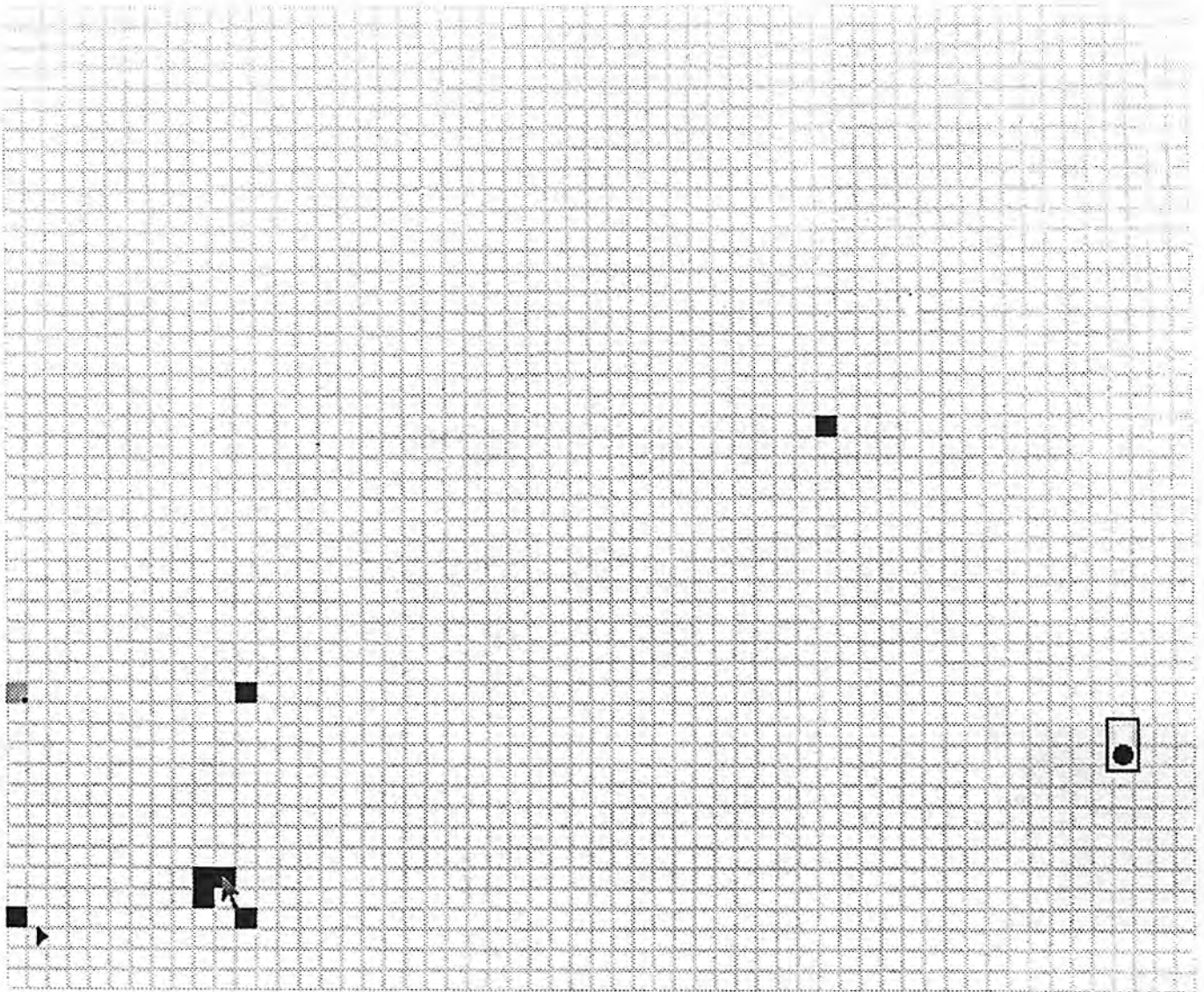
Start by drawing a cell black. Notice that the user agent remembers where key highlights should be drawn, even though the key images are not visible. Like the interlock, key images are assigned fixed positions.



Select the cell, revealing the location of the puffbox.



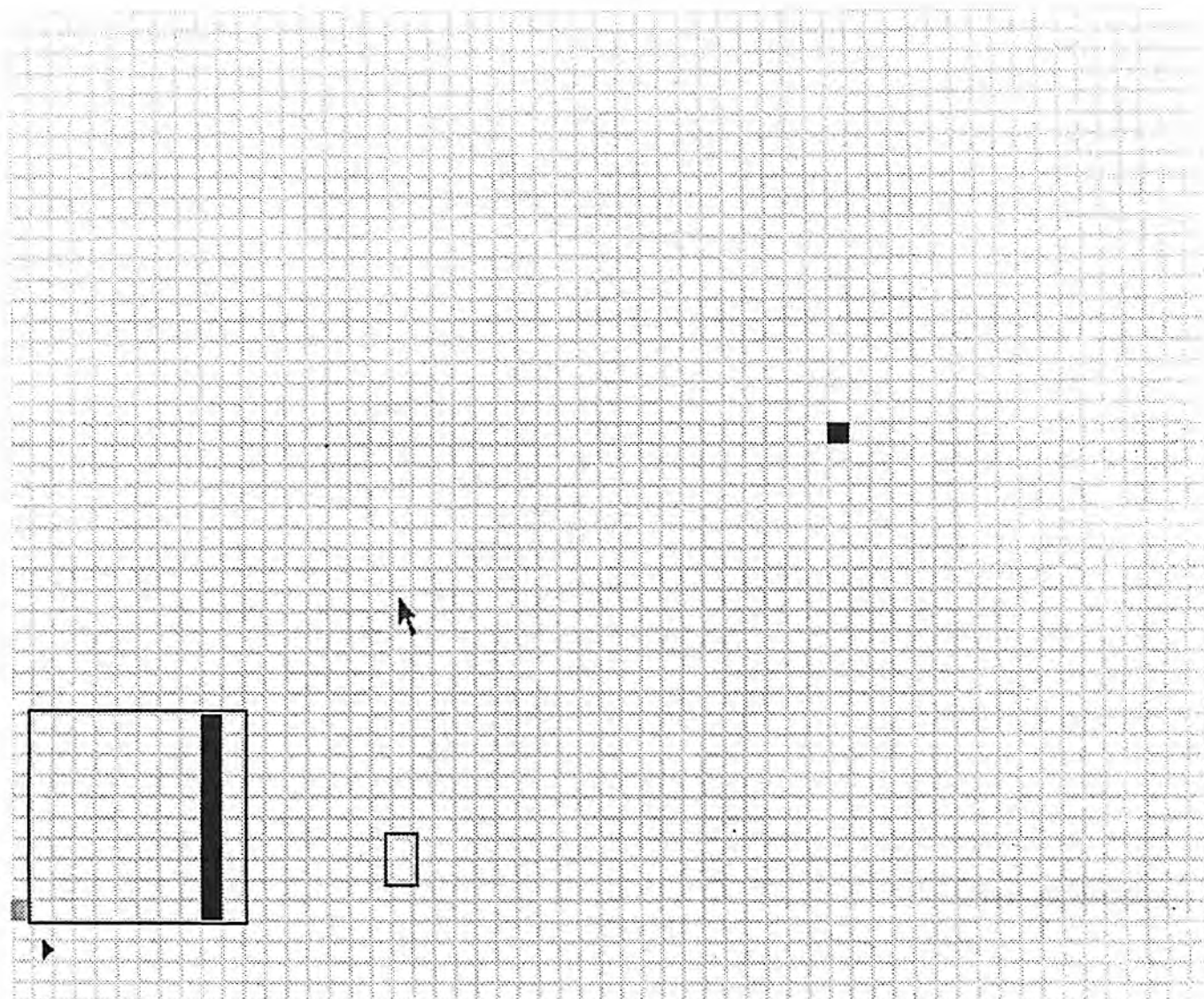
Mark the corners of the puffbox so we can remember its position even when we've selected a white cell.



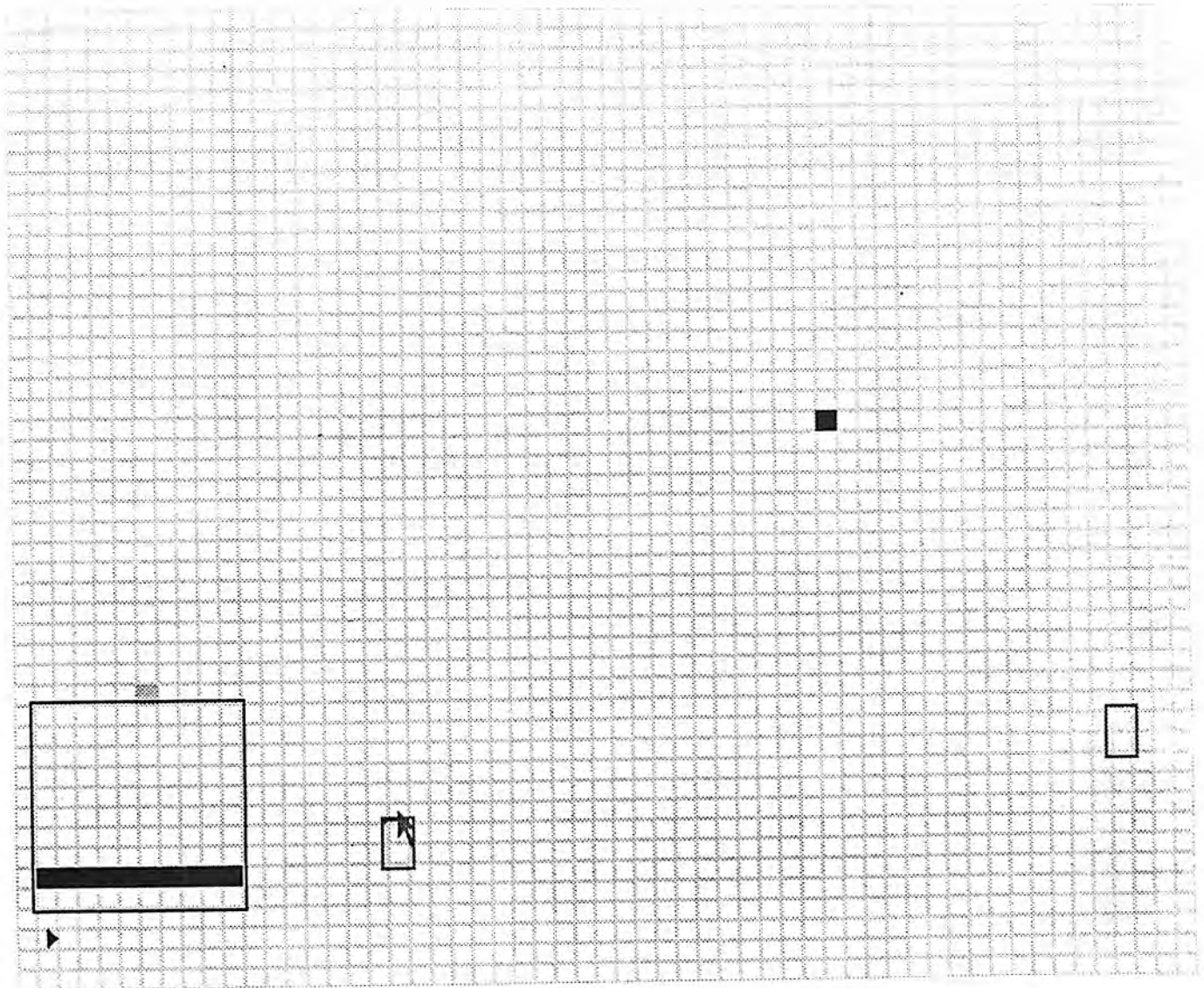
Select a corner of the puffbox and reshape it by drawing in the puffbox.





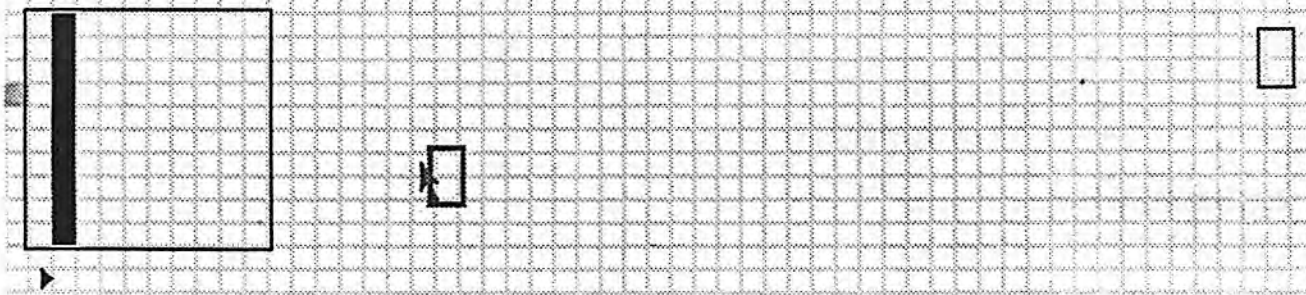


Move the cursor to an empty spot and press the A key to locate the position of the corresponding key image. Again, the user agent remembers where to draw the highlight, even though the keyboard is not visible in the frame buffer.



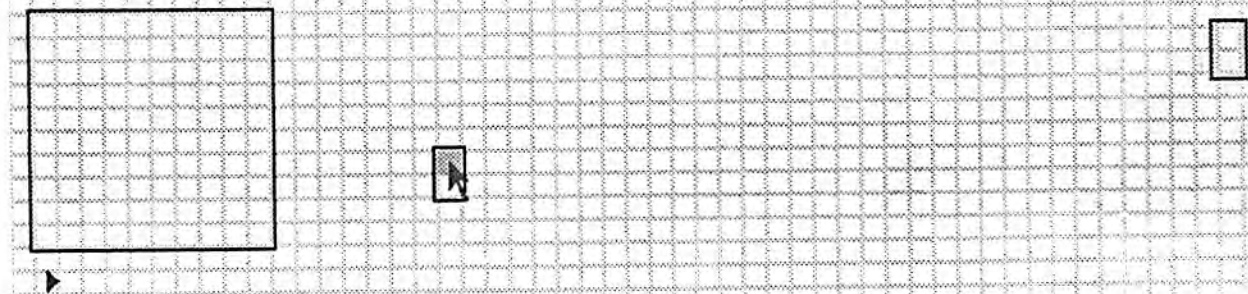
Continue to hold the A key. Copy the border of the puffbox to the location of the A highlight to make a border for the A key image.



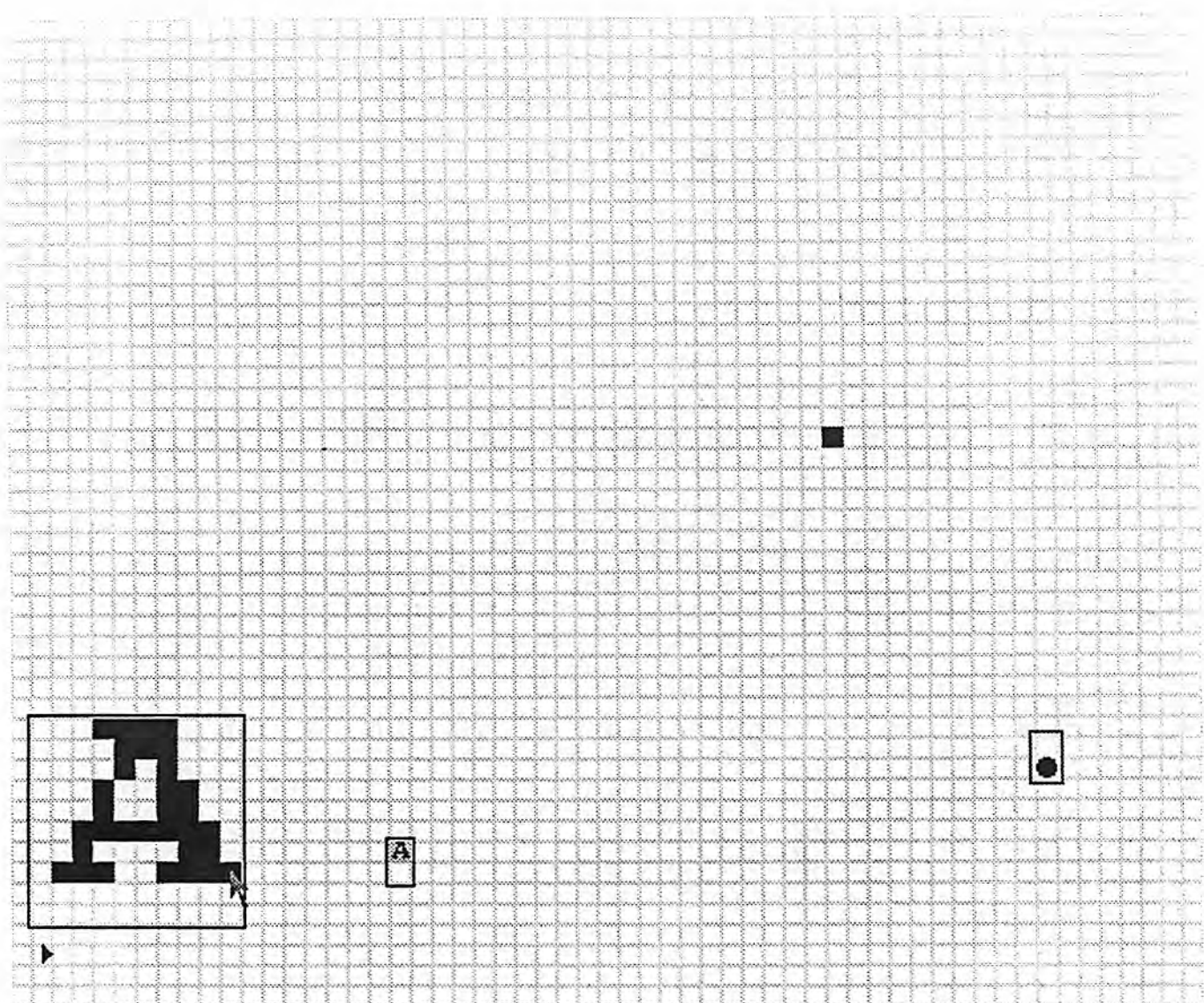


Release the A key and finish drawing the key border. Notice that we can draw boxes of many different sizes by this method. Viewpoint users gradually learn to see all images as palettes of square patterns for copying.

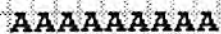




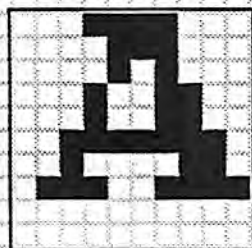
Select the upper half of the *A* key image. The upper half of the key image corresponds to uppercase *A*, the lower half with lowercase *a*. As with key borders, the locations of shifted and unshifted key images are remembered separately from the frame buffer, but this time by the processor agent, not the user agent.



Draw an *A* by drawing in the puffbox.



AAAAAAAA

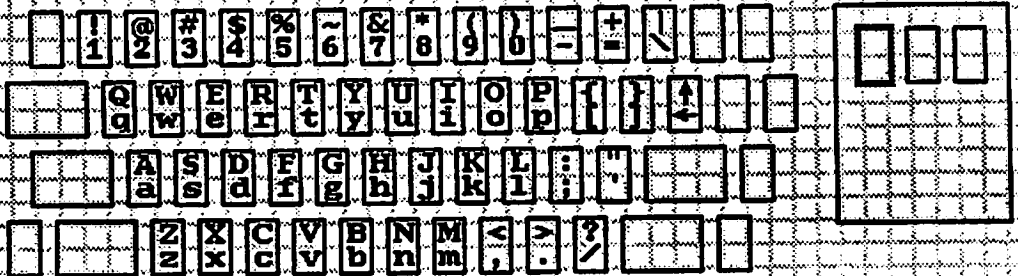
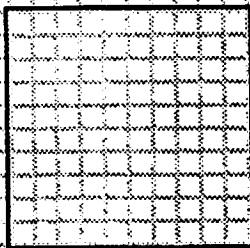
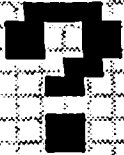


Move the cursor to a blank spot. Hold down the Shift key and type the A key to test out your letter.

Here is the complete set of cells required for building up the rest of the keyboard. Notice that edges of two or three adjacent keys often fall in the same cell. This means that we cannot simply copy the edges of the puffbox to make the borders of the rest of the keys without erasing parts of other keys. For instance, drawing the border of the S key by the same method would erase the right edge of the A key.

# WHAT HAPPENS IF WE

type right off the edge of the scr



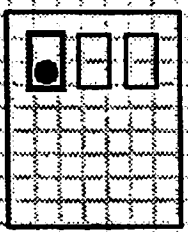
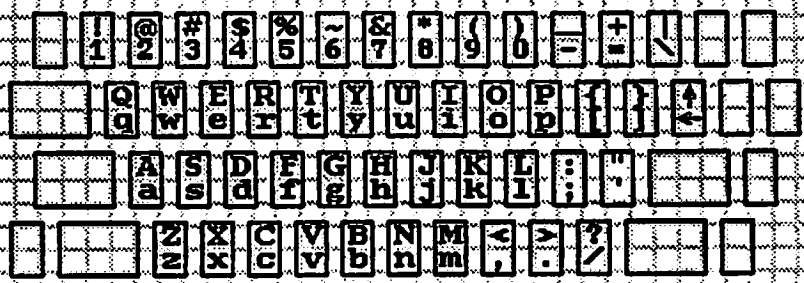
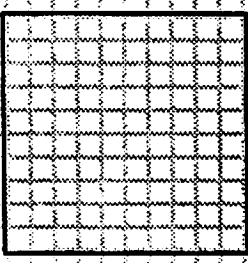
User Processor

What happens if...? Now, we will explore edges of the system where rules threaten to break down. What happens if we type off the edge of the screen? The cursor simply wraps to the beginning of the next line. When typing reaches the bottom of the screen, it wraps back to the top. If the cursor were to fall off the edge of the screen, then the cursor position would be undefined.



scr  
een  
?  
**WHAT  
HAPPENS  
IF WE**


type right off the edge of the  
draw on a black & white cell?

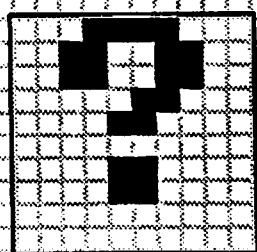


User  
Processor

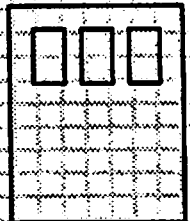
What happens if we draw on a cell that contains both black and white pixels? One possibility would be for the cell to be complemented: White pixels turn black and black pixels turn white. I have chosen a different rule: Any cell that contains at least one white pixel is considered a white cell and therefore turned all black.

screen?  
**WHAT HAPPENS IF WE**

type right off the edge of the  
 draw on a black ■ white cell?  
 draw or type over the selection? 



	1	2	3	4	5	6	7	8	9	0	=	+	N	
	Q	W	E	R	T	Y	U	I	O	P	[	]	^	
	A	S	D	F	G	H	J	K	L	:	"			
		Z	X	C	V	B	N	M	<	>	?			



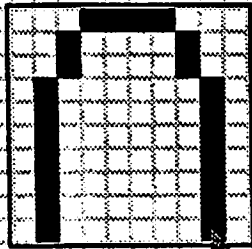
►User Processor



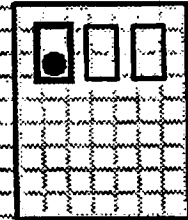
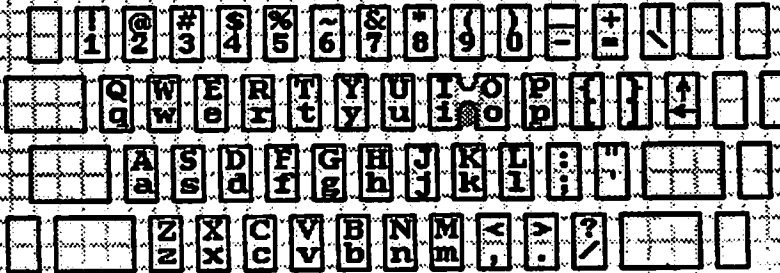
What happens if we draw over the selection? The selection is mapped back to the puffbox at the end of every action; consequently, the puffbox always reflects the current state of the selection. Another possibility would be for the puffbox to be updated only when the middle mouse button was pressed.

scr  
een  
?  
**WHAT  
HAPPENS  
IF WE**

type right off the edge of the  
draw on a black ■ white cell?  
draw or type over the selection?  
draw over a key border?



User  
Processor



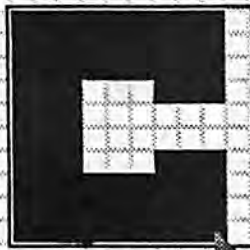
What happens if we draw over a key border? Nothing special happens—the key borders, like the interlock labels, are strictly for the benefit of the user. The agents do not rely on the key borders in order to locate the keys. Another possibility would be for the agents to look for the key borders in order to determine the locations of the keys. Redrawing the key borders would relocate the key images.



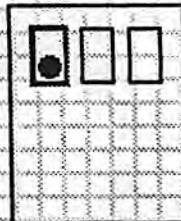
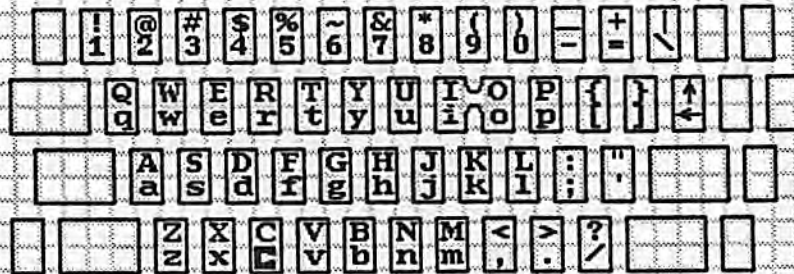
scr  
een  
? **WHAT  
HAPPENS  
IF WE**



type right off the edge of the  
draw on a black ■ white cell?  
draw or type over the selection?  
draw over a key border?  
draw over a character image?



►User  
Processor

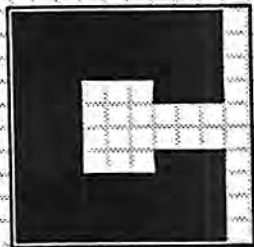


What happens if we draw over a key image? As we have seen, the change does not affect previously typed characters but does affect subsequently typed characters. Another possibility would be for changes to a key image to be reflected immediately in all cells with the same content. This would be especially appropriate if we were using Viewpoint as a font editor.

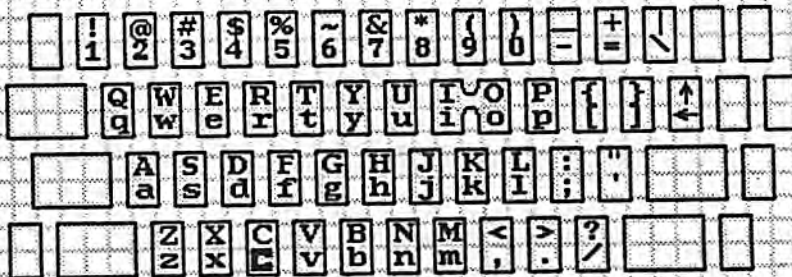
screen?  
**WHAT HAPPENS IF WE**



- type right off the edge of the
- draw on a black ■ white cell?
- draw or type over the selection?
- draw over a key border?
- draw over a character image?
- draw over the draw Color?



▶User Processor

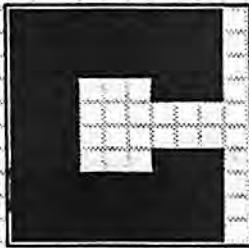


What happens if we draw over the draw color? Since the processor agent determines the current draw color by checking the draw color cell, we might expect that changing the draw color to something other than a solid or hollow circle might confuse the processor. In fact, the processor determines the current draw color by checking only a single pixel in the middle of the draw color cell. Drawing a black cell over the draw color cell can occur only when the draw color was already a black (solid) circle, and drawing a white cell over the draw color cell can occur only when the draw color was already a white (hollow) circle. Consequently, drawing over the draw color causes no anomolous behavior.

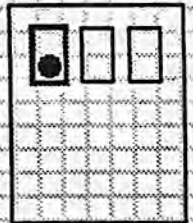
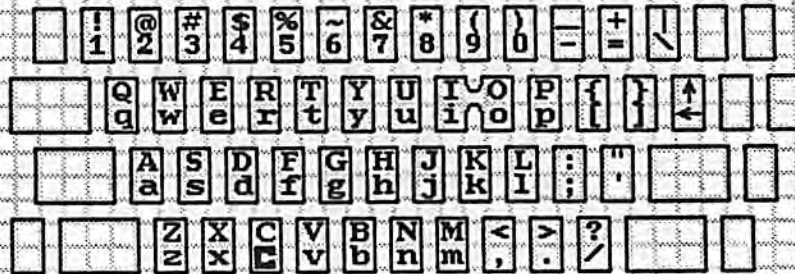
scr  
een  
?  
**WHAT  
HAPPENS  
IF WE**



- type right off the edge of the
- draw on a black ■ white cell?
- draw or type over the selection?
- draw over a key border?
- draw over a character image?
- draw over the draw **Color**?
- draw over the interlock?



User  
Processor



What happens if we draw over the interlock? Since agents determine who has control by checking the interlock, we might expect that changing the interlock to something other than a triangle might cause deadlock, with both agents waiting for their interlock to appear. In fact, agents redraw both interlocks at the end of every cycle, so the effect of drawing on top of an interlock cell is immediately nullified.



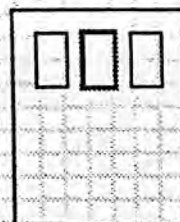
scr  
een  
?  
**WHAT  
HAPPENS  
IF WE**



- type right off the edge of the
- draw on a black white cell?
- draw or type over the selection?
- draw over a key border?
- draw over a character image?
- draw over the draw Color?
- draw over the interlock?
- Copy a Cell into the puffbox?



▶User  
Processor

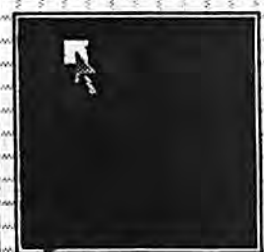


What happens if we copy a cell into the puffbox? Since black cells in the puffbox correspond to black pixels in the selection, and white cells in the puffbox correspond to white pixels in the selection, we might expect a mixed cell to cause some confusion. Let's find out what actually happens. Select a black cell.

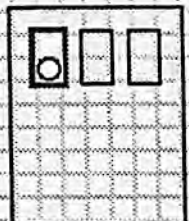
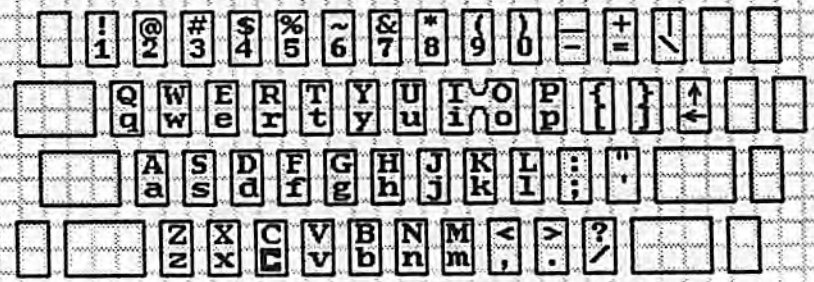
scr  
een  
? **WHAT  
HAPPENS  
IF WE**



- type right off the edge of the
- draw on a black ■ white cell?
- draw or type over the selection?
- draw over a key border?
- draw over a character image?
- draw over the draw **Color**?
- draw over the interlo**ck**?
- Copy a Cell into the puffbox?**

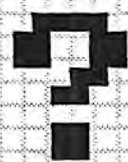


►User  
Processor



Erase one pixel in the selection by drawing white on one cell in the puffbox. (The white pixel may not be visible if your copy of this dissertation is the result of multiple photocopying.)

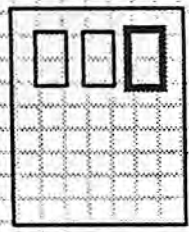
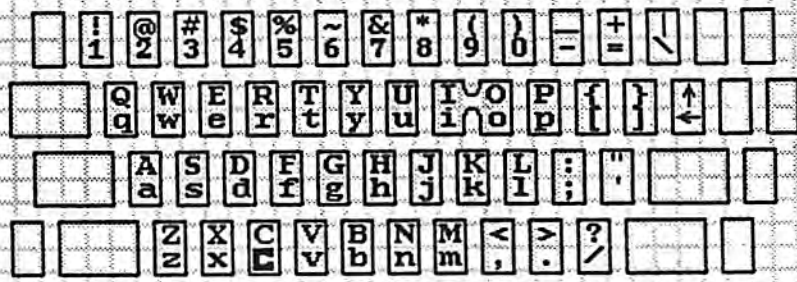
screen?  
**WHAT HAPPENS IF WE**



- type right off the edge of the
- draw on a black ■ white cell?
- draw or type over the selection?
- draw over a key border?
- draw over a character image?
- draw over the draw **C**olor?
- draw over the interlock?
- C**opy a **C**ell into the puffbox?



User Processor



Copy the selected cell, which contains both black and white pixels, into the puffbox by pressing the right mouse button. The rule is that a mixed cell in the puffbox corresponds to a white pixel in the selection. Thus the copy action causes another pixel in the selection to turn white. The selection now contains two white pixels.



scr  
een  
?

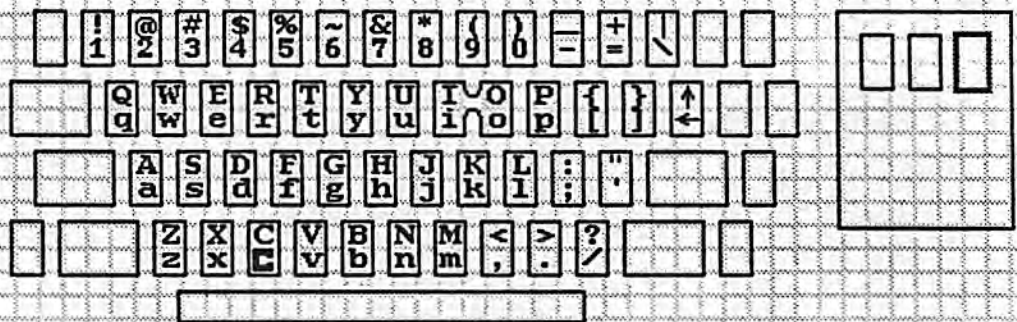
# WHAT HAPPENS IF WE



- type right off the edge of the
- draw on a black ■ white cell?
- draw or type over the selection?
- draw over a key border?
- draw over a character image?
- draw over the draw **C**olor?
- draw over the interlo**C**k?
- C**opy a **C**ell into the puffbox?



User Processor



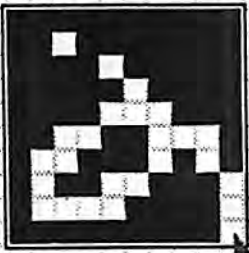
Before the processor has completed its turn, however, it must map the selection back into the puffbox. The new white pixel in the selection causes the corresponding cell in the puffbox to turn white. Thus a cell can never stay mixed in the puffbox; it is always turned completely black or completely white



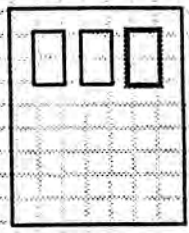
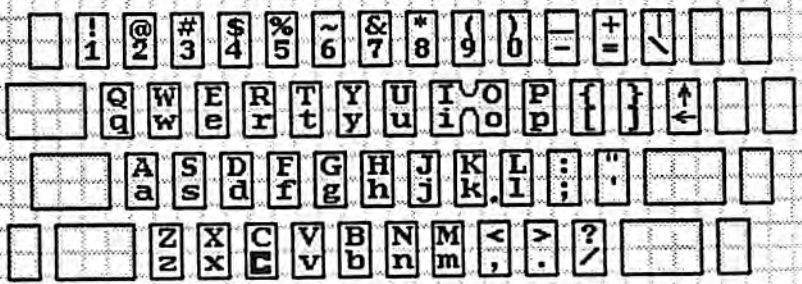
scr  
een  
?  
**WHAT  
HAPPENS  
IF WE**



- type right off the edge of the
- draw on a black  white cell?
- draw or type over the selection?
- draw over a key border?
- draw over a character image?
- draw over the draw  color?
- draw over the interlock ?
- Copy a  cell into the puffbox?



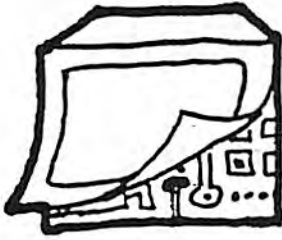
▶User Processor



▶User Processor

If you continue copying the selected cell into different locations in the puffbox by moving the mouse, you will leave a trail of white pixels in the selection and a corresponding trail of white cells in the puffbox.

If Viewpoint did not map the selection back into the puffbox at the end of every operation, then the trail in the puffbox would appear as shown to the left. The trail records the history of the selected cell as more and more pixels turned white. Notice the recursive relation between the cells in the puffbox and the puffbox as a whole.



## 4 Implementation

The chapter describes the decisions I made while implementing Viewpoint—what I did, why I did it that way, and how else it might have been done.

- Purpose: visibility
- Behavior: text, font, and graphics editing
- Objects & actions: draw, select, copy, type
- Encoding: color, position, shape, structure
- Cursor
- Key highlights and key triggers
- Selection
- Puffbox
- Font
- Text
- Ink color
- Interlock
- Grid
- Inner workings

## **Purpose: visibility**

What is the program for?

**Decisions.** The purpose of the Viewpoint program is to study visibility. My priorities for the program were, first, to satisfy visibility, second, to satisfy the stricter condition of input visibility, third to communicate clearly to the user (comprehensibility), and fourth, to be able to perform interesting tasks (functionality).

**Reasons.** I chose to study visibility because it is a key requirement for a computer that supports visual thinking. I pursued visibility in its strictest form to discover its limits.

I emphasized the theoretical goal of visibility over the practical goals of comprehensibility and functionality to keep the focus on principles. If my main goal had been ease of use or speed of execution, theoretical principles would have been swamped by practical compromises.

I used comprehensibility to balance visibility. After I had designed an object to be logically deducible by the computer, I tested it to make sure that it was visually comprehensible to the user. Many objects, such as the cursor and key highlights, went through many graphic revisions. On the other hand, I did not try to be thorough; graphic design is only intended to be adequate.

I adjusted the level of functionality freely to keep it challenging, but not unmanageable. For instance, I chose a form of word wrap that was simple to implement and rich enough to illustrate interesting issues. In conventional software design, functionality is usually the driving force.

I studied visibility by building an actual program rather than by imagining an abstract theory because user interface ideas must be experienced to be understood. Writing a program forced me to make my ideas precise. It also allowed me to explore many variations on visibility and to experience the consequences of each variation.

**Alternatives.** Visibility uses the frame buffer as the primary representation of system state. Other possible primary representations include a string of characters as in a text editor, a tree as in Lisp, an array as in the programming language APL, or the state of input devices.

Viewpoint encodes the entire state of the system in a single screenful of information. Alternatively, I could allow a frame buffer larger than the screen, only part of which would be visible at any one time.

Other ways to study study visibility besides building a program include analyzing existing software, exploring many visualizations of a single object, or finding ways to visualize difficult concepts.

## **Behavior: graphics, text and font editing**

What does the program do?

**Decisions.** Viewpoint behaves as a pixel-oriented graphics editor, an overtyping text editor, and a fixed-width font editor. An editor allows objects of a certain sort to be created and modified conveniently. A graphics editor acts on arrays of pixels. A text editor acts on strings of characters. A font editor acts on arrays of pixels associated with characters.

**Reasons.** I originally wanted Viewpoint to behave as a programming language. I implemented an editor instead because I realized that from the user's point of view, editing is more basic than programming: To program you must first use an editor. As it turned out, implementing an editor presented more than enough challenges to keep me busy.

Graphics editing is the foundation of Viewpoint because it is the most general type of editing in a frame buffer. Once I had built a graphics editor, I was able to use Viewpoint itself to mock up future versions of itself.

I added text editing because integrating text and graphics presents interesting difficulties. Most so-called integrated systems treat text and graphics as distinct forms of data. Viewpoint takes a different approach, treating text as a kind of graphics.

I added font editing in response to the need for a font to use with the text editor. At first, I considered creating the font by writing programs. Then I realized it would be easier and more in keeping with the spirit of the project to use the graphics editor I had already built. After all, a font is just a special kind of graphics.

**Alternatives.** The behavior of Viewpoint is neither as simple nor as powerful as it could be.

On the one hand a simpler system would be better for exploring definitions of fundamental user interface concepts. Even simpler than editing is direct action. In Viewpoint, direct action appears in the representation of user input. It took me many years to accurately portray the state of the input devices. Viewpoint contains many such microexperiments. Each could be a project in itself.

On the other hand, a more powerful system would be better for studying conventional computer science issues. I originally wanted Viewpoint to be able to handle complex graphic structures, such as polygons and spline curves, which would raise parsing issues similar to those in compiler design. I also thought about a pixel-based programming language, which would raise issues of machine architecture.

## **Objects & actions: draw, select, copy, type**

What objects exist in the system and how does the user manipulate them?

**Decisions.** Objects include the cursor, key highlights, key triggers, selection, puffbox, font, text, ink color, and grid. Actions include drawing, selecting, copying, and typing. Each action is associated with a different key or set of keys: drawing with the left mouse button, selecting with the middle mouse button, copying with the right mouse button, and typing with the keyboard keys.

**Reasons.** Actions implement the desired behaviors of graphics, text, and font editing. For ease of implementation, as well as ease of use, all actions are essentially modeless; any action may be performed at any time, even during the middle of another action.

The details of each action were worked out to demonstrate a range of problems: Drawing demonstrates modes, selecting demonstrates parallel representations, copying demonstrates multiplication of effort, and typing demonstrates pixel parsing. Together, the actions demonstrate the equivalence of drawing and typing: Typing a character has the same effect as copying a keyboard cell.

Drawing acts on cells rather than pixels because individual pixels are too small to see accurately. The one-button draw command, borrowed from MacPaint, was chosen for simplicity and because it required a mode.

Objects were designed to go with the actions. Details are described in the rest of this chapter. I left some objects, such as the key borders, so they could easily be accidentally erased to make the point that everything on the screen is just pixels. In a practical system, I would probably protect such objects.

**Alternatives.** The three-button mouse proved frustrating, as I expected. Users frequently hit the wrong button, even after extended use. I tried a one-button mouse plus held-down keyboard keys, but this proved even more confusing. Other possibilities include adding a tool palette and changing cursor shape to show state.

The draw command could be replaced by the copy command plus two fixed cells, one all white and one all black.

The copy command could cause the selection to move in parallel with the cursor. This would allow large areas to be copied easily, even though the selection covered only a single cell.

## Encoding: color, position, shape, structure

How are objects encoded visually so they can be unambiguously recognized?

**Decisions.** Objects are encoded first by color (look only at certain values), second by position (look only at certain locations), third by shape (match against a particular pixel pattern), and fourth by structure (parse the frame buffer to extract information). For instance, the ink color is recognized by filtering out all but the black plane, looking at pixels inside the image of the left mouse button, and comparing them against a solid and a hollow circle.

Every object is one of four colors: red, green, black, or blue. Colors are arranged in parallel planes in the frame buffer. Overlapping colors on the screen mix to give the illusion of transparency. Positions align with a grid of ten by ten-pixel cells, delimited by blue lines. The *contents* of a cell refers to the black pixels in the cell.

**Reasons.** Objects demonstrate a range of encoding techniques from simple (color) to complex (structure).

Color planes allow objects to overlap while remaining visibly distinct. This ability is essential for objects, such as the cursor, that use position to convey information. Color mixtures were chosen to give the illusion of transparency. Simple additive or subtractive color mixing yielded colors that were not distinct, so I adjusted intensity to increase contrast and biased hue toward the plane with higher priority: red first, green second, black third, and blue last.

Cells simplify the parsing problem, especially for text, since object boundaries are never in question. Square cells allow cell proportions in the puffbox to mimic pixel proportions in the selection. The need for color and square pixels helped determine my choice of the Cedar programming environment.

**Alternatives.** Each object in Viewpoint can be shown many different ways. For instance, keyboard state could be shown as a list of key names, and the cursor could be shown as crosshairs or numerical coordinates.

We could limit Viewpoint to black and white or a smaller screen. Alternatively, we could add new attributes such as animation, interactivity (wiggle the mouse to find the cursor), and alternate input devices (different devices for different objects). Finally, we could add graphic objects that require complex visual parsing to be understood.

Changing the application changes which encoding techniques are appropriate. For instance, a cursor in a color painting system cannot be uniquely identified by color, so must use another attribute, such as blinking.

## Cursor

How is the current cell visually indicated?

**Decisions.** A red arrow of fixed shape indicates the cell containing the red pixel at the tip of the arrow. The position of the cursor is updated by input from the mouse, which specifies a change in  $x$  and  $y$  coordinates. Cursor position is clipped to the screen area so that at least one pixel of the cursor remains on the screen at all times. There is exactly one cursor on the screen at all times.

**Reasons.** Red makes the cursor stand out. Incremental updating allows processes other than the mouse to be able to change the cursor position. (The alternative would be to position the cursor absolutely on the screen based on an absolute accumulated  $x,y$  position.) Positioning the cursor to the nearest pixel rather than the nearest cell makes the cursor move smoothly and predictably. Clipping the cursor keeps it visible.

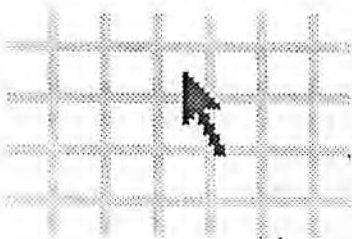
**Alternatives.** I tried many cursor shapes before I found one that was visually satisfying. My first attempt was a hollow red square just inside the edge of the current cell. Users found it difficult to predict when the box would suddenly jump to the next grid position. At this stage, I had not yet added the blue grid.

Next, I added a continuously moving arrow to the jumpy box. To visually separate the arrow and the box, I erased a one-pixel-wide border around the arrow. Users found the combination of motions distracting.

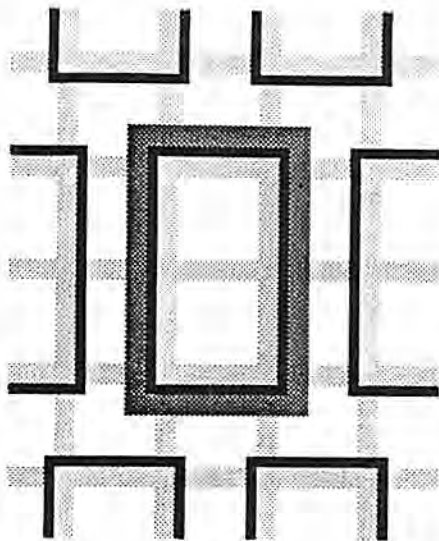
I also experimented with a user-definable cursor: The shape of the cursor was defined by the contents of the cell in the lower left corner of the screen. Watching the cursor change as it redrew the cursor shape in the puffbox was an interesting experience in self-reference. Since the cursor could be blank, however, it violated visibility.

Finally, I took away the moving box and added a static blue grid. Users liked this solution. The continuously moving cursor seemed a more lifelike companion to the continuously moving mouse.

I had problems programming the cursor. The Cedar programming environment reported mouse input in absolute  $x,y$  coordinates. Relative coordinates were not easily available. The mouse position could be changed by a procedure call from a program, but this would fight with mouse inputs initiated before the call but processed after the call. Viewpoint users had to slow down to avoid synchronization problems. Conventional editors solve this problem by dividing the cursor into a mouse-controlled arrow and a keyboard-controlled insertion point.







## Key highlights and key triggers

How is the current state of all keys indicated?

**Decisions.** Each key corresponds to a fixed rectangular group of cells on the screen (the "key image"). The edge of each key image is delimited by a one-pixel-thick hollow rectangle ("halo") of black pixels (the "key border"). As long as a key is held down, a red halo around the corresponding key border is turned on ("the key highlight"). When a key has just been pressed or released, a red halo around the key highlight is turned on (the "key trigger"). Highlights stay on while the key is down; triggers stay on only for the instant a key is pressed or released.

**Reasons.** Key highlights and key triggers report the state of the keyboard in full generality. All keys are treated uniformly. This allows subsequent procedures to interpret key input without arbitrary restrictions; e.g. any key may be used as a Shift key, even a mouse button.

In static screen illustrations, key highlights and triggers are not as visually prominent as they might be; they are designed to work best in animation. Red was chosen to match the cursor color—all user input is red.

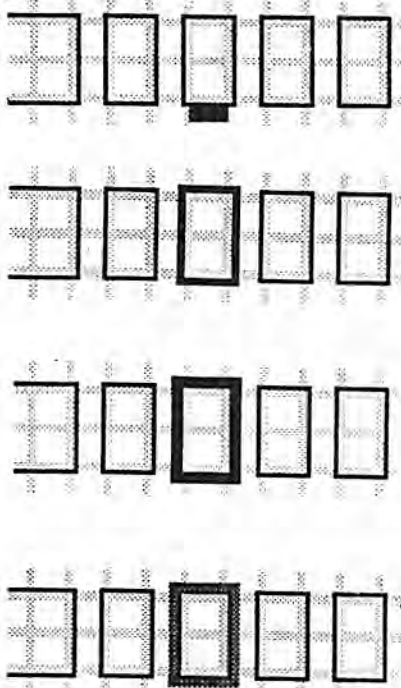
**Alternatives.** I tried many variations before I found a visual representation that captured key state in full generality. My mistakes show how I misunderstood the nature of the keyboard as an input device.

At first, I reported key presses by hanging a black rectangular "flag" from the bottom of the last key pressed. When the next key was pressed, the previous flag was erased. This did not show multiple key presses correctly.

Changing the flags to highlights that stayed on as long as the key was down made the keyboard picture feel much more "live". In addition, halos looked better. Halos gave the impression that the existing key borders were being visually emphasized, while flags appeared to be extra visual elements that bounced around from key to key.

I later realized that the key highlight did not show whether the key had just been pressed or whether it had been down for a while. This information is needed by commands, such as typing or drawing, that act differently at the start or end of a key press. To show whether a key had just been pressed or released, I added key triggers.

I changed the highlights and triggers from black to red to avoid ambiguity: A black highlight can be erased, resulting in a picture that no longer reports keyboard state correctly. I left key borders black as a taste of the confusion. Since the locations of key images are fixed, erasing a key border does not confuse the processor.



## Selection

How is a selected cell indicated?

**Decision.** A solid green square indicates the cell it fills.

**Reasons.** I chose a green square to avoid visual confusion with the red arrow. Since the selection does not move as often as the cursor, I was able to round the selection position to the nearest cell.

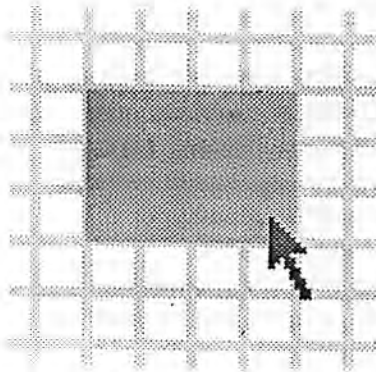
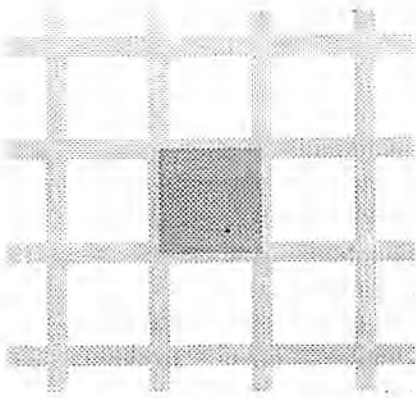
**Alternatives.** The highlight was originally yellow, inspired by yellow highlighting pens. Unfortunately, users had trouble spotting a yellow selection a white screen, especially when the contents of the highlight were empty. The contrast between yellow and white was just too low.

Viewpoint would have been difficult to implement on a black and white display. Black and white interfaces often use black-white reversal to indicate a selection. For a reversed selection to stand out clearly, all objects must be predominantly the same color (black or white) in their normal unselected state, as is true in text. In a graphics editor such as Viewpoint, any combination of pixels may appear in a picture, so reversed selection is ambiguous.

One of the most obvious limitations of Viewpoint is that the selection covers only one cell. This makes copying large regions tedious. One solution is "dragging" as used in the Apple Desktop Interface [Apple]: To select a rectangular region, press and hold the Select button at one corner of the selection, move the cursor to the opposite corner of the selection while continuing to hold the button (the selection rectangle will stretch to fit between the initial selection point and current cursor), and release the mouse button.

As conventionally implemented, dragging would violate visibility since there is no visual indication of which corner of the selection was the initial anchor point. Normally, the anchor will be the corner opposite the corner nearest the cursor, but potentially, the cursor can leap from any position to any other position in a single bound. This ambiguity is not much of a problem for the user since the mode lasts only for a moment, is reinforced by having to hold down a button, and is quickly disambiguated by watching the motion of the selection.

One way to make dragging satisfy visibility is to visually mark the initial anchor point. For instance, the spreadsheet program Excel visually distinguishes the first cell when the user selects a rectangle of cells by dragging [Microsoft]. Another solution is to require that dragging always proceed from the upper left corner toward the lower right, so the anchor point is always the upper left corner.



## Puffbox

How can the user edit individual pixels given that the cursor indicates cells, not pixels?

**Decisions.** The puffbox, a ten-cell by ten-cell region of the screen in a fixed location, shows a magnified view of the contents of the current selection, in which each cell in the puffbox corresponds to a pixel in the selection. Black cells correspond to black pixels and white cells correspond to white pixels. Drawing in the puffbox changes the corresponding pixels in the selection. Changing the contents of the selection changes the corresponding cells in the puffbox.

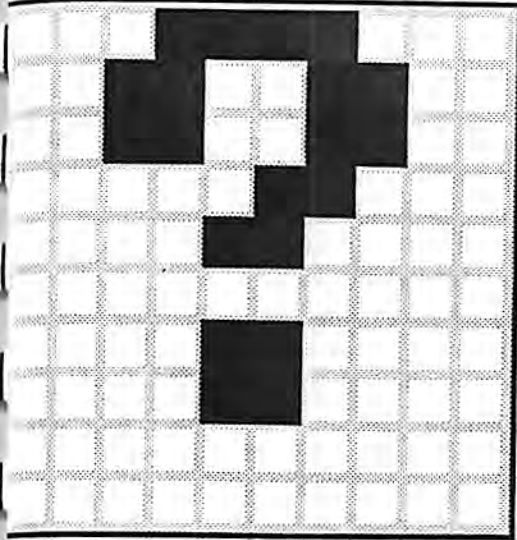
**Reasons.** The ability to edit pixels is needed for font editing. Pixels are too small to be edited at actual scale. The puffbox allows pixel editing without introducing a visual mode, which would violate visibility. To assure that the puffbox and selection always match, the contents of the selection are mapped back to the puffbox at the end of every action.

**Alternatives.** Most drawing programs include a close-up editing mode in which the entire screen changes scale. I avoided such modes since they violate visibility.

Since the puffbox shows only a single cell, users found it difficult to align parts of a drawing that spanned more than one cell. One solution would be to enlarge the puffbox to display neighboring rows of pixels in adjacent cells.

As with key borders, the puffbox border is easy to erase accidentally. This forces users to interrupt their work to patch up the puffbox border. One solution would be to forbid the cells in the puffbox border to be changed.

The puffbox shows an enlarged view of a smaller portion of the page. If Viewpoint were designed to handle multiple pages, a "shrinkbox" might show a reduced view of a ten by ten collection of 100 pages. Selecting a cell in the shrinkbox would switch the screen to the appropriate page.



## Font

How is the current font shown?

**Decisions.** The font is shown as part of a picture of the keyboard that appears on the screen. Each key image that corresponds to a printing character is composed of exactly two cells. Key images that do not correspond to printing keys may be any size. The upper cell shows the image of the appropriate uppercase character; the lower cell shows the image of the appropriate lowercase character. Every character image occupies exactly one cell.

The one exception is the space character. The Spacebar appears as a rectangle many cells wide and only one cell high, to match the the shape of the actual spacebar. The left-most cell in the space key image defines the current space character image. There is no separate uppercase space character.

**Reasons.** Showing the font as part of the keyboard image was a natural consequence of the decision to show key state in terms of a picture of the keyboard. Once I had built the graphic editing portion of Viewpoint, the easiest way to define the font was to use Viewpoint itself to draw the character images, rather than program the bitmaps in the Cedar programming language. The keyboard image was inspired by active keyboard diagrams on the Xerox Star and Apple Macintosh.

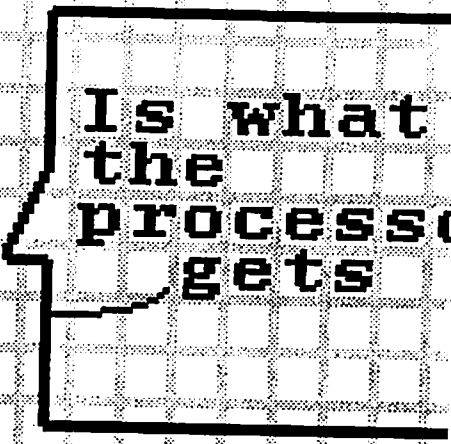
**Alternatives.** The simplest alternative would be not to show the font at all. If the font were fixed, this would not violate visibility. Another alternative would be to separate the font and the keyboard images. Perhaps the font would appear twice, once in keyboard order and once in ASCII order. Editing either representation would automatically update the other.

Going the other direction, we could merge the keyboard image and the actual keyboard by turning the screen into a touch screen and throwing out the keyboard. Or we could add little displays to the top of each keytop and throw out the display.

Showing the font on the keyboard revealed to me that conventional keyboards are labeled inconsistently—letter keys show only uppercase letters whereas number and symbol cases show both lower and uppercase forms.

Other variations on the font include characters that do not have to be aligned with the grid, variable pitch fonts, multiple font styles, multiple fonts, and rotated fonts. With each new font variation, the problem of parsing characters becomes harder.

## Text



Is what  
the  
process  
gets

What forms of structured graphics require pixel parsing to recognize their structure?

**Decisions.** Text is the representative form of structured graphics in Viewpoint. The structure of text is revealed by word wrap, which occurs when the user types a word that will not fit on the current line. To wrap text requires pixel parsing to recognize the beginning and end of the current text line, the beginning of the next text line, and breaks between words.

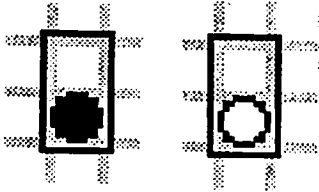
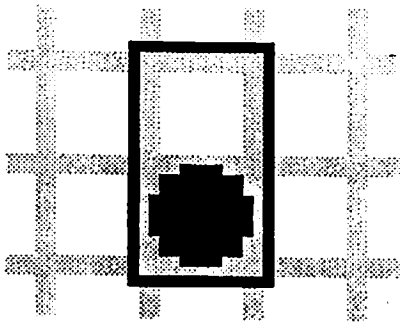
**Reasons.** All objects in Viewpoint require pixel parsing to be recognized. Some objects, however, are easier to recognize than others. Key highlights, key triggers, the puffbox, font, interlock, and ink color require only table lookup with a bit of filtering. The cursor and selection require only simple search.

Word wrap was included in Viewpoint to demonstrate nontrivial pixel parsing. Recognizing line endings and word breaks requires scanning cells and comparing them against the current font. By redefining the font, the user can vary the interpretation of a line of text.

Since full word wrap is rather complicated, I chose the form of word wrap that was simplest to implement. Word wrap in Viewpoint works only in the forward direction. Forward word wrap would not be practical in a real system, but it is adequate to raise issues.

**Alternatives.** I originally wanted to show margins, tab settings and other structural information in a separate plane. The "structure" plane would be light blue, by analogy with layout grids used in publication design. Only the image plane would be directly editable. To edit the structure plane, the user would first interchange the image and structure planes. This interchange would show clearly that graphics can serve either as data to be edited or as structure to alter the interpretation of data. In the current system, the image plane serves as data, and the cursor and selection planes serve as structure.

An obvious next step would be to implement full word wrap. Other text operations that require pixel parsing include dynamic font editing (when a character is edited, all instances change simultaneously), printing (which requires recognizing which character is which), searching, and compiling. Beyond text are structured graphic objects such as lines, rectangles, connected regions, and curves.



## Ink color

How is the color currently being drawn by the Draw command indicated?

**Decisions.** When the Draw button is first pressed, a special symbol called the “ink color” is drawn inside the Draw button image to remember which color is being drawn. If the Draw command is drawing black, the ink color looks like a solid circle; if the Draw command is drawing white, the ink color looks like a hollow circle.

When the Draw button is still being held down but has not just been pressed, the processor agent checks a particular pixel near the middle of the ink color to determine which color to draw.

**Reasons.** The ink color is positioned inside the key associated with drawing. The ink color represents color information as itself.

As with the interlock, making the ink color black raises the danger that it will be overwritten by a draw or copy operation. The ink color solves the self-reference problem by making sure that overwriting it never alters its meaning: A white ink color can be overwritten only by a white cell and a black ink color can be overwritten only by a black cell. Since the Draw command checks only a single pixel near the center of the ink color, overwriting the cell never changes its meaning.

The hollow circle was chosen to be simple and distinct from other objects. For instance, a solid black or white cell would have been simpler, but easily confused with a cell drawn by the Draw command. In general, I chose object shapes that could be differentiated even without color, since I found that people find it hard to associate meaning with color alone.

**Alternatives.** As with key triggers, it took many months to realize that I needed to display the ink color.

The other likely place to display the ink color is the cursor. In general, the cursor is a good place to display mode information since it is always at the center of the user’s visual field. I decided against this operation because the cursor was red and I wanted to represent color literally.



# ▶ User Processor

## Interlock

How is the current agent controlling the screen indicated?

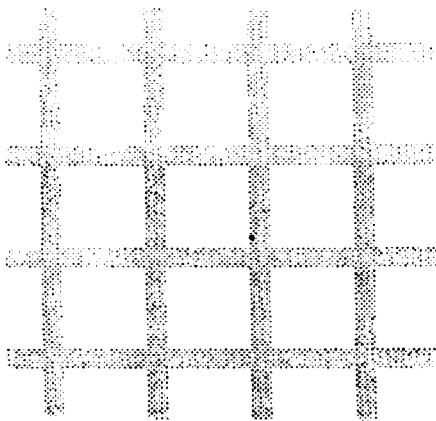
**Decisions.** A triangle in the lower left corner of the screen called the "interlock" jumps back and forth between two cells to indicate one of two states. At all times, one of the two cells will be blank and the other will contain the triangle. The names of the two agents—"user" and "processor"—appear to the right of the two interlock cells.

**Reasons.** Making the interlock black raises the danger that it will be overwritten by a Draw or Copy operation. If for some reason the interlock is erased, the system will freeze since neither the user nor processor can proceed. Changing the interlock to another color, as I did with the cursor and selection, would avoid the problem altogether.

I chose to make the interlock black to demonstrate a monochromatic solution to the disambiguation problem. At the end of every action Viewpoint rewrites the black planes of both interlock cells. Thus if the user attempts to draw a black square on top of one of the interlock cells, it is immediately overwritten by either a blank cell or a triangle before the offending black cell can cause confusion. In effect, a region of the image plane is declared off-limits to editing operations. So instead of a reserving a color for special interpretation, I have reserved a region.

The appearance of the interlock was chosen for two reasons: The triangle occupies only a single cell and is thus minimally distracting when it bounces back and forth, and the presence of both names clearly shows the range of possible states even when only one is active.

**Alternatives.** The interlock is similar to semaphores used in concurrent programming to synchronize parallel processes [Brinch Hansen]. Alternatives to the interlock include mechanisms similar to other process synchronization methods. In particular, reserved regions are similar to windows. In a typical multi-tasking window environment, collisions are avoided by allowing only one process at a time to control a given window. Windows are related to modes in that the interpretation of user action varies according to which window the cursor is in. As Larry Tesler says, "a window is a mode in sheep's clothing." [Tesler 1981]



## Grid

How are cell boundaries indicated?

**Decisions.** Cell boundaries are shown by light blue lines along the border of each cell. The grid never changes; it is strictly for the benefit of the user.

**Reasons.** The grid is needed only by the user; the computer locates cell boundaries by counting pixels. If Viewpoint did not display the grid, the system would still be technically visible. I added the grid to satisfy comprehensibility.

The grid was designed to complement the cursor. Together, the red arrow cursor and the blue grid indicate the current cell.

I chose the light blue color because it is visually unobtrusive. Graph paper often uses a similar color for the same reason.

**Alternatives.** Since the grid is constant it could easily be moved out of computer memory and onto a physical overlay on the display surface.

If the grid had a moveable origin, then the computer would need to be able to see the grid in memory. Shifting the grid would have drastic effects on the definition of the font. Similarly, a grid with changeable cell size would have a drastic effect on the puffbox.

By giving the cursor, selection, image, and grid each its own plane, I realized that the roles are potentially interchangeable. The cursor need not be small nor must it cohere as a single rigidly moving object. In general, a cursor, a selection or a grid may be any image at all.

Here's a somewhat absurd example: Imagine that the cursor is an entire page of text and the image is a single arrow in the middle of the screen. Moving the cursor slides the text over the arrow. Typing a key causes the appearance of the cursor to change at the point marked by the image.

## Inner workings

How are objects and actions represented internally?

**Decisions.** Externally, Viewpoint behaves as if the frame buffer is the sole representation of the system state. Internally, Viewpoint uses auxiliary variables to cache many aspects of system state.

The following objects are cached: the current key action (down or up), the key that just went up or down (if any), the state of all keys (down or up), the current and previous cursor positions, the ink color, and the current and previous selection position.

**Reasons.** Ideally, Viewpoint would never use any auxiliary variables. Unfortunately, current computer architectures do not support fast operations on large groups of pixels. Consequently, when I implemented portions of Viewpoint without caching, the system was prohibitively slow.

Redrawing the cursor, for instance, would ideally be implemented by locating the cursor, erasing it, then drawing the new cursor. Unfortunately locating the cursor in the red plane is prohibitively slow. Caching the previous cursor position allows Viewpoint to operate more efficiently since the program does not have to parse the screen to infer cursor position.

I have allowed cached variables because they do not compromise the external behavior of the system. Every cached variable either can be parsed from the screen or is logically unnecessary. The key and cursor variables repeat information present in the red plane. The previous cursor and selection position variables accelerate erasing old cursor and selection before drawing the new. The ink color variable accelerates inferring the current ink color during a Draw operation.

The other implementation compromise has already been mentioned in connection with the cursor. Viewpoint is implemented as if mouse input were reported in  $x,y$  increments rather than absolute  $x,y$  coordinates. The illusion works as long as the user doesn't type fast.

**Alternatives.** Cached variables can also be used to accelerate text operations: For every cell, Viewpoint could cache the name of the character whose image matches its current contents. Since the current implementation of Viewpoint often bypasses the frame buffer as the primary representation of state, there is always the risk I have missed something. Now that I have found faster algorithms, I suspect that the entire system could be implemented purely without sacrificing performance.



## **5 Theory**

**To verify that Viewpoint satisfies visibility,  
I needed a formal theory of interactive systems.  
Since none existed, I built my own.**

- The need for formal definitions**
- A model of interactive systems**
- Visibility**
- Modeling Viewpoint**
- Verifying visibility**

## The need for formal theory

Computers have been formalized; human-computer interaction has not. Books on user interface design offer metaphors and slogans, but few definitions [Heckel]. Metaphors are useful, but without formal definitions, the field of user interface design lacks a foundation for systematic research.

Consider modes. A mode is an invisible program state that changes the interpretation of user actions. For instance some text editors have overwrite and insert modes. Suppose the cursor is on the *l* in *modl*. If the editor is in overwrite mode, typing an *e* changes *modl* to *mode*. If the editor is in insert mode, typing an *e* changes *modl* to *model*. To predict the outcome, the user must keep track of the current mode.

Larry Tesler compiled a definition of "mode" by conducting an informal survey at Xerox PARC.

A mode of an interactive computer system is a state of the user interface that lasts for a period of time, is not associated with any particular object, and has no role other than to place an interpretation on operator input [Tesler 1981].

Tesler's definition is often quoted as the authoritative definition of "mode". Even so, it fails cursory inspection, since the terms "interactive computer system", "state", "user interface", "period of time", "associated", "object", "interpretation", and "operator input" remain undefined.

Imprecise definitions make for futile conversations. Are modes bad? The question is meaningless until people agree on the definitions of "mode".

Inconsistent definitions lead to inconsistent design. Suppose a system is required to be modeless. One way to avoid modes is to require that the user hold down a special key to indicate the current state. Some people would call the hold-down key a mode. Others would disagree. Unless everyone agrees on what a mode is, the system may be implemented inconsistently.

Finally, imprecise definitions lead to sloppy thinking. The opposite of modality is modelessness. Modelessness is often characterized as "the same action always causes the same result." But this is *not* the logical opposite of modality. Between modality and modelessness is a third possibility: In a system that behaves randomly, the same action does not always cause the same result, but the link between action and result is not governed by a mode. Thus modelessness is poorly named.

To build a formal model of human-computer interaction, it is helpful to look at existing models of computers, humans interacting with computers, and interactive systems.

The best known formal model of computers is the Turing machine [Manna]. The Turing machine reduces the computer to a tape on which are written symbols, a tape head that can read or write from the tape, and a program that tells the tape head which symbols to write and which way to move. The Turing machine was developed for verifying the computability of mathematical functions. Turing machines are closed systems that exclude input and output and therefore don't model interactive systems.

Models of interactive systems first arise in the verification of concurrently executed programs [Hoare]. In such systems, interaction occurs between program modules, not necessarily between the program and the user. Models of interaction also arise in the use of temporal logic to analyze reactive systems such as process controllers and text editors. Both concurrent programming and temporal logic models of computation are concerned with program correctness and are therefore inappropriate for studying human-computer interaction issues.

Models of the psychology of humans interacting with computers have been developed by Norman [Norman] and by Card, Moran, and Newell [Card]. Such models do not account for the computer's side of the interaction.

William Newman has developed a rough model of the structure of computers interacting with humans [Newman]. Gene Ciccarelli has developed a more formal model of user interfaces that carefully distinguishes the data base describing the data itself and the data base describing the visual presentation of the data [Ciccarelli]. Both models are useful but assume too many of the particulars of current computers to serve as clean theoretical foundations.

In mechanical engineering, control theory models formal properties of tightly coupled human-machine systems, such as a person balancing a stick on one finger. Control theory defines formal properties such as controllability and observability but applies only to continuous mechanical systems modeled by differential equations, not discrete symbolic systems modeled by algorithms.



## A model of interactive systems

To give Viewpoint a theoretical foundation, I built a formal model of interactive computational systems. The model focuses exclusively on processor state changes and does not attempt to describe user psychology.

A *location*  $l$  is an undefined term. Locations model storage cells in a computer, including both memory cells and registers. A set of locations is notated as  $L$ .

A *value*  $v$  is an undefined term. Values model the possible contents of a memory location, for instance, 0 and 1. A set of values is notated as  $V$ .

A state  $s: L \rightarrow V$  is defined to be a function that associates a value  $v$  in  $V$  for each location  $l$  in  $L$ . State models the instantaneous contents of all memory locations in a computer. A set of states is notated as  $S$ .


An *input*  $i$  is an undefined term. Inputs model possible user actions such as key presses or mouse moves. A set of inputs is notated as  $I$ .

An *action*  $a: S \times L \rightarrow L$  is an ordered set that associates a state and an input with another state, where both states are defined over the same domain and range. A set of actions is notated as  $A$ .

A *machine*  $m$  is an ordered set  $[L, V, S, I, A]$  of locations, values, states, inputs and actions with states  $s$  in  $S$  defined over  $L$  and  $V$  such that  $A$  forms a function with domain  $[S, I]$  and range  $[S]$ . Notice that  $S$  does not necessarily include every possible assignment of values in  $V$  to locations in  $L$ . States not in  $S$  are called "illegal" states.


This model makes several simplifying assumptions:

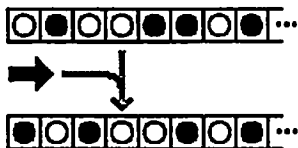
- State is consolidated. In contrast a Turing machine distributes state among the tape, the tape head, and the program.
- Every input is possible at every state.
- State transitions do not depend on time. Time can be modeled by including a clock pulse in the input.

locations: 

values: 

state: 

input: 

action: 

We may visualize a machine as an "interaction diagram" — a labeled directed graph in which states are nodes and actions are edges. Each edge is labeled with an input. Each node has exactly one exiting edge for each input.

For example, a light switch may be visualized as an interaction diagram with two states, on and off. Each state has exactly one exiting edge leading to the other state. Notice that a conventional, two-position light switch contains redundant information—you can't turn on a light that is already on. Since only one action is possible at each state, a simpler trigger button would suffice. The redundancy allows the user to always perform the same action to get the same result, without having to first check the current state of the light bulb.

For the purposes of an interaction diagram, the internal structure of a state does not matter. Therefore we may drop the concepts of location and value. We are interested only in whether two states are the same.

Interaction diagrams are similar to the finite state automata known as "synchronous sequential machines." [Kohavi] A synchronous sequential machine includes a set  $I$  of inputs, a set  $O$  of outputs, a set  $S$  of states, a state transition function  $\delta: I \times S \rightarrow S$  mapping an input and a state to another state, and an output function  $\lambda$  that gives the current output.

There are two types of synchronous sequential machines. In a Mealy machine the output function  $\lambda: I \times S \rightarrow O$  depends on both the current input and the current state. In a Moore machine, the output function  $\lambda: S \rightarrow O$  depends only on the current state.

Mealy machines and Moore machines are logically equivalent. To convert a Mealy machine into a Moore machine, simply fold the current input into the state, multiplying the number of distinct states by the number of possible inputs. This is essentially what we have done in Viewpoint. To make Viewpoint a Moore machine, we have included the input (keyboard or mouse input) as part of the state (frame buffer).

## Visibility

The formal model suggests a theory of interactive systems. We will develop only the part of the theory relevant to Viewpoint: visibility.

A system is "visible" if it behaves as if the state of the frame buffer is the entire state of the system. In other words, the next frame buffer state is entirely determined by the current frame buffer state and the current input. Notice that information that never changes does not have to be present in the frame buffer, since it is never necessary for determining the next frame buffer state. Only information that changes must be present.

Visibility is the logical extreme of direct manipulation. Direct manipulation systems give the illusion that the pictures on the screen are the objects being manipulated. In a visible system the illusion is perfect—the frame buffer always shows the complete situation.

In practice, strict visibility is neither practical nor desirable. Usually there is more information than will fit on a single display. Even when all the information would fit on the screen, it is often better to show only facts relevant to the immediate situation. A little ambiguity may be better than visual clutter. Finally, a single frame buffer is not the only way to show system state to the user: Blinking cursors use motion, pop-up menus use interactivity, and alert beeps use sound. The only perfectly visible systems are video games that tailor their world to fit on a single screen.

A system is "comprehensible" if the screen is at all times understandable to the user. Comprehensibility is not a formally provable condition but is important to mention in connection with visibility: Visibility alone does not guarantee comprehensibility since a memory dump to the screen is sufficient to satisfy visibility.

We assume that values stored at each location in the frame buffer are mapped to colors of corresponding pixels on the screen according to a fixed color map so that there is a close correspondence between frame buffer and screen. We will often speak of the frame buffer and screen as if they were the same, but it is important to realize that they are distinct—the computer does see the screen, and the user does not see the frame buffer.

Viewpoint is built to satisfy a stricter form of visibility called, "input visibility". A system is "input visible" if it is visible and the previous input is derivable from the current state. I pushed Viewpoint to satisfy input visibility because I wanted to explore visibility in its most extreme form.

Input visibility permits a program architecture called the "split brain". In such a system, two simultaneously executing programs, the input program and the processor program, communicate with each other solely through the frame buffer. First, the input program reads the current frame buffer and input devices and writes a special intermediate frame buffer. Then, the processor program reads the intermediate frame buffer (but not the input devices) and writes the next frame buffer.

To avoid collisions, the two programs must agree on special ways of using the frame buffer. In Viewpoint, the input and processor programs write prearranged symbols to particular locations to give the other program permission to proceed. Another solution is to divide the frame buffer into separate regions, each of which can be changed only by a particular program. These solutions are similar to devices used to coordinate concurrent programs.

To verify that Viewpoint satisfies visibility, we must be able to construct a program that reads the current frame buffer and user input and then writes the next frame buffer. If Viewpoint were actually implemented this way, the program itself would constitute a proof. Since it is not, we must take another approach.

We will verify visibility in three steps. First, we will describe Viewpoint in terms of the formal model of interactive computational systems. Next, we will determine what information the program must be able to deduce from the frame buffer. Finally, we will show that all such information is unambiguously represented in the frame buffer at all times. To verify input visibility, we will show that the previous input is unambiguously represented in the frame buffer at all times. This will suffice to prove that an explicit frame buffer processing program could be written.

## Modeling Viewpoint

Recall that a machine is defined as a collection of locations, values, states, inputs, and actions. In Viewpoint, these correspond to the following elements.

**Locations** correspond to pixels in the frame buffer. There are  $640 \times 480 = 307,200$  different pixel locations.

**Values** correspond to colors a pixel may be assigned. There are  $2^4 = 16$  different color values, each a mixture of 4 basic colors: black, red, green and blue.

A **state** assigns each pixel in the frame buffer a color.

An **input** includes the current state (up or down) of all keys on the keyboard and mouse, and either the name of the key that just changed state or the current mouse motion. We assume that input occurs only when the last action was either a press or release of a single key, or a mouse move. If the user presses or releases several keys in a row, they are registered as separate inputs.

Key states and key transitions reflect two different uses of keys: as indicators that remain active as long as they are held down and as triggers that fire the moment they are pressed or released. All keys are treated the same way: Logically, the mouse buttons are considered part of the keyboard; any key can act as a Shift key. Since input tells us the current state of all keys, we can deduce whether the key that just changed state just went down or just went up.

Mouse input is given as a pair of relative coordinates that measure the distance between the previous mouse position and the current. To determine the next cursor position, the coordinates are added to the current cursor position as deduced from the frame buffer.

**Actions** are implemented by a program that transforms a state and an input into another state. For the purposes of analysis, we may consider that all actions are handled by a single action program.

To show that Viewpoint satisfies visibility, we must find out what information the program must deduce from the frame buffer. The table shown opposite lists the information the action program reads (sees) and writes (draws) at each step. For Viewpoint to be visible, the information read at each step must either be deduced from the frame buffer or be unchanging.

ACTION	STEPS	SEES	DRAWS
<b>Main program</b>	see interlock if interlock = user then do User program if interlock = processor then do Processor program erase current interlock draw other interlock	interlock	interlock interlock
<b>User program</b>	see current cursor position erase key highlights erase key triggers erase cursor draw highlights for all keys down draw trigger for key that just changed draw new cursor	cursor	key highlights key triggers cursor key highlights key trigger cursor
<b>Processor program</b>	do Type, Draw, Select and Copy map selection back to puffbox	selection, cells	puffbox
<b>Type</b>	if CR key just went down then move cursor to start of next line  if BS key just went down then move cursor to previous cell  if printing key just went down then if cursor on nonchar then word wrap if necessary type character  move cursor to next cell	key trigger cells, font, cursor  key trigger cells, font, cursor  key trigger cells, font cells, font, cursor cells, font, cursor key highlights cursor	cursor  cursor  cells, cursor  cursor
<b>Draw</b>	if left mouse button just went down see color of current cell draw current cell pen color draw ink color  else if left mouse button down then draw current cell ink color if current cell is in puffbox map puffbox to selection	key trigger cursor, cells cursor  key highlight ink color, cursor cursor, puffbox puffbox, selection	cells ink color  cells cells
<b>Select</b>	if middle mouse button down then draw current cell select color	key highlight current loc	
<b>Copy</b>	if right mouse button down then copy selection to current cell if current cell is in puffbox map puffbox to selection	key highlight selection, cursor cursor, puffbox puffbox, selection	cells cells



## Verifying visibility

To verify visibility we must verify that all information read by the action program can be deduced unambiguously from the frame buffer. We must check three items. (1) Encoding: All information is encoded in pixels. (2) Assumptions: Any conditions the encoding assumes remain true at all times. (3) Visibility: The encoding remains unambiguous at all times.

The problem of encoding information in an array of pixels is similar to the problem of encoding information in a string of characters. Programming languages use conventions of punctuation, reserved words, and word order to achieve unambiguous syntax. Similarly, Viewpoint uses conventions of color, position, and shape to achieve unambiguous graphic syntax. Many graphic encoding techniques are possible. Viewpoint uses only the simplest techniques.

In the discussion below, the "contents" of a cell refers to the values in the black plane of the cell.

---

<b>Grid</b>	Encoding: blue. Used only by user. Not used by program. Assumptions: Never changed. Visibility: Nothing else is blue.
<b>Selection</b>	Encoding: green, shape. Specifies a particular cell. Assumptions: There is exactly one selection, drawn as a solid green cell. Visibility: Nothing else is green.
<b>Key highlight</b>	Encoding: red, position. Specifies a keyboard or mouse key. Assumptions: Each key has a fixed key highlight pattern. Visibility: Doesn't overlap key trigger, not hidden by cursor.
<b>Key trigger</b>	Encoding: red, position. Specifies a keyboard or mouse key. Assumptions: There is at most one key trigger. Each key has a fixed pattern for its key trigger. Visibility: Doesn't overlap key highlights, not hidden by cursor.
<b>Cursor</b>	Encoding: red, shape. The arrow tip specifies a pixel. Assumptions: There is exactly one cursor, drawn as a red arrow clipped to the frame buffer area. Visibility: Shape is unambiguous even when overlapping key highlights or triggers.

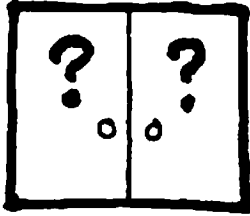
---

---

<b>Cells*</b>	<p>Encoding: black. Encodes "data", including both text and graphics.  Assumptions: None.  Visibility: There is no inherent distinction between text and graphics. The interpretation is determined by the operation being performed.</p>
<hr/>	
<b>Interlock</b>	<p>Encoding: black, position, shape. Encodes which program is currently running: the user or the processor.  Assumptions: There are two interlock cells, one of which contains no black pixels and the other of which contains a black triangle of fixed shape.  Visibility: The interlock is redrawn after each action.</p>
<hr/>	
<b>Font</b>	<p>Encoding: black, position. Encodes a ten by ten image of black and white pixels for each printing character on the keyboard.  Assumptions: Each printing character has a fixed cell that specifies the corresponding character image. Printing characters include uppercase and lowercase letters, numbers, symbols, and the space character.  Visibility: No ambiguity possible.</p>
<hr/>	
<b>Puffbox</b>	<p>Encoding: black, position. A magnified view of the contents of the selection.  Assumptions: The puffbox is ten cells by ten cells in size, and occupies a fixed position. Each cell in the puffbox is all white or all black in its black plane. A black cell in the puffbox corresponds with a black pixel in the selection; a white cell corresponds with a white pixel in the selection.  Visibility: The contents of the puffbox is made to correspond to the contents of the selection at the end of every processor action. User actions do not affect the contents of either the puffbox or selection.</p>
<hr/>	
<b>Ink color</b>	<p>Encoding: black, position. Specifies the color (black or white) currently being drawn.  Assumptions: The ink color is a fixed cell. While the left mouse button is held down, the ink color remembers the color drawn when the left mouse button was first pressed. Color is determined by looking at the color of a particular pixel in the ink color cell.  Visibility: When the left mouse button is first pressed, the ink color is drawn correctly. While the left mouse button is held down, the only color that can be drawn on top of the ink color is the same color it is already drawn as.</p>

---

\* Pieces of information encoded in the black plane occupy fixed positions that do not overlap with one another and therefore cannot interfere with one another.



## 6 Opportunities

This chapter recommends opportunities for future research suggested by Viewpoint.

- Extending the program
- Algorithms and machine architecture
- Theory
- Editor-based systems
- Visual programming
- Interaction design
- Visual thinking

## Extending the program

The most immediate opportunity for future research is to extend the Viewpoint program itself while maintaining visibility. Other alternatives are discussed in Chapter 4.

**Actions.** Implement alternate versions of existing actions: cursor and selection move in parallel during Copy, Copy instead of Draw, and extended selection.

**Input devices.** Redesign actions to work with input devices, such as a pen, dual mice, or touch screen.

**Sizes.** Allow Viewpoint to act on a space larger than a single screen. Invent a mechanism for navigating the space. Redefine visibility accordingly. Alternatively, keep the screen size the same but change pixel size.

**Variability.** Allow features that are constant in the current version of Viewpoint to be variable: puffbox location, interlock location and shape, keyboard image location, mapping between keys and key images, grid position and spacing, cursor shape, and mapping between pixel values and screen colors. This requires making the variable information visible.

**Fonts.** Generalize text to allow variable width fonts, multiple font styles (bold, italic, etc.), or multiple fonts. This makes character parsing harder.

**Layout.** Allow complex page layouts such as multiple columns. Dynamically reformat text to fit the layout during editing operations. This makes text parsing harder.

**Structured graphics.** Generalize drawing to make use of such geometric relations as connectedness, colinearity, rectilinear alignment, containment, overlapping, and proximity. This requires new parsing algorithms and graphic representations.

**Color.** Allow all colors to be edited. One solution is to allow color planes to be interchanged. Redesign Viewpoint to work on a monochromatic display. This means color cannot be used as a graphic encoding technique.

**Graphic design.** For each object, explore alternate graphic appearances. Evaluate their effectiveness. Test Viewpoint on users. Improve it.

**Application.** Enhance Viewpoint so it can be used for real application such as user interface prototyping or animation.

**Multiple users.** Generalize the interlock mechanism to allow more than one user to act on the screen without conflict, perhaps simultaneously.

**Purity.** Reimplement Viewpoint so that the frame buffer is the only persistent data structure.

## Algorithms and machine architecture

Another area for future research is pixel processing algorithms and machine architectures that allow Viewpoint to run more efficiently. Some work is already happening in this area. The new push will be support for pixel parsing, not just pixel display.

**Pixel parsing algorithms.** One direction is to extend text parsing techniques used by compilers into the realm of pixels. Another direction is to borrow digital filtering techniques from visual pattern recognizers. For instance, the Viewpoint cursor contains a solid 4 by 5 square of red pixels. Hence we only need to sample one pixel in 20 to locate the cursor approximately. Searching the local 4 by 5 vicinity would then reveal the exact location. Similar techniques are used by programs simulating human vision, except that they must cope with noise.

Bernard Mont-Reynard has explored the use of bitblit (fast operations on rectangular blocks of pixels) for pattern recognition. Here is a way to use bitblit to find an isolated red pixel in a frame buffer: First, use bitblit to OR the top half of the frame buffer onto the bottom half. Then, fold the bottom half into the bottom quarter. Keep folding the screen until it is reduced to a line. We can now locate the  $x$  coordinate of the red pixel by searching 1,000 pixels linearly. To find the  $y$  coordinate, fold the screen repeatedly from left to right. [Mont-Reynard].

**Hardware.** The obvious next step is to move pixel parsing algorithms into hardware. Special purpose display hardware is becoming common as the demand increases for three-dimensional modeling and real-time simulation. The challenge is to use such processors for image analysis and to make them fast enough that an entire frame buffer can be processed in a single cycle.

**Applications.** Pixel processing algorithms and hardware are driven by applications such as window systems, data compression, and optical character recognition. As computers merge with high bandwidth visual communication media, we can expect more and more demand for pixel processing power.

**Pixel calculus.** There is a strong bias in computer science that treats bitmaps as "imperfect and mathematically uninteresting approximations to the ideal images of plane geometry." [Guibas] Leo Guibas and Jorge Stolfi have proposed a calculus of bitmap operations which leads to a programming language for bitmap computations. Such research will be important for giving pixel algorithms a mathematical foundation.

## Theory

The model of interactive computational systems described in this dissertation was developed for a specific purpose: to verify visibility. Many directions remain to be explored.

**Criticism.** Is it sound? Is it adequate for describing existing interactive systems? How does it relate to other formal models of computers? How does it relate to models of interactive systems in fields outside computer science, such as control theory, communications, psychology, and biology?

**Extensions.** My model applies only to a single user interacting with a single processor without dependence on timing. Furthermore, I have assumed that the entire state is visible on the screen. Extend the model to apply to multiuser systems, multiprocessor systems, networked systems, real-time systems, and systems that are not strictly visible. This will mean distinguishing subclasses of symbols, locations, and inputs. For instance, users can be modeled by distinguishing

**Principles.** Use formal models to define interaction design principles such as modes, modelessness, monotony, transparency, etc.

Analyze existing systems to see where they do or do not follow principles. Test systematic variations of programs to discover the consequences of following principles.

More generally, study the structure of visual representation [Levy], the notion of representation itself [Brian Smith], or the nature of computers as language machines [Winograd].

**Prototypes.** Design thought experiments and software prototypes to study interface design principles in their purest form. Viewpoint is one such experiment; there need to be more. To study modality rigorously, Jef Raskin developed a minimal interface called the "buzzer"—one bit of input and one bit of output [Raskin]. Its properties have yet to be explored.

Most user interface research projects are driven by practical applications and are consequently too complex to let principles stand out clearly. We need to build simpler programming environments tuned to the needs of user interface prototyping. Trying to build a clean interface in the type of programming environment typical of computer research institutions is like trying to build a well-structured program in Fortran: You can do it, but only if you have already learned the discipline in a cleaner environment.



## Editor-based systems

Viewpoint acts on pixels. The lessons of Viewpoint apply equally well to other representations of data.

**Bits.** Bits were the first flat form of data. Early computers explored many strategies for encoding information in bits. Tradeoffs included fixed-length bytes versus variable length words, binary versus binary coded decimal representations of whole numbers, and uniform memory space versus special-purpose registers. It's worth re-examining these decisions to see what we can learn about encoding information in pixels.

Register switches and console lights made up the earliest direct manipulation interface for computers. Machine language programmers experience much of the same satisfaction as users of WYSIWYG page layout systems. What can we learn from the similarities?

**Text.** One of the biggest inspirations for Viewpoint was text editor based systems. Such systems insist on a uniform model of interaction—editing—and a uniform model of data—a linear string of characters.

Now that I have built a pure pixel editor, I see that text editors could be pushed much further. Current text editors store many elements of system state, such as the cursor and selection, separately from the main text. To fold these into the text, we could use parallel text strings, much the way Viewpoint uses parallel pixel planes.

Current text editors force users to keep track of several kinds of white space: space, tab, return and no character. Displaying explicit tab and return characters is no solution; the user must then keep track of which characters will print on the printer. The problem arises because the two-dimensional display derives from a one-dimensional text.

In contrast, Viewpoint derives text from the screen. Similarly, the innovative WYSIWYG editor WE [Burchfiel] stores text as a rectangular array of character cells. Every cell contains exactly one character. There is only one kind of space. WE scans patterns of spaces and characters to locate paragraph and column breaks, just as Viewpoint parses pixels to locate margins and word breaks. WE and Viewpoint leave many hard questions unanswered.

**Structured graphics.** Fred Lakin's text-graphic manipulation environment PaM parses spatial structures from low-level collections of line segments [Lakin]. Other graphic primitives need to be explored. Fanya Montalvo proposes higher-level visual pattern recognizers capable of making analogies [Montalvo]. How could such mechanisms assist visual editors?

## Visual programming languages

I originally wanted Viewpoint to be a programming language, not just an editor. Having built a visual editor, I now see a clear direction for future visual programming languages.

**Closing the loop.** Current visual programming languages are blind: They cannot "see" their own visual representations. The images on the screen are strictly for the user. Programs themselves see only abstract data structures.

I advocate visual programming languages that have full access to the same program representations the user sees. Just as the textual programming languages Lisp, Smalltalk, and Snobol are able to parse their own textual structure, so a "self-seeing" visual programming language should be able to parse its own graphic structure. This reflexive quality makes an interpreter for the language easy to write in the language itself. It also leads to overall simplicity and uniformity. Without full reflexivity, the visual aspects of visual programming languages are doomed to the same fate as strings in Fortran: a useful but second-class data structure.

**Bringing in visual experts.** Once a programming language can see its own visual representation, programming language design becomes a new type of problem. Visual representations must serve both the user's eye and the computer's parser. Therefore, the design of visual programming language should include cognitive psychologists and graphic designers as well as programmers.

Designing a visual programming language purely on programming principles would be as naive as designing a scientific diagram purely on visual aesthetics. In truth, psychology and art have always been important for software design. Visual programming only makes the issue less avoidable.

**Rethinking programming.** Today's programming languages such as C, Lisp, and Smalltalk are as much editing environments as they are programming languages. Visual programming languages go a step further. Visual programming will be most at home in environments that are built first as editors for manipulating graphic objects and only secondarily for programming. Therefore, we need to shift our thinking from editing in support of programming to programming in support of editing.

## Interaction design

Working on Viewpoint made me aware of the embryonic state of user interface design as a field of research. Most of my own tools and theory I had to build from scratch.

**What's missing.** Programmers are building computer systems. Psychologists are studying the effects of computers on people. But no one yet is systematically exploring of the space of *possible* interface designs based on precisely defined principles. In other words, there is practice and analysis, but no theory.

Without such exploration, programmers and psychologists work in a vacuum. Building a window system without visual representation theory is like building a compiler without formal language theory—the solutions are *ad hoc*. Studying the efficiency of cursor keys versus the mouse is like studying the efficiency of hard calculation versus the slide rule before the advent of the pocket calculator—a better solution may render the issue irrelevant.

**New priorities.** User interfaces can be systematically studied. But it won't look like conventional computer science or psychology. Most current user interface research is added as a footnote to research with other goals. For the field of user interface design to come into its own, the user interface as communication medium must get top priority.

The first change is to include users as full team members from the very beginning of the research, not just as test subjects brought in at the end. Researchers would do well to imitate video game manufacturers, who include user feedback as an integral part of the design process.

The next change is to think top-down: first, about problems, second, about human-machine partnerships for solving problems, and third, about the particular needs of humans and computers. Researchers would do well to study the conceptual framework for human augmentation laid out by Douglas Engelbart in the early 60s [Engelbart].

**Setting up research.** A good example of a new style of interaction design research is the Media Lab at MIT [Brand]. Notice that the word "computer" is entirely missing. From the point of view of the Media Lab, communication comes before computation. This shift has several consequences: printing presses and paint brushes become research tools, animators and musicians become colleagues, and videotapes and performances become the products of research. For this style of work to thrive, researchers in many disciplines must work cooperatively.

## Visual thinking

To build better visual interfaces, we must understand visual thinking itself apart from computers.

**Definition.** I have yet to see a convincing definition of "visual thinking". Many authors offer examples (e.g. "Count the doors in your house"), but few offer ways to distinguish visual thinking from other sorts of thinking. Psychologist Roger Shepard has devised pioneering experiments on mental rotations [Shepard]. We need to probe other aspects of visual thinking with equal precision.

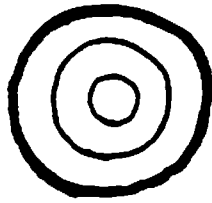
On the other hand, the term "visual thinking" was never meant to survive close scrutiny. Robert McKim once expressed to me his own doubt: Thinking itself has no form; only the products of thinking can be called visual. I think of "visual thinking" as a political cartoon: a deliberate oversimplification intended to raise an issue. Therefore, researchers should not take the words too literally but instead ask how the idea might be refined.

**Practice.** Visual thinking must be practiced to be understood. Beginners often try to understand visual thinking by reading essays. This won't work. Sometimes a picture is worth a thousand words, but when the topic is visual perception itself, no amount of words can substitute for a picture. Even books on visual thinking often suffer for lack of pictures. One visual thinking book that practices what it preaches is the picture essay "How to See" by George Nelson [Nelson].

Visual thinking researchers must be equipped to present their ideas in the correct media. For instance, I devote over 50 pages of this dissertation to snapshots of a system demonstration. Since a black and white printed document cannot show color and motion, I have also published a Viewpoint videotape. (Video is now the preferred medium for publishing user interface research.) Publishing the program itself would be even better.

**Computers.** Today's computers allow us to see and draw in ways that go beyond the limits of other visual media.

Computers can also help visual thinking in another way. Viewpoint sprang from the question "How can a computer use the visual mode of thinking?" To answer this question, I had to stretch the definition of "visual thinking" to apply to computers as well as people. This led to the generalization "pictures first", which in turn led to the idea of pixels as the primary representation of data. I find that stretching an idea always helps me sharpen my understanding. Future research could stretch "visual thinking" in other directions.



## **7 Conclusions**

**This chapter summarizes  
the results of my research.**

- Accomplishments**
- Techniques**
- Insights**

## Accomplishments

The main accomplishment of Viewpoint is to demonstrate by example that a strictly visible text and graphics editor is possible. Both the precise definition of visibility and the possibility of attaining it are new ideas.

**Theory.** To give Viewpoint a theoretical foundation, I constructed a formal model of interactive computational systems, defined visibility in terms of the model, and proposed a procedure for verifying visibility. Both the model and the procedure are new.

**Program.** To demonstrate the consequences of visibility, I built the Viewpoint system, used it, watched other people use it, and repeatedly revised the system. The program was invaluable for developing my ideas and for explaining the ideas to other people.



## Techniques

The Viewpoint program illustrates many new programming and visual presentation techniques for implementing graphic user interfaces. Though I put much effort into refining these techniques, I did not try to study them thoroughly. Therefore, they should be taken as exploratory guesses, not conclusive recommendations.

**Pixels.** The most important new technique in Viewpoint is the use of pixels as the primary representation of structured state information. Previous systems have used pixels only for unstructured data, as in a painting system, or as a secondary representation of hidden data structures. The primary role of pixels in Viewpoint allows text and graphics to be integrated in a new way—the difference between text and graphics is determined not by the data but by the operation.

Equally important is the use of cells as the primary chunking of pixels. Pixels and cells are visually coarse, but convenient for editing. There will always be a role for fixed width fonts and low-resolution pixels, even as higher-resolution displays become available.

**Graphic encoding techniques.** Viewpoint demonstrates four ways of encoding information unambiguously in pixels: color, position, shape, and structure. Viewpoint uses transparent color planes to separate different sorts of information. To keep color mixtures distinct while giving the illusion of transparency, I mixed hues accurately but varied intensity freely. To show which object was in front, I shifted hues toward foreground colors.

**Graphic design.** Interesting visual devices include the grid, the cursor, and key highlights. The light blue grid was effective for showing the structure of the screen without being visually intrusive. The spare use of bright colors proved effective in attracting attention to the cursor and selection. A principle that seemed to be helpful was to differentiate objects by shape even when they were already differentiated by color.

**Interactive elements.** Most mouse-driven text editors use two cursors: a pointer and an insertion point. Viewpoint uses only a single cursor. The puffbox allows an editor that acts on cells to edit individual pixels.

**Algorithms.** The implementation of Viewpoint demonstrates techniques of pixel parsing and structure caching. These techniques extend ideas from compiler design and program optimization from the world of text strings into the world of graphics.

## Insights

Working on Viewpoint required that I rethink computers from the viewpoint of graphic design. Here are my insights:

**Computers.** Computer science as currently practiced is not equipped to think about user interface design. I once asked a computer scientist why there isn't more research in text editor design. He admitted, after some hesitation, that editor design was "a job for hackers", implying that editors are not respectable computer science. Despite advances in user interfaces, such attitudes persist. Consequently, there are enormous opportunities for innovative research in user interface design but little support within conventional computer science.

**Graphics.** For graphic designers and other visual thinkers, pictures have first priority. I sum this up in the slogan "pictures first". Just as a programmer will feel uncomfortable looking at a demonstration of a program until you explain how it is implemented, a graphic designer will feel uncomfortable listening to a description of an image until you show what it looks like. Any computer tool for assisting visual thinking must respect this priority.

**Design.** Computer concepts can be divided into two categories according to whether they relate to early or late stages in the design process. This scheme matches the classification shown below, invented by the Xerox Star design team [David Smith]. I've starred concepts particularly relevant to Viewpoint and added five concept pairs of my own. The insight of Viewpoint is to design systems that start in the left column and allow movement to the right.

### STAR

Easy	Hard
concrete	abstract
visible	* invisible
copying and modifying	creating from scratch
choosing from a list	filling in a blank
recognizing	generating
editing	* programming
interactive	batch

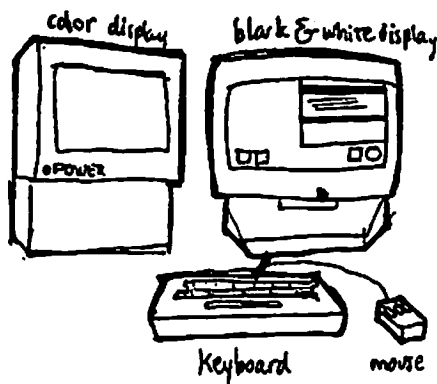
### VIEWPOINT

early design process	* late design process
flat structure	* hierarchical structure
pixels	* structured graphics
graphics	* text

# Appendix: Viewpoint manual

This manual describes a slightly earlier version of Viewpoint that did not yet include key triggers. I used this manual to test Viewpoint on users. Since I wanted the manual to be entirely self-contained, I configured the system to start up Viewpoint directly on log in.

## Getting Started



1. Viewpoint runs on the Dorado, a personal computer built by Xerox mainly for internal research. The Dorado comes with a 1024 pixel wide by 792 pixel high black and white display, a 61-key keyboard, and a 3-button mouse.

2. Because most Dorados are at Xerox PARC, people who try Viewpoint will usually be PARC employees or escorted visitors. If you are a visitor, be sure you have the assistance of an experienced Dorado user to help you log in.

3. To use Viewpoint, you will need a Dorado with a color monitor, running the Cedar programming environment. If the color monitor is a Conrac, make sure it is switched to 640x480 resolution.

4. Make sure both the color and the black and white monitors are turned on. The color monitor's power switch is on the front in the lower left corner. The black and white monitor's power switch is around the back of the base (up = on).

5. Find the small red button on the back of the keyboard ("the boot button") and click it three times in a row ("triple click").

6. You will see the message "Please log in". If you are a visitor to PARC, ask someone to log in for you. To log in, type your user name then press the RETURN key. Use the BS key to correct mistakes. Type your password followed by RETURN. For instance:

```
Please log in...
Name: kim.pa Password: .....
```

7. If the computer asks you

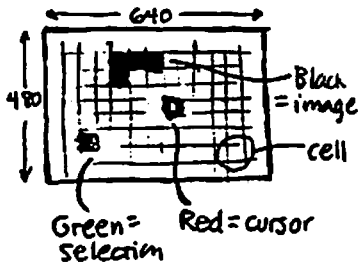
```
Do you wish to install a personal profile?
```

answer Yes by typing the RETURN key twice.

8. After 30 to 90 seconds, an image will appear on the color monitor and you will be ready to use Viewpoint.

## Using Viewpoint

1. From now on, you will be working only on the color screen. The color frame buffer is 640 pixels wide and 480 pixels high. The 640 by 480 pixels are your entire world; the screen cannot be scrolled or expanded. Each pixel may be a mixture of four colors: blue, red, black and green. Each color indicates a different type of information.



2. The blue lines divide the screen into square cells. Each cell is 10 pixels wide and 10 pixels high. To turn the grid on or off, type the unmarked key in the bottom right corner of the keyboard.

3. The red arrow is called the "cursor". Moving the mouse moves the cursor.

4. The black areas are the text and graphics you can edit. Every pixel on the screen can be turned black or white. To edit the text and graphics, you use the mouse buttons and keyboard.

5. Clicking the left mouse button draws by filling in the cell at the cursor either black or white. Move the cursor to an empty area and experiment. Drawing on a white cell turns it black. Drawing on a black cell turns it white. If you keep holding the left button and move the mouse, you will keep drawing in the same color. Everything is fair game for drawing over, even the image of the keyboard. (Caution: there is no "undo" in Viewpoint.)

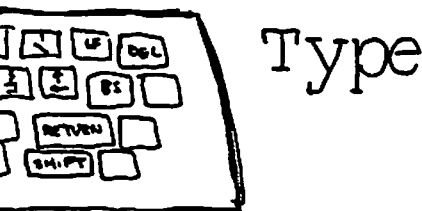
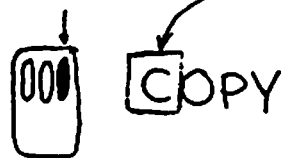
6. Clicking the middle button selects the cell at the cursor by shading it green. The selected cell, also called the "selection", is displayed in magnified form in an area called the "puffbox". Each cell in the puffbox corresponds to a pixel in the selection.

7. Drawing in the puffbox redraws the pixels in the selection. Select the "M" in mouse and change it to an "H" by drawing in the puffbox. Watch the pixels in the selected cell change as you draw.

8. Clicking the right button copies from the selection to the cell at the cursor. Try changing "House" back into "Mouse" by copying the "M" on the keyboard to the first cell of "House".

9. Typing a key copies the contents of the appropriate cell (the "key image") to the current cell, then advances the cursor one cell. Try moving the cursor to the "u" of "Mouse" and type an "o". Typing acts as if it were in "overwrite" mode; there is no "insert" mode. If you type past the edge of the screen, the cursor will wrap to the beginning of the next line.

10. To get the image on the top half of a key, hold either SHIFT key while typing the key. To move the cursor back a cell, use BS. The DEL key erases the current cell by shifting the line of cells to its right to the left one cell (caution: this may disrupt



your drawing). Other keys have no effect.

11. Notice that pressing a key highlights the corresponding key image by drawing a one-pixel-wide border (the "key highlight"). Releasing a key erases the key highlight. Because the font is visible on the screen, editing key images affects subsequently typed letters.

### *Exercises*

1. Redraw your initials as they appear on the keyboard on the screen. Then move the cursor to an empty area and type your name.

2. Exchange two cells.

3. What is the quickest way to turn a cell white?

4. Erase the "8" key without disturbing the nearby keys. Hint: don't use Draw; use only Select and Copy. Widen the "7" key to take its place. Draw a new symbol where the "8" used to be. What happens if you type "8"?

5. What happens if you copy into the puffbox?

6. Select a cell of the gray cursive "L" and you will see it is actually an alternating pattern of black and white pixels. By continuously copying one of these cells with the Copy button, you can paint with a gray brush. Make up another texture patterns to paint with.

7. Erase the entire screen by typing CTRL-DEL. How would you reconstruct the original screen?

# Bibliography

- | <i>Referred to on page</i> | <i>Reference</i>  |
|----------------------------|---|
| 20                         | Alzofon, David; Caulkins, David; Raskin, Jef; Winter, Dr. James. <i>The Canon Cat How-to Guide</i> . Canon Inc., 1987.  |
| 90                         | Apple Computer. <i>Human Interface Guide: The Apple Desktop Interface</i> . Addison Wesley and Apple Computer, 1987.  |
| 13                         | Arnheim, Rudof. <i>Visual Thinking</i> . University of California Press, 1969. ISBN: 0-520-01378-6.   |
| 22                         | Atkinson, Bill. <i>MacPaint</i> . Claris, 1988.   |
| 15                         | Bigelow, Charles; Day, Donald. "Digital Typography." <i>Scientific American</i> , vol. 249, no. 2 (August 1983), pp. 106-119.   |
| 117                        | Brand, Stewart. <i>The Media Lab. Inventing the Future at MIT</i> . Viking, 1987. ISBN: 0-670-81442-3.  |
| 95                         | Brinch Hansen, Per. <i>Operating System Principles</i> . Prentice-Hall, 1973.   |
| 115                        | Burchfiel, J.; Myer, T. <i>Message Technology Research and Development</i> . Bolt Beranek & Newman, 1976. BBN Report no. 3440. Quarterly progress report: 7/2/76 through 10/2/76. |
| 101                        | Card, Stuart; Moran, Thomas; Newell, Simon. <i>The Psychology of Human-Computer Interaction</i> . Lawrence Erlbaum Associates, 1983. ISBN: 0-89859-243-7.                         |
| 16                         | Chang, Shi-Kuo; Ichikawa, Tadao; Ligomenides, Panos A. <i>Visual Languages</i> . Plenum Press, New York and London 1986. ISBN 0-306-42350-2.                                      |
| 101                        | Ciccarelli, Eugene. <i>Presentation-Based User Interfaces</i> . MIT PhD dissertation. MIT Artificial Intelligence Laboratory Technical Report no. 794, 1984.                      |
| 22                         | Cutter, Mark. <i>MacDraw</i> . Claris, 1988.  |
| 117                        | Engelbart, Douglas. "A Conceptual Framework for the Augmentation of Man's Intellect." in <i>Vistas in Information Handling</i> , volume 1. Spartan Books, 1963.                   |



- 113 Guibas, Leo; Stolfi, Jorge. "A Language for Bitmap Manipulation." *ACM Transactions on Graphics*, ACM Siggraph, vol. 1, no. 3 (July 1982), pp. 191-214.
- 100 Heckel, Paul. *The Elements of Friendly Software Design*. Warner Books, 1984. ISBN: 0-446-38040-7.
- 101 Hoare, C. A. R. *Communicating Sequential Processes*. Prentice/Hall International, 1985.
- 18 Jones, J. Christopher. *Design Methods: Seeds of Human Futures*. Wiley-Interscience, 1970. ISBN 0-471-44790-0.
- 18 Kim, Scott. "Factory." *The AI Magazine*, American Association for Artificial Intelligence, vol. 5, no. 3 (Fall 1984), front cover.
- 14, 15 Knuth, Donald E. *The Metafont Book*. Addison Wesley, 1986. ISBN: 0-201-13444-6.
- 101 Kohavi, Zvi. *Switching and Finite Automata*. 2nd edition. McGraw-Hill, 1978.
- 16, 115, 116 Lakin, Fred. "Spatial Parsing for Visual Languages," in Chang, Shi-Kuo; Ichikawa, Tadao; Ligomenides, Panos A. *Visual Languages*. Plenum Press, New York and London 1986. ISBN 0-306-42350-2.
- 17 Lanier, Jaron Z. "Mandala." *Scientific American*, vol. 251, no. 3 (September 1984), cover illustration.
- 114 Levy, David. *Formalizing the Figural. Aspects of a Foundation for Document Manipulation*. To be published, 1988.
- 101 Manna, Zohar. *Mathematical Theory of Computation*. McGraw-Hill, 1974. ISBN: 0-07-039910-7.
- 17 Marcus, Aaron; Baecker, Ron. "On the Graphic Design of Program Text." *Proceedings, Graphics Interface '82*. National Research Council of Canada, 1982. ISSN: 0713-5424. NRCC: 20193.
- 4 Marcus, Aaron. "Graphic Design for Computer Graphics." *IEEE Computer Graphics and Applications*, vol. 3, no. 7 (July 1983), pp. 63-82.

- 11 McKim, Robert. *Experiences in Visual Thinking*. Brooks/Cole, 1972. ISBN 0-8185-0031-X
- 90 Microsoft Corporation. *Excel*. Microsoft Corporation, 1987.
- 113 Mont Reynard, Bernard. Private communication, 1987.
- 115 Montalvo, Fanya. "Diagram Understanding: Associating Symbolic Descriptions with Images," pp. 4-11 in *Workshop on Visual Languages*. IEEE Computer Society, 1986. IEEE Catalog Number 86CH2343-2. ISBN: 0-8186-0722-X.
- 16 Myers, Brad A. "Visual Programming, Programming by Example and Program Visualization; A Taxonomy," *Human Factors in Computing Systems*, 1986, pp. 59-66. Proceedings of SIGCHI '86, Boston, Massachusetts, April 13-17, 1986.
- 18 Negroponte, Nicholas. "On Being Creative with Computer Aided Design." *Information Processing 77*, B. Gilchrist, ed.
- 118 Nelson, George. *How To See. A Guide to Reading our Manmade Environment*. Little, Brown, 1977. ISBN: 0-316-60312-0.
- 101 Newman, William M.; Sproull, Robert F. *Principles of Interactive Computer Graphics*. McGraw-Hill, 1979. ISBN 0-07-045338-7.
- 101 Norman, Donald A.; Draper, Stephen W. *User Centered System Design*. Lawrence Erlbaum Associates, 1986. ISBN: 0-89859-781-1, 0-89859-872-9.
- 25 Pier, Kenneth A. "A Retrospective on the Dorado, A High-Performance Personal Computer." *Proceedings of the 10th Symposium on Computer Architecture, SigArch/IEEE* (June 1983), pp. 252-269.
- 16 Raeder, Georg. "A Survey of Current Graphical Programming Techniques." *IEEE Computer*, vol. 18, no. 8 (August 1985), pp. 11-25. ISSN: 0018-9162.
- 16 Robinett, Warren. *Rocky's Boots*. The Learning Company, 1981.
- 118 Shepard, Roger N.; Cooper, Lynn A. *Mental Images and their Transformations*. MIT Press, 1982. ISBN 0-262-19200-4.

- 2 Shneiderman, Ben. "Direct Manipulation: A Step Beyond Programming Languages." *IEEE Computer*, vol. 16, no. 8 (August 1983), pp. 57-
- 14 Siegel, David R. *The Euler Project at Stanford*. Stanford Computer Science Department, 1985.
- 114 Smith, Brian. *The Correspondence Continuum*. Center for the Study of Language and Information. CSLI-87-71. January 1987. An earlier version appeared with the Proceedings of the Sixth Canadian Conference on Artificial Intelligence, Montreal, Quebec, Canada, May 21-23, 1986.
- 2, 20, 122 Smith, David Canfield; Irby, Charles, Kimball, Ralph; Verplank, Bill; Harslem, Eric. "Designing the STAR User Interface," pp. 297-331 in Degano, Pierpaolo; Sandewall, Erik. *Integrated Interactive Computing Systems*. North-Holland Publishing Company, 1983. ISBN: 0-444-86595-0.
- 13 Sommer, Robert. *The Mind's Eye: Imagery in Everyday Life*. Dale Seymour Publications, 1978. ISBN 0-86651-259-4 (previously ISBN 0-440-03950-9).
- 15 Southall, Richard. *Designing New Typefaces with Metafont*. Stanford Computer Science Department Report no. STAN-CS-85-1074, 1985.
- 115 Stallman, Richard. "EMACS: The Extensible, Customizable, Self-Documenting Display Editor." Pp. 300-325 in Barstow, David R.; Shrobe, Howard E.; Sandewall, Erik. *Interactive Programming Environments*. McGraw-Hill, 1984. ISBN 0-07-003885-6.
- 12 Stewart, Douglas. "Teachers aim at turning loose the mind's eye." *Smithsonian Magazine*, vol. 16, no. 5 (August 1985), pp. 44-55.
- 113 Stone, Maureen; Plass, Michael. "Curve-Fitting with Piecewise Parametric Cubics." *Computer Graphics, ACM Siggraph*, vol. 17, no. 3 (July 1983), pp. 28-32. Proceedings of Siggraph '83 conference.

- 16 Sutherland, Ivan. *Sketchpad: A Man-Machine Graphical Communication System*. Garland Publishing, Inc. 1980. ISBN 0-8240-4411-8.
- 25 Swinehart, Daniel C.; Zellwegger, Polle T.; Hagmann, Robert B. "The Structure of Cedar." *SIGPLAN Notices*, vol. 20, no. 7 (July 1985), pp. 230-244. Proceedings of the ACM SIGPLAN '85 Symposium on Languages Issues in Programming Environments.
- 95, 100 Tesler, Larry. "The Smalltalk Environment." *Byte magazine*, vol. 6, no. 8 (August 1981), pp. 90-147.
- 20 Tesler, Larry. "Programming Languages." *Scientific American*, vol. 251, no. 3 (September 1984), pp. 70-78.
- 117 Verplank, William; Kim, Scott. "Graphic Invention for User Interfaces: An Experimental Course in User Interface Design." *SIGCHI Bulletin*, vol. 18, no. 3, pp. 50-65.
- 20 Weinreb, Daniel; Moon, David. *Lisp Machine Manual*. Symbolics Inc., 1981.
- 23 Wexelblat, Richard. *History of Programming Languages*. Academic Press, 1981. ISBN: 0-12-745040-8.
- 3 Wigginton, Randy; Ruder, Ed; Breuner, Don. *MacWrite*. Claris, 1988.
- 114 Winograd, Terry; Flores, Fernando. *Understanding Computers and Cognition*. Academic Press.