

Example-Based Graphical Programming:
An approach for Graphic Design in Electronic Media

by

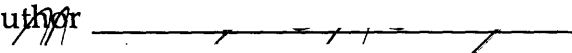
Suguru Ishizaki

Bachelor of Art and Design
Tsukuba University, Tsukuba, Japan
1986

SUBMITTED TO THE MEDIA ARTS AND SCIENCES SECTION IN PARTIAL
FULFILLMENT FOR THE REQUIREMENTS OF THE DEGREE OF
MASTERS OF SCIENCE IN VISUAL STUDIES
AT THE MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUNE, 1989

© Massachusetts Institute of Technology, 1989 All right reserved

Signature of the author



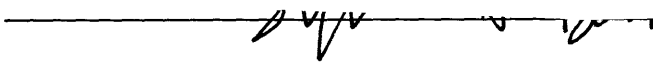
Suguru Ishizaki
Media Arts and Sciences
May 12, 1989

Certified by



Muriel Cooper
Professor of Visual Studies
Thesis Supervisor

Accepted by



Stephen Benton
Chairman
Departmental Committee for Graduate Students

ROTC
MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

OCT 23 1989

LIBRARIES

Example-Based Graphical Programming:

An approach for Graphic Design in Electronic Media

By

Suguru Ishizaki

Submitted to the Media Arts and Sciences Section on May 12, 1989 in partial fulfillment of the requirements for the degree of Master of Science in Visual Studies

Abstract

The Example-Based Graphical Programming System is an interactive computer environment that allows a graphic designer to create a computer program that mimics design examples. The system records the designer's graphical editing process. Then the recorded procedures are generalized into a Lisp function that can be applied in future examples. This system is intended for programming the visual style of graphical elements in a dynamic environment. An example of using this system as a tool for designing the dynamic display of animal migration is presented.

This work was supported in part by DARPA, NYNEX and Hewlett Packard.

Thesis Supervisor: Muriel Cooper
Title: Professor of Visual Studies

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Muriel Cooper, for introducing me to the field of design and technology at the Visible Language Workshop.

I would also like to thank Henry Lieberman for helping me focus and clarify my ideas. His research in programming-by-example was one of the main inspirations of this work.

Thanks are also due to Ron MacNeil and Patrick Purcell for their useful comments and encouragement.

I am particularly grateful to Mark Gross for his constructive criticism and for useful suggestions in organizing this thesis.

Acknowledgement must also be made to my colleagues at the Visible Language Workshop. I would like to thank Sylvain Morgaine for his friendship, Bob Sabiston and Russell Greenlee for providing the graphical interface environment. I also received much support from Ming Chen, Laura Robin and David Small, while I was writing this thesis.

Finally, my thanks to the Media Laboratory for the opportunity to work in a stimulating environment.

TABLE OF CONTENTS

| | | |
|-----------|--|----|
| | Abstract | 2 |
| | Acknowledgements | 3 |
| CHAPTER 1 | Introduction | 5 |
| | 1.1 Motivation | 5 |
| | 1.2 Examples as a means of communication | 7 |
| | 1.3 Related work | 9 |
| | 1.4 The structure of the thesis | 12 |
| CHAPTER 2 | Overview | 13 |
| | 2.1 The Example-Based Graphical Programming System .. | 13 |
| | 2.2 Scenario | 16 |
| CHAPTER 3 | Implementation | 26 |
| | 3.1 Representation and basic recording mechanism | 26 |
| | 3.1.1 Graphical objects | 26 |
| | 3.1.2 Representation of the manipulation | 27 |
| | 3.1.3 Recording mechanism and design procedure .. | 28 |
| | 3.2 Generalization | 30 |
| | 3.2.1 Creation of parameters | 32 |
| | 3.2.2 Creation of conditional structures | 34 |
| | 3.2.3 Generalization as a search | 36 |
| | 3.3 An example of the generalization process | 38 |
| CHAPTER 4 | Migration Graphics | 44 |
| | 4.1 Design requirements | 45 |
| | 4.2 A session with the system | 46 |
| | 4.2.1 Display of the herd migration | 46 |
| | 4.2.2 Display of temperature information | 52 |
| | 4.2.3 Display of snow information | 55 |
| CHAPTER 5 | Conclusion | 59 |
| | 5.1 Future work | 60 |
| | Bibliography | 62 |
| | Appendix A | 65 |
| | Appendix B | 66 |
| | Appendix C | 67 |

CHAPTER 1

Introduction

1.1 Motivation

Traditional graphic design has been concerned with the design of visual forms for static media like paper. As computer-based communication media become more dynamic and interactive, the complexity of visual design problems increase. Three characteristics in electronic media bring new concerns to graphic design: interactivity, personalization and dynamics.

Interactive media allow non-linear access to information. A designer may not be able to control the exact sequence of presented information eventually chosen by the user. This new type of problem requires graphic design in real-time.

Personalized media, such as NewsPeek* developed by the Electronic Publishing group at the MIT Media Lab, are an applications that

* NewsPeek is a system that reads closed-captioned data encoded in the video signal of a news broadcast and compares it with keywords representing a viewer's interest.

demonstrated the need for the dynamic control of the information presentation. Traditionally a newspaper is designed before it is published and read by many readers. However in electronic media, each edition should be designed for each individual, hence it is impossible to previously determine the design of the screen.

The display of dynamic data also presents a new type of design problem. For instance, in dynamic information design, such as weather information, air traffic control or physiological monitoring, the data is unpredictable. Complex real-time information must be clarified depending on the viewer's interest or purpose at a particular time.

In traditional media, each individual design problem is solved directly by the designer. In electronic media, design rules and specifications can be encoded in a computer program and each design problem may be solved by a program.

Ideally, such a computer program would be created for the individual graphic designer, since each one has different methods or styles. General design principles are very difficult to acquire in graphic design. It would be better if the graphic designer can directly create a program without the assistance of a computer programmer. However, no existing system allows a graphic designer to create such a program unless he/she learns a conventional programming language such as C or Lisp.

There is a need for systems that provide individual graphic designers with ways of programming their own design objectives.

What can be considered a natural language for the graphic designer to communicate a design concept? Graphic designers find it difficult to articulate their ideas in verbal form. They usually prefer visual forms, such as drawings or diagrams, to communicate their ideas. One comfortable way for individual graphic designers to program their design styles would be to use visual expressions as a means of communication with the machine.

1.2 Examples as a means of communication

Examples are often used to communicate design ideas, especially in teaching. The design teacher demonstrates examples to illustrate a concept. Students extract the essence of a design concept from a series of examples and apply it to new design problems. The teacher then criticizes the students' design solutions to refine the concept. Students incrementally learn concepts through many examples. In teaching, it is crucial to have an appropriate selection of examples. Students may misunderstand a concept from inappropriate examples. The teacher must clarify the idea and carefully select examples. Finding good examples can be considered as a primary task for teaching design.

Examples are also used as ways of finding a clear description of a design concept. Suppose the designer first has a concept which solves a certain design problem but does not know how to describe it. The designer may be able to find a description of a concept by exploring examples that fit the idea. For instance, consider an example of corporate identity design. The designer makes a specification of the design concept by examining different concrete cases of design problems.

This thesis investigates Example-Based programming as a tool for the graphic designer to program a design concept. Design examples provide ways of communicating the idea as well as feedback to the designer. The system helps a graphic designer, who is also a novice programmer, to generate design rules and specifications in the form of a computer program using design examples. The designer also uses the system as a tool to explore design solutions. The metaphor of this programming technique is teaching by examples: The graphic designer teaches a computer system by demonstrating design examples. The system first watches and remembers the designer's demonstrations, then generalizes them in a computer program. The system also asks questions to the designer when it gets confused about a particular action. In order to avoid being "misunderstood", the designer, as a teacher, also has to clarify a concept and explain it to the system with good examples.

This thesis describes a prototype of the Example-Based Graphical Programming System for graphic design. The following are specific goals for the implementation of the system.

- To build a highly interactive environment that allows the graphic designer to create computer programs without conventional programming. This includes the implementation of an object oriented graphics editor where the designer can demonstrate examples.
- To design mechanisms that can generate a computer program from concrete examples of graphical editing. This includes the mechanisms to record a designer's demonstrations, and to generalize and generate a computer program from the recorded demonstrations. The generation of two programming structures are investigated: conditionals and parameters.

1.3 Related Work

Example-Based programming has been investigated for various applications. The Tinker program [Lieberman 86] was the most influential to this research. Tinker , a Lisp programming environment for beginning programmers, formulates a procedure from demonstrations of concrete examples. Tinker allows beginning programmers to create simple

procedures through fairly complex structures such as recursions and conditionals. Tinker has been tested for various kinds of programs, such as interface design and video game design. Using Tinker, examples are entered in Lisp, whereas in the proposed system, the user demonstrates examples graphically.

Halbert has developed a programming by example system for an office information system, SmallStar [Halbert 81]. The user can create macros by demonstrating graphical interactions with icons. The SmallStar records a series of actions and generalizes concrete examples by replacing constant expressions with parameters. To generate conditionals and loops, the user must explicitly specify the control structure. SmallStar also provides an English-like high level programming language, Cusp*, as a static representation of the program. This allows the user to learn and edit a program created by example.

Peridot [Mayer 88] is an example-based programming environment that has been developed as a rapid prototyping tool for dynamic user interface design. Peridot allows to program a user interactions by demonstration. Using rule-based inference mechanisms, Peridot can guess the intent of the designer. For example, it can decides on a condition to distinguish different demonstrations. The system developed in this thesis does not guess conditions, it requires the users to provide them.

* Customer Programming.

Introduction

Automating graphic design is a current interest of the Visible Language Workshop at the MIT Media Lab. DAIS (Do As I Sketch) [Greenlee 88] is a computer-aided page layout system that allows a graphic designer to specify a design style using a freehand sketch. DAIS uses the designer's sketches as a guideline for search among space allocation. Sketch is a very abstract way of specifying a design style. The proposed system, on the other hand, uses concrete examples as a way of communicating the concept to the machine.

PACKIT [Amari 87] and GRID [Badshah 87] are rule-based expert systems for automatic layout. These system have shown the feasibility of automating some parts of design. Traditional knowledge engineering approaches have been taken to building rule bases: A knowledge engineer interviews graphic designers to capture their design knowledge and encode it in the form of if-then rules. Rules in these systems are then entered by a programmer. The designer could not directly specify individual design styles. The idea of a programming system for the graphic designer is originally inspired by this distancing problem that PACKIT and GRID exhibit.

1.4 The structure of this thesis

This thesis is organized as follows. Chapter 2 introduces the Example-Based Graphical Programming System and a simple scenario of the designer-machine interaction. Chapter 3 describes the implementation of the system. The mechanism of graphical editing, recording, and the generalization technique are described. Chapter 4 shows how the system might assist the designer in designing for electronic media. The use of the system is illustrated in the design of a dynamic display of animal migration.

CHAPTER 2

Overview

This chapter introduces the Example-Based Graphical Programming System. Section 2.1 presents the overall structure of the system and shows how the system is used. Section 2.2 shows a simple interaction with the system and shows how the system functions in the process of designing.

2.1 The Example-Based Graphical Programming System

The Example-Based Graphical Programming System is an interactive computer environment that allows the graphic designer to create a computer program of a design concept. A design concept is translated into a computer program through examples. A design concept, such as the layout of a business card, the color coordination of a textile, or how an element can catch the attention of the viewer, can be programmed with concrete examples.

There are essentially two steps to creating a program using the system (figure 2-1). First, the designer decides on a concept to program and

starts by showing an example by using the *design editor*. The *design editor* is an object oriented graphics editor, like MacDraw, where the graphic designer demonstrates design examples. The design editor provides tools for editing some of the parameters used in graphic design, such as color, font style, position, size or translucency. The attributes of graphical objects are initially set to default values for convenience. The designer can edit a graphical object either by direct or textual manipulation. The system records the designer's graphical editing process.

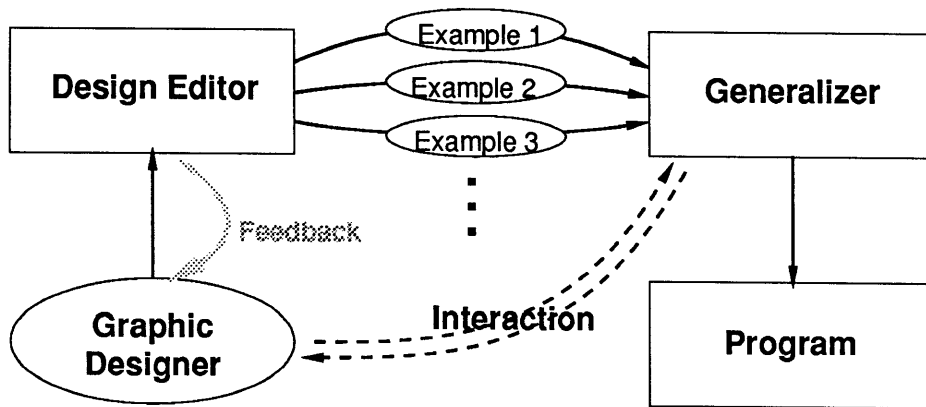


Figure 2-1 A schematic diagram of the Example-Based Graphical Programming System.

The second step is generalization. Once an example is demonstrated, the recorded procedure is passed to the generalizer. The generalizer incrementally creates a general procedure from previously demonstrated examples. Generalization in this system is done through repeated interactions with the graphic designer. The system will query the designer when a decision must be made. Since the system assumes that

Overview

each example illustrates a particular design concept, when a new example conflicts with previous examples, the system tries to learn how to distinguish the new one from the others.

A selection of a unit of design concept depends on the designer. For example, consider a design problem for a page of an electronic magazine. The designer may create a program for *head line* and *author*, and another program for *picture* and *article*. However, the designer can also create a single program which lays out all the elements. A program that lays out a *head line* and *author* can be used in other situations which have no *picture* or *article*. This way of designing is more flexible. If the designer knows there are always four elements on a page, then the program, which lays out four elements, might be appropriate.

The designer can demonstrate as many examples as needed to perform the desired result. In general, the more examples demonstrated the more flexible the program becomes. The designer starts from a specific example. Then, the design procedure can be refined by demonstrating more examples with explanations. Thus the designer does not have to make a program all at once. Rather, the program is refined as the design process goes along. If later in the process, the designer finds a new design problem, he/she can just show a new design solution. A design procedure is always refinable.

2.2 Scenario

A short scenario of the designer's interaction with the system is presented. The task is to design a dynamic temperature display of major American cities, as they might appear on the nightly news (figure 2-2).

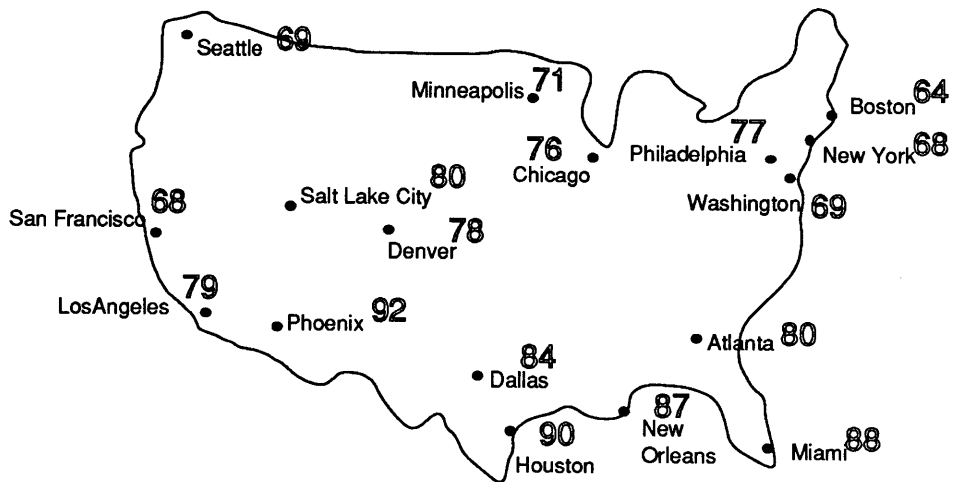


Figure 2-2 *Dynamic temperature display of main American cities: original data.*

Before we start programming, we must briefly describe the structure of the information and graphical objects. Graphical objects are hierarchically organized. Figure 2-3 shows the hierarchy used in this example. Graphical objects are always attached to some information. For example, the Boston-Temperature object is a text object attached to the temperature of Boston. In this system, the information can be a class, which we call an *information type*. The temperature of a city is a type, as

can be the weather or humidity. The information type is defined by the designer. In this scenario, we assume that the information type is already defined.

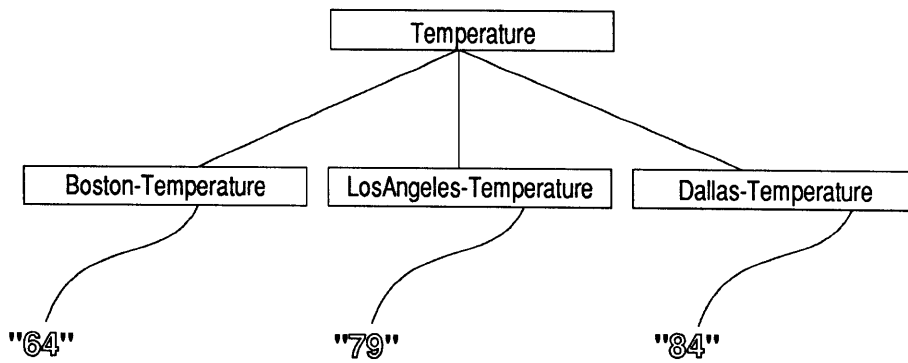


Figure 2-3 A hierarchy of graphical elements.

Suppose we want to make a program (design procedure) that displays temperature information as follows: The color of a text string representing the temperature is Yellow by default, Orange when the temperature is greater than 70 degrees and Red when the temperature is greater than 80 degrees.

The first step is to tell the system to create a new procedure by selecting the menu item **NEW**. The system then asks the designer to enter a

name of the new procedure (Appendix A):

```
Enter procedure name  
=> display-temperature*
```

Now the system is ready to learn a new design procedure, "display-temperature." The second step is to select a graphical object used as the argument of the procedure. The menu item **ADD ELEMENT** is used to choose a new element. Suppose we have chosen a Boston-Temperature object for the first demonstration, with a temperature value lower than 70 degrees. We now use the design editor to edit the graphical object. At this point, the designer usually tries out design solutions to test and clarify his/her ideas.

In order to record an example, we click the menu item **SHOW EXAMPLE**. We then set the color of the Boston-Temperature object to Yellow (figure 2-4) using the design editor.

* Throughout this thesis, the symbol "=>" is used to indicate a prompt for the user's input and the bold type represents the text typed from the keyboard by the user.

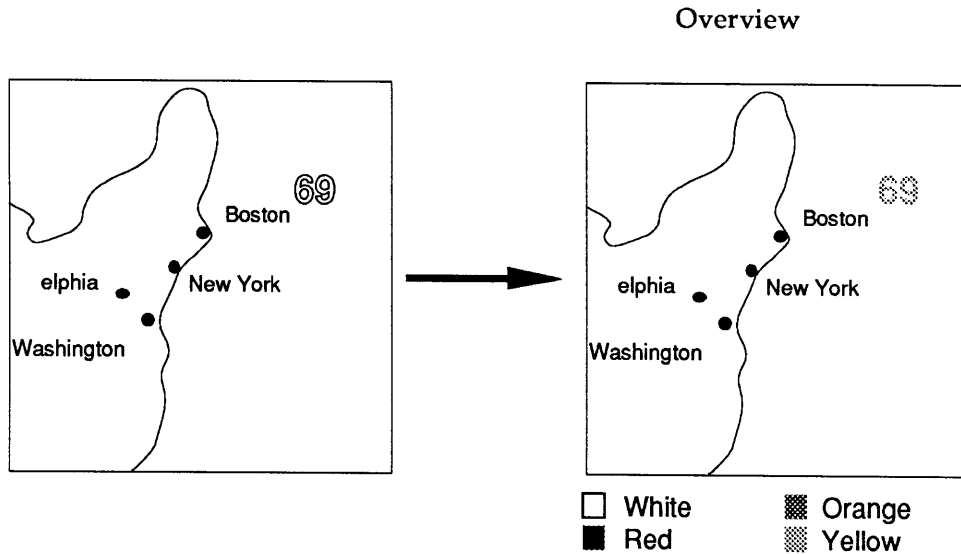


Figure 2-4 Set the color of the Boston-Temperature object to Yellow.

During the example, the editing operation is recorded by the system.

By clicking the menu item **DONE**, we tell the system that the example is over. The recorded example is then passed to the generalizer in the form of a design procedure. At this point, the specific design procedure is generalized so that it can be applied for other objects that belong to the same *information type*. The generalization mechanism is described in detail in chapter 3.

Now, "display-temperature" has generalized the user's actions by applying the procedure to a different graphical object. We replace the Boston-Temperature object by a LosAngeles-Temperature object. The LosAngeles-Temperature object belongs to the same *information type*. The "display-temperature" procedure can also be applied. The current version

Overview

of the design procedure, "display-temperature" sets the color of the LosAngeles-Temperature object to Yellow.

However, since the temperature in LosAngeles is 79 degrees, which is higher than 70, we want the color of the LosAngeles-Temperature object to be Orange. Hence we show a new example that sets the color of LosAngeles-Temperature to be Orange. The system then generalizes the new demonstration along with the previous "display-temperature" procedure. Here, since the system assumes that both examples cover the same concept, the system finds a conflict and asks:

This case matches the operation:

```
(send Temperature :set-color Yellow).
```

However you demonstrated a different example

How do you distinguish

```
(send Temperature :set-color Orange) from
```

```
(send Temperature :set-color Yellow)?
```

The system uses a lisp-based object oriented language to represent and communicate design procedures. In the current system, the designer must learn a few fairly easy statements of the language. The question given here is read as follows: "This situation matches to the operation of setting the color of the object to Yellow. However you demonstrated a different example. How do you distinguish the new example, which sets the color to

be Orange, from the previous example?" The system's internal representation is described in detail in the next section.

We now must describe a condition that differentiates the new example from the previous one. We explain that the value of a temperature object is greater than 70:

```
=> (> (send Temperature :value) 70).
```

Now "display-temperature" has been programmed to make a decision. It sets the color of any temperature objects to Orange if its value is higher than 70, otherwise it can set the color of the object to Yellow.

Let us demonstrate one more example here. Suppose we have set the color of the Dallas-Temperature object to be Red. The system generalizes the new example with the previously generalized procedure, and asks: "Since the temperature of the object is higher than 70, the expected action is to set the color of the object to be Orange. However you demonstrated a different action. How do you distinguish the new example, that sets the color of an object to be Red, from the previous example, that sets the color of an object to be Orange?"

Overview

```
(> (send Temperature :value) 70) is TRUE
Therefore the situation matches the operation:
      (send Temperature :set-color Orange).
However you demonstrated a different action.
How do you distinguish
      (send Temperature :set-color Red) from
      (send Temperature :set-color Orange)?
```

Since the temperature of Dallas is greater than 70, the system first finds that this case matches the action which sets the text color to be Orange for the Dallas-Temperature object. However, the demonstrated action is different. Therefore, the system asks the designer how to distinguish the new example from the previous examples. We explain that the value of the temperature object is greater than 80 as a condition to distinguish the new design solution:

```
=> (> (send Temperature :value) 80).
```

Now we have programmed a satisfactory design procedure. We assume that the procedure, "display-temperature", is already a part of the temperature display program. Figure 2-5 shows the result of applying the procedure to all the temperature objects.

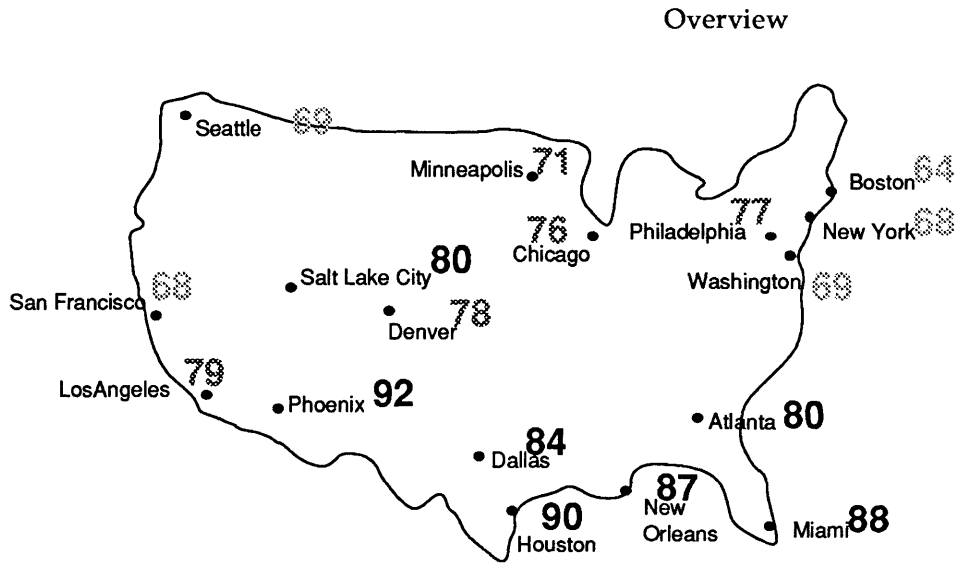


Figure 2-5 Dynamic temperature display after the procedure is applied.

Suppose we have tested the design procedure with dynamic temperature data and we noticed that the Phoenix-Temperature object has a value of 118 degrees in a particular scene. This makes us realize that we want to differentiate the object when the temperature is very high. We might decide to use a bigger font to indicate that fact. The fontsize of the Phoenix-Temperature object is set to 32 point as an example (figure 2-6).

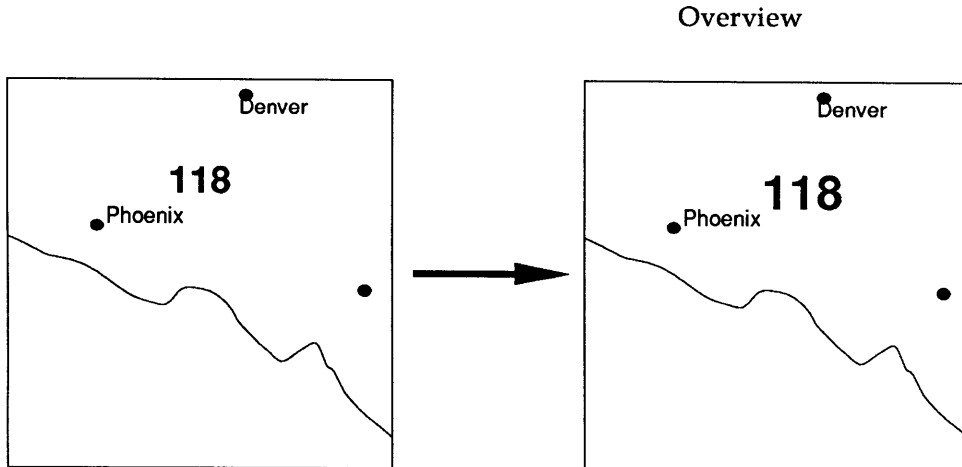


Figure 2-6 Set the fontsize of Phoenix-Temperature object to 32 point.

The system tries to generalize and asks the question:

```
The situation does not match the operation: set-fontsize
However you demonstrated an example.
How do you distinguish
(send Temperature :set-fontsize 32)?
```

Then, we explain that the value of the temperature object is greater than 110:

```
=> (> (send Temperature :value) 110).
```

Figure 2-7 shows the scene when we apply the new display-temperature procedure to the dynamic information.

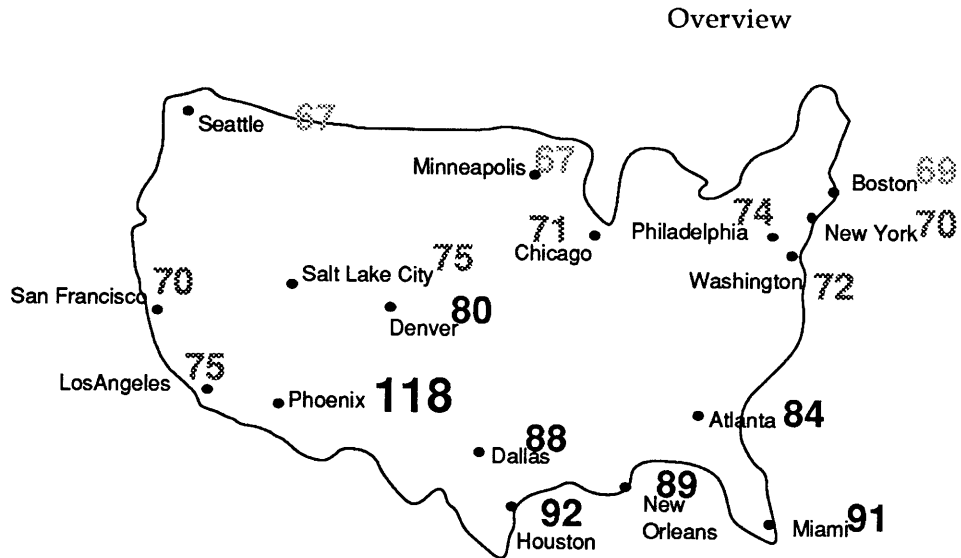


Figure 2-7 Dynamic temperature information after the new procedure is applied.

We have looked at how the system is used from a designer's standpoint. However, a real design session may not be as linear as we presented. There is usually a planning session where the designer explores different cases before showing different examples. In other words, the designer has to plan what examples must be demonstrated to create a procedure that performs his/her objectives. The system also has the potential of making the designer think about his/her own decisions. The designer may discover new problems or solutions by applying the current procedure to a new case.

CHAPTER 3

Implementation

We have seen the operation of the system from a user's standpoint in the previous chapter. This chapter describes the implementation of the Example-Based Graphical Programming System referring to the scenario presented in chapter 2. Section 3.1 describes the representation used for design procedures and the basic recording mechanism. Section 3.2 describes the generalization techniques used in the system.

3.1 Representation and the basic recording mechanism

This section shows how the system is structured. The basic data structures and the recording mechanism are described.

3.1.1 Graphical objects (design elements)

In the *design editor* two classes of graphical objects are used: *text objects* and *image objects*. These are defined as follows:

Implementation

```
Text object
  :top
  :bottom
  :left
  :right
  :width
  :height
  :string
  :fontname
  :fontstyle
  :fontsize
  :color
  :translucency
```

```
Image object
  :top
  :bottom
  :left
  :right
  :width
  :height
  :translucency
```

Attributes of the design object can be updated either through a menu or by direct manipulation. (Appendix B)

3.1.2 Representation of the manipulation

The design editor automatically generates a *procedural description* of each editing operation. The *procedural description* is a textual representation of the graphical manipulation. A lisp-based object oriented programming language is used to represent the procedural descriptions. Suppose we have a graphical object, Boston-Temperature, which is an instance of the text class object. The expression:

```
(send Boston-Temperature :set-color Yellow),
```

means "send a message to the Boston-Temperature object to set its color to Yellow."

3.1.3 Recording mechanism and the design procedure

The *design procedure* is a sequence of *procedural descriptions* generated and recorded by the design editor. To see how the design procedure is recorded, recall the demonstration of the display-temperature procedure. When we set the color of the Boston-Temperature object to Yellow, the demonstration is recorded as follows:

```
procedure = ((send Boston-Temperature :set-color Yellow)).
```

If the designer then sets the fontsize to be 32 point, the design procedure is updated:

```
procedure = ((send Boston-Temperature :set-fontsize 32)
             (send Boston-Temperature :set-color Red))
```

The following data structure represents the *design procedure*:

```
Design procedure
  :lisp-function
  :arguments
  :body
  :current-elements
  :current-example
```

The most recent *design procedure* is stored in the slot ":current-example." This slot is updated by the designer's each graphical manipulation. The

Implementation

design elements used in the demonstration are stored in the slot ":current-elements." Once the generalization is done, a generalized design procedure is stored in the slot ":body." The slot ":arguments" stores variables generalized from specific graphical objects. The slot ":lisp-function" stores the function definition, which is used internally.

The following is a state of the design procedure after the second example is given.

```
Design procedure
  :name          display-temperature
  :lisp-function (defun display-temperature (Temperature)
                  (send Temperature :set-color Yellow))

  :arguments     (Temperature)

  :body          ((send Temperature :set-color Yellow))

  :current-elements (LosAngeles-Temperature)

  :current-example ((send LosAngeles-Temperature :set-color Orange))
```

Notice the LosAngeles-Temperature is in the ":current-elements" slot and the generated procedural description is stored in the ":current-example" slot. The generalized first example is in the slot ":body". The element used in the first example is in the slot ":arguments" as an argument to the program.

3.2 Generalization

Simple recording of a demonstration generates a program that always performs exactly the same action. This program can not be applied to any other objects or in any other situations. By generalizing recorded design procedures, a more flexible program can be generated.

Generalization is defined as the problem of finding important characteristics from a number of specific observations. In this system, the generalization creates a general procedure from one or more specific design examples, that can be applied for other similar kind of design problems. A number of generalization techniques have been developed for programming by example.

One way is to use an inference mechanism to guess what the user means through examples and then generate a program. For example, Peridot [Mayer 88] can infer when to make conditionals and loops by using rules. This method is very easy to use for non-programmers. One problem of inference-based systems is that their guesses are not always accurate. The inference also does not work if the system lacks sufficient knowledge. Mayer reports that Peridot seems to have most of the rules necessary for interface design, unless the graphic designer invents a new graphical style. Other systems require a series of examples for the generalization. This method is tedious when the user knows what he/she exactly wants the program to do.

Implementation

An other way is to require some user input during or after the demonstration is recorded. This method does not have an inference mechanism. For example, SmallStar [Halbert 81] requires the user to specify what to parameterise when an example is recorded. To create conditionals and loops, the user must explicitly specify the control structures. A program generated using this method can perform exactly what the user tells it to do, while the inference-based method uses examples to create a program that performs what the users means. A disadvantage of this method is that the user must understand programming concepts. However, this method can be more flexible, so that the kinds of program the user can produce are not limited by the inference engine. This method also does not require the user to repeat examples.

The generalizer developed in this thesis automatically generates parameters for graphical elements in specific design examples. As described above, the designer demonstrates an example based on the type of information, such as graphical behavior of temperature for weather display, or placement of page number for a magazine.

In order to have a program make design decisions, conditional statements are needed. The system does not use inference to create conditions: rather, the designer has to provide conditions. Explaining each example is more suitable for graphic design because the designer usually knows the purpose, or reason, for a particular decision. Without

the designer's specification, it is hard for the system to guess the designer's intention from examples. There is also a need for a series of examples to guess appropriate conditional statements correctly, and this is not guaranteed to be accurate. The following sections describe the generalization mechanisms in detail.

3.2.1 Creation of parameters (Changing constants to variables)

A procedural description that has a constant can be generalized by turning a constant into a variable. A variable can stand for any constant, which means it is more general than a constant. The system generalizes a specific design object to a variable by climbing the hierarchy of design elements (figure 3-1).

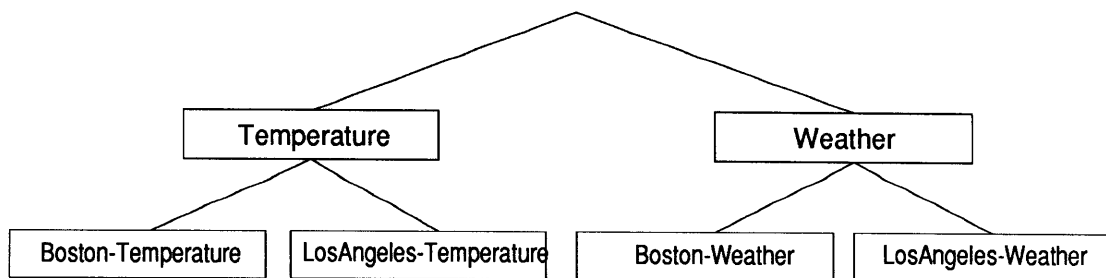


Figure 3-1 A class of design elements.

This type of generalization is called "IS-A hierarchy" generalization and is used to avoid over generalization. [Charniak 84] When the system

Implementation

transforms a specific object into a variable, the system tries to keep the generalization as minimal as possible. Using this technique, the system can remember whether the design procedure is created for a particular information type.

Suppose we want to demonstrate an example of the concept, "catch the viewer's attention", by setting the color of the text to Red. The following two demonstrations:

```
(send Boston-Temperature :set-color Red) and
(send LosAngeles-Temperature :set-color Red)
```

can be generalized into

```
(send Temperature :set-color Red).
      *where Temperature is a variable stands for
      any Temperature objects.
```

The following demonstrations:

```
(send Boston-Temperature :set-color Red) and
(send Boston-Weather :set-color Red)
```

can be generalized into

```
(send X :set-color Red)
      *where X is a variable stands for any
      graphical objects.
```

Implementation

In the system, the first level of generalization is automatically done even with one example. A specific graphical object in a design procedure is generalized into a variable which stands for any object that belongs to the same information type. For example, if a design example is shown with a specific temperature information object, such as Boston-Temperature, the design procedure is generalized such that the same operation can be applied for any other temperature objects.

3.2.2 Creation of conditional structures (Adding options)

An example can also be generalized by adding options. When the designer demonstrates two or more different specifications for the same attribute, the system considers them as alternate solutions. Having alternate actions in a linear procedure necessitates a conditional structure to select one action from others. The system uses "if" statements to generate conditional structures. For example, consider the scenario of chapter 2. After the first two examples are shown, the system generates a conditional structure with the condition provided by the designer:

```
(if (> (send Temperature :value) 70)           ;condition
    (send Temperature :set-color Orange)      ;then
    (send Temperature :set-color Yellow)).    ;otherwise
```

Implementation

This is more general than the design procedure generalized from the first example:

```
(send Temperature :set-color Yellow).
```

When the system finds that an alternate action is introduced in the generalization process, it asks the designer to provide a condition that distinguishes one example from another. Then the system creates a conditional structure in a design procedure. The designer can specify either graphical conditions or application based conditions. Graphical condition is a situation independent from information, such as overlapping. Application based conditions is a situation depending on the content of information, such as the temperature value.

Two types of generalization techniques have been described in previous sections. However, the scope of the generalizations was not clearly described. In the generalizer, variables are created only for constants which stand for the graphical object. The other constants, such as color, typeface or fontsize, can not be variablized since these values are intentionally selected by the designer. These constants are generalized by adding options.

3.2.3 Generalization as a search

In this thesis, we consider generalization as a search problem. The search space is defined as a set of partially ordered hypotheses, based on the relation "more-specific-than." [Mitchell 78,79] Generalization is performed for each graphical manipulation, such as "set-color" or "set-fontsize." The design procedure is a conjunction of all the generalized graphical manipulations. Figure 3-2 shows a simple "general-to-specific" search space for "set-color" using the example shown in chapter 2. To keep the search space small, we present the object with only one attribute, color, and the choice of color is limited to Red, Orange or Yellow.

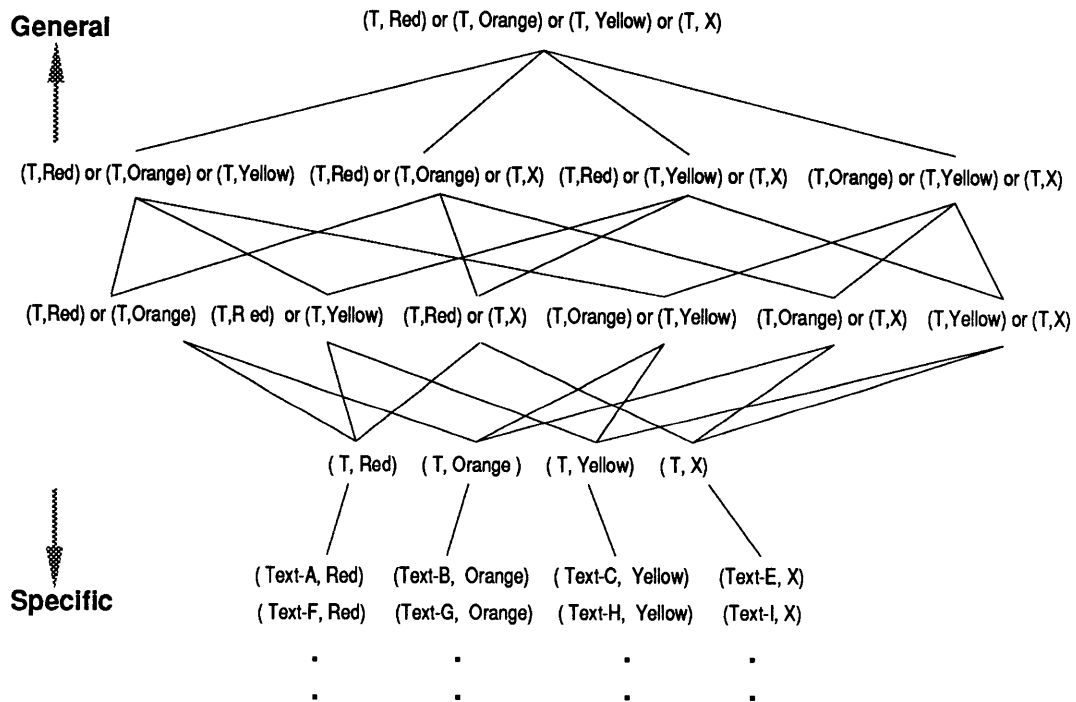


Figure 3-2 A simple general-to-specific search space.

Implementation

In the diagram, "Text-A~I . . ." are specific graphical objects and "T" represents a variable stands for Temperature. The most specific level in the diagram contains the examples demonstrated by the designer, such as (Text-A, Red) * A specific example becomes more general when the specific object is changed to a variable, such as (T, Red), which is translated as "set the color of any temperature object to Red." Two or more examples can generate optional actions. For example, if (Text-A, Red) and (Text-B, Orange) are demonstrated, the system can climb up the search space and generalize them into (T, Red) or (T, Orange). In the design procedure, this becomes a conditional structure with a condition provided by the designer. (T, Red) or (T, Orange) is translated as "set the color of an object to Red or Orange depending on the condition."

The "X" used in this diagram means "do not perform" the action "set-color." In other words, the action "set-color" is not demonstrated by the designer. "Do not perform" an action in a program means there is no expression which specifies the action in a program. Therefore, we do not see any expression in the design procedure. "Do not perform" is an imaginary action that is used by the generalizer. For example, if the designer sets Text-A to Red as in the first example and does not specify the color for the second example, the system automatically generates (Text-A, X), and generalizes them. The system climbs the search space and finds (T,

* For brevity, the expression, (Object, Value), is used to represent a procedural description, (send Object :set-color Value).

Red) or (T, X) as a result, which is translated as "set the color of an object to Red or do not set, depending on a condition."

3.3 An example of the the generalization process

Let us look at the generalization process with the scenario presented in chapter 2. Suppose that the system has already seen the first example:

```
(send Boston-Temperature :set-color Yellow)
```

Since the system tries to keep the design procedure as specific as possible, the system generalizes the design procedure to be:

```
(defun display-temperature (Temperature)
  (send Temperature :set-color Yellow))
```

Notice that Boston-Temperature is generalized into Temperature and a lisp function is constructed.

Now, consider the second example:

```
(send LosAngeles-Temperature :set-color Orange)
```

is demonstrated. The system then finds the current design procedure is too specific and it needs to be generalized. The system first finds that an option must be added:

Implementation

```
(or (send Temperature :set-color Orange)
    (send Temperature :set-color Yellow))
```

It then asks the designer to distinguish

```
(send Temperature :set-color Orange) from
(send Temperature :set-color Yellow).
```

Suppose we have already answered that this is because the temperature is above 70:

```
=> (> (send Temperature :value) 70)
```

The system creates a conditional structure, updating the design procedure as follows:

```
(defun display-temperature (Temperature)
  (if (> (send Temperature :value) 70)
      (send Temperature :set-color Orange)
      (send Temperature :set-color Yellow)))
```

Here is the third example. We set the color of the Dallas-Temperature object to Red:

```
(defun display-temperature (Temperature)
  ((send Dallas-Temperature :set-color Red)))
```

The system first examines the condition against this case and finds that the action:

```
(send temperature :set-color Orange)
```

Implementation

has to be performed, since the temperature value of the new object is above 70. However, the new example makes the system realize that the result is too specific and needs to be generalized. The system then asks how to distinguish

```
(send Temperature :set-color Red) from
(send Temperature :set-color Orange).
```

The condition:

```
=> (> (send Temperature :value) 80)
```

updates the design procedure to be:

```
(defun display-temperature (Temperature)
  (if (> (send Temperature :value) 70)
      (if (> (send Temperature :value) 80)
          (send Temperature :set-color Red)
          (send Temperature :set-color Orange))
      (send Temperature :set-color Yellow))))
```

Figure 3-3 shows the resulting search space after the third example. Arrows show the path the generalizer has taken.

Implementation

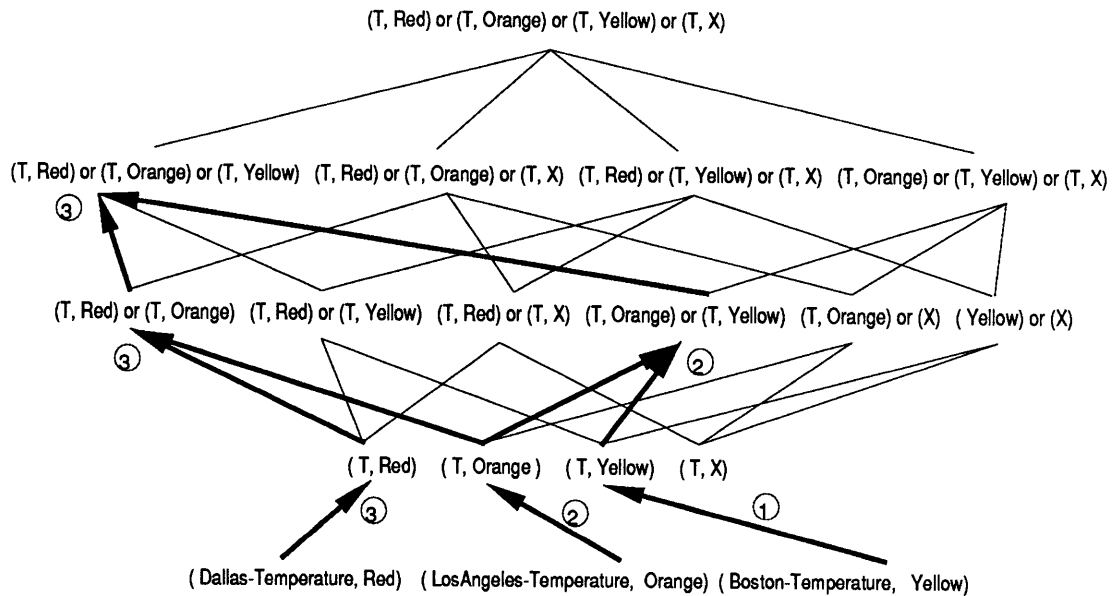


Figure 3-3 The search space after the third example is generalized.

Finally we show a new example that sets the size of a text object to be 32 point when the temperature value is greater than 110. The new example is shown with the Phoenix-Temperature object:

```
((send Phoenix-Temperature :set-color Red)
 (send Phoenix-Temperature :set-fontsize 32))
```

The system then tries to generalize and find that the previous examples do not use the set-fontsize manipulation in a design procedure. However, the new example tells that the fontsize must be set to 32 point. The system then asks how to distinguish the new example:

```
(send Temperature :set-fontsize 32).
```

Implementation

The answer:

```
=> (> (send Temperature :value) 110)
```

updates display-temperature to be the following:

```
(defun display-temperature (Temperature)
  ((if (> (send Temperature :value) 70)
    (if (> (send Temperature :value) 80)
      (send Temperature :set-color Red)
      (send Temperature :set-color Orange))
    (send Temperature :set-color Yellow))
  (if (> (send Temperature :value) 110)
    (send Temperature :set-fontsize 32)))
```

Notice that a conditional structure has been created. In a previous examples, the fontsize was not specified and treated as "do not perform," The new conditional structure can distinguish "do not perform" and "set the fontsize to 32 point." This conditional structure is created separately from the condition clause used for "set-color", since this is a different graphical manipulation. Figure 3-4 shows the resulting search space for "set-fontsize" after the fourth example is generalized.

Implementation

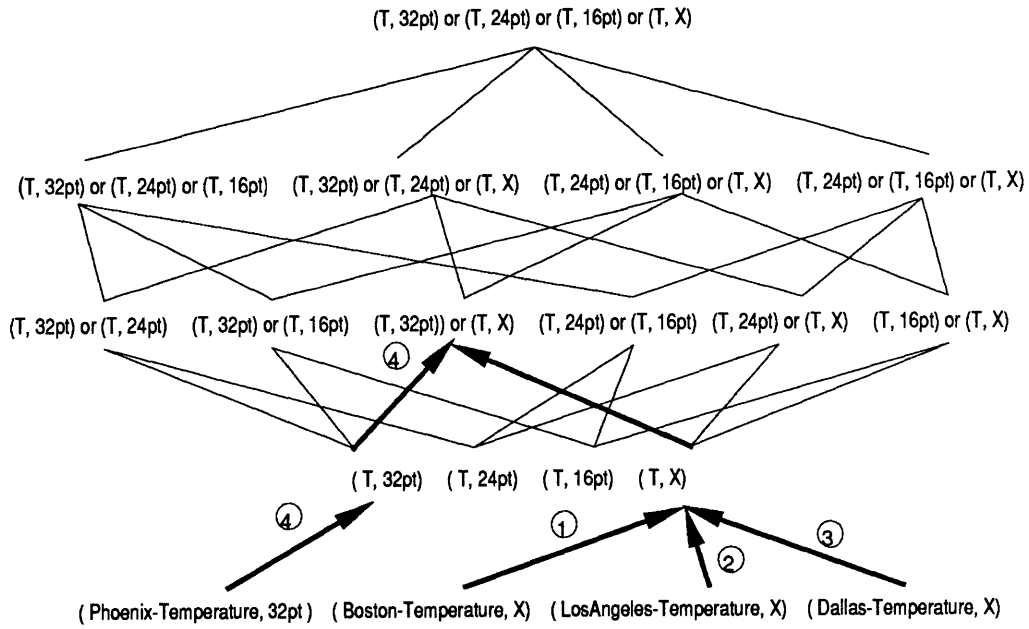


Figure 3-4 The search space for "set-fontsize" after the fourth example is generalized.

CHAPTER 4

Migration graphics

We have seen the mechanisms for recording and generalizing the designer's demonstrations and simple examples have been presented that illustrate how the system works. In this chapter, the Example-Based Graphical Programming System is used as a tool for designing the dynamic display of mule deer migration.* Appendix B shows the interface of the Example-Based Graphical Programming System specially customized for the display of the migration.

Design procedures are programmed through examples that show different design solutions to important situations. First we graphically set a situation, then we demonstrate a solution using the *design editor*. For example, if we want to program a design procedure to change the color of a text object when a herd of mule deer is in a green grass area, we place the object in the correct area on the map. Then the right color of object is set as an example. In this chapter, more complex examples of *design*

* In Dynamic Information Display, a research effort being developed at the Visible Language Workshop, the graphical representation of *time and space varying* information is investigated to enhance communication.

Migration graphics

procedures are presented. Section 4.1 briefly introduces the requirements for the dynamic display of mule deer migration information. Section 4.2 covers an interactive session of programming-by-example.

4.1 Design requirements

Before looking at a programming session, let us look at the design requirements for the dynamic migration graphics. Mule deer are likely to develop a seasonal return migration on the slope of hills and mountains. The autumn migration begins sometime in October with the first significant snow fall. [Baker 78] As a reference, figure 4-1 shows examples of static views of the migration.

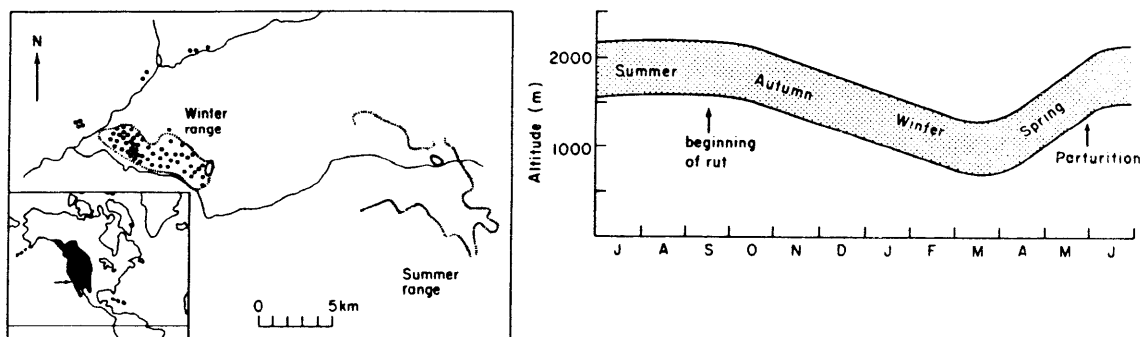


Figure 4-1 Static visualization of mule deer migration.

The dynamic information includes the movements of herds of mule deer, snow and temperature. The environmental information includes feature types, such as rivers, grass and altitude. The design task is to graphically represent the relation between dynamic and environmental information.

4.2 A session with the system

4.2.1 Display of the herd migration

We begin by programming the behavior of a graphical object as the visualization of a herd of mule deer in relation to the environment. We assume that we already constructed a text object, "HERD 1", to represent the herd information, and the procedure to place "HERD 1" in the right position is previously programmed. We want a graphical object to indicate the topographic feature type near the herd. Let us create a design procedure, "display-mule-deer", as follows: The size of the text represents the Altitude of the terrain: 32 point represents HIGH (> 1500ft). 24 point represents MEDIUM (1000 ~ 1500). 16 point represents LOW (< 1000ft). The color of the text represents the features of the terrain: Green represents Grass; Blue represents Rivers; Yellow represents other features.

We start from the most general situation: "HERD 1" is neither on river nor grass. "HERD 1" is placed where the Altitude is 1340.

Migration graphics

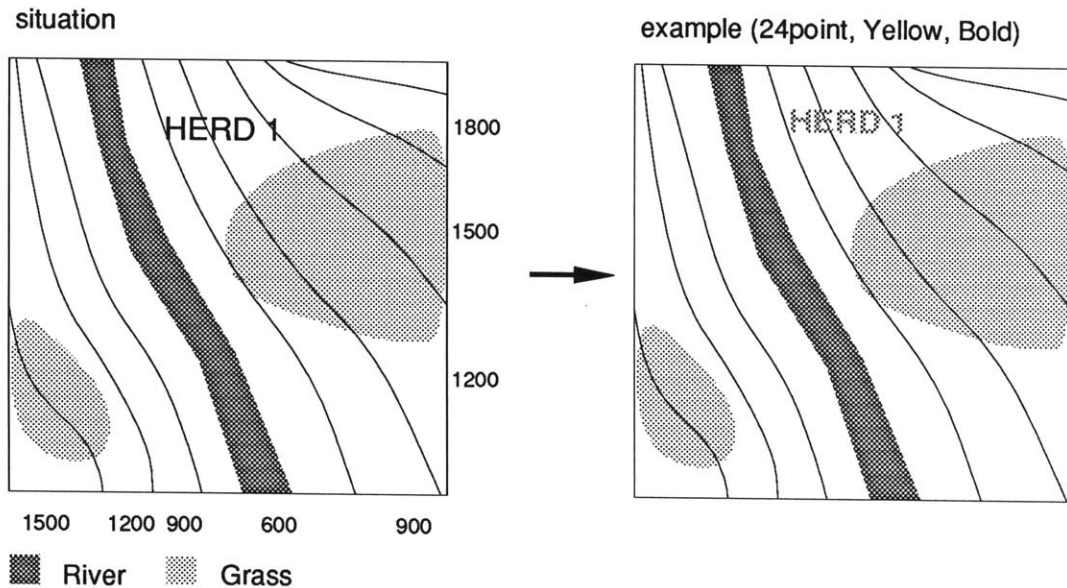


Figure 4-2 The first demonstration for "display-mule-deer."

We demonstrate a solution to the first situation (figure 4-2): We set the color to Yellow, the fontsize to 24 point and the fontstyle to Bold. At this point, the design procedure, "display-mule-deer", has been generalized to perform exactly the same procedure as the first example. However, the procedure now can be applied for other mule-deer objects, since "HERD 1" in the example is variablized into its information type.

The second example shows the behavior of "HERD 1" when it is on Grass. Since we cannot find a place where the Altitude is higher than 1500 feet or less than 1000 feet, the fontsize is still within the range of 24 point (1389ft). We demonstrate the design solution for the second situation (figure 4-3): We set the color to Green and leave the fontsize at 24 point. The fontstyle remains Bold.

Migration graphics

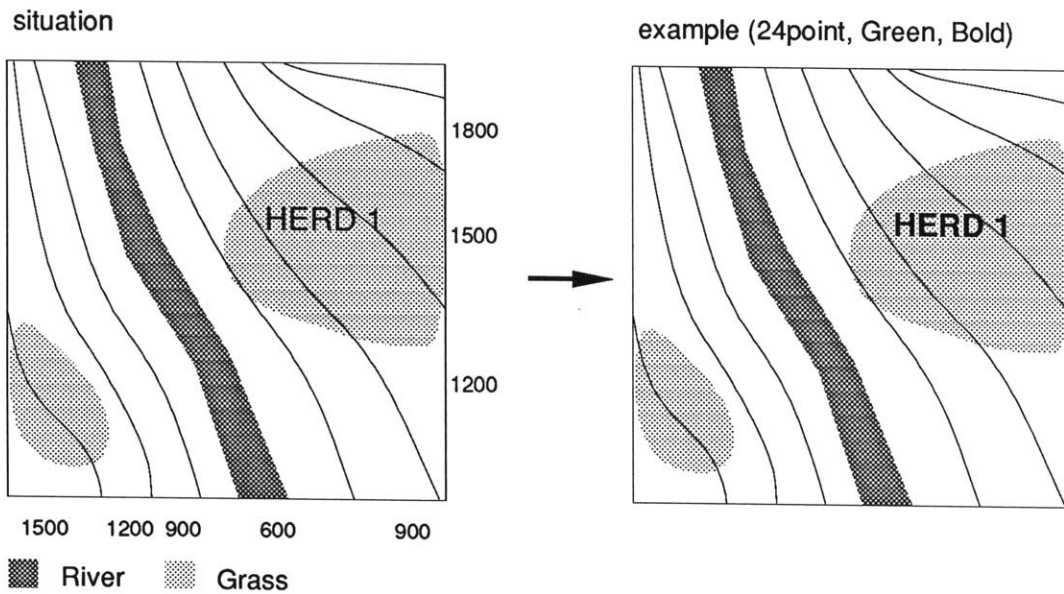


Figure 4-3 The second demonstration for "display-mule-deer."

After we have shown the example, the system now tries to generalize the first and the second examples. The system first asks:

```
This case matches the operation:  
  (send mule-deer :set-color Yellow).  
However you demonstrated a different example  
How do you distinguish  
  (send mule-deer :set-color Green) from  
  (send mule-deer :set-color Yellow)?
```

We explain that this is because the herd is on the grass:

```
=> (send mule-deer :on-grass?).
```


Migration graphics

Since the fontsize and the fontstyle are exactly as in the previous example, the system accepts the rest of the new demonstration as general ideas.

In the system, a set of predefined messages for objects are used to access information related to a graphical object. Figure 4-4 is a list of messages currently defined for the migration graphics. Using these messages the designer can tell the system important situations.

```
on-grass?  
on-river?  
on-inhabitants?  
any-overlapping?  
overlapping-to  
distance-to  
any-object-within?  
get-altitude  
snow-fall
```

Figure 4-4 Predefined messages.

The third situation places "HERD 1" in the River area. In this case the altitude is set to 790ft. The design solution to the third situation is to set the color to Blue. The fontsize is set to 16 point. The fontstyle is Bold (figure 4-5).

Migration graphics

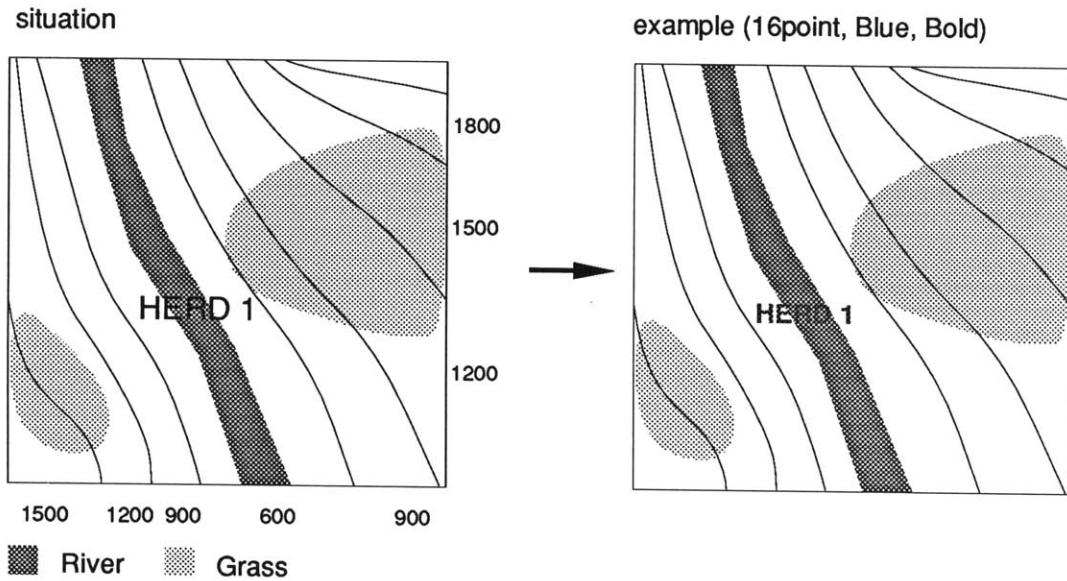


Figure 4-5 The third demonstration for "display-mule-deer."

In order to generalize the third example from the current best hypothesis, the system asks:

```
This case matches to the operation:  
    (send mule-deer :set-color Yellow).  
However you demonstrated a different example  
How do you distinguish  
    (send mule-deer :set-color Blue) from  
    (send mule-deer :set-color Yellow)?
```

We explain that this is because the mule deer is in a river area:

```
=> (send mule-deer :is-river-there?).
```

Then the system asks another question:

Migration graphics

This case matches to the operation:

```
(send mule-deer :set-fontsize 24)
```

However you demonstrated a different example

How do you distinguish

```
(send mule-deer :set-fontsize 16) from
```

```
(send mule-deer :set-fontsize 24)?
```

We explain that this is because the altitude is lower than 1000 ft.

```
=> (< (send mule-deer :get-altitude) 1000)
```

The only situation left is when a herd is in an area where the altitude is higher than 1500 feet. We set the last situation and show the new example: The color of the object is set to be Yellow. The fontsize is set to be 32 point. The fontstyle is set to be Bold. (figure 4-6)

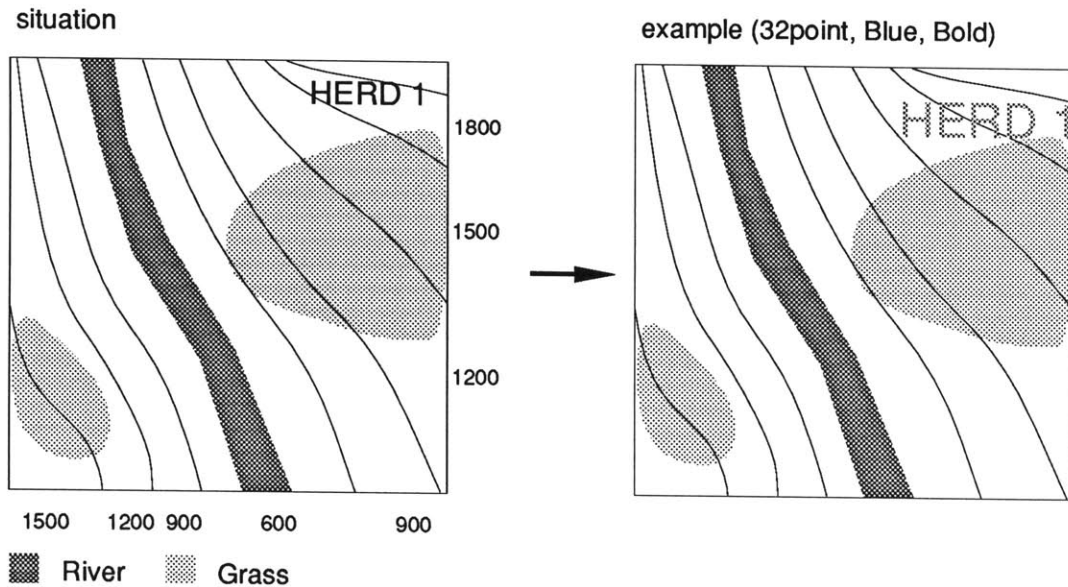


Figure 4-6 The fourth demonstration for "display-mule-deer."

The system tries to generalize the new example and asks:

This case matches the operation:

```
(send mule-deer :set-fontsize 32)
```

However you demonstrated a different example

How do you distinguish

```
(send mule-deer :set-fontsize 24) from
```

```
(send mule-deer :set-fontsize 32)?
```

We explain that this is because the altitude of the current position is higher than 1500 ft:

```
=> (> (send mule-deer :get-altitude) 1500).
```

We have shown how the design procedure, "display-mule-deer", is programmed. Appendix C shows the result of applying this procedure to a simulated dynamic information with the other design procedures presented in the following sections.

4.2.2 Display of temperature information

In the previous section, we have programmed the display of dynamic information in relation to its environment. This section introduces a design procedure which has to consider the relation between two dynamic information: temperature and the herd migration. Suppose that the temperature information will be measured in different locations and needs to be visualized. However, if we display all temperature

Migration graphics

information on the map, the display gets rather cluttered. A design solution to this problem is to use translucency to reduce the contrast of objects. Since the herd information is the more important, the translucency of the temperature object at a specific location according to the distance from the mule-deer object to that point. The design procedure, "display-temperature", is programmed as follows: The translucency of the temperature object is set to be high (80%) when any herd of mule deer is within 2 miles, otherwise low (30%).

We start demonstrating a general situation: No herd of deer is within 2 miles. We choose a temperature object, Temperature-1, from the display and set the situation. After testing several different translucency, we decide and demonstrate an example of setting the translucency of Temperature-1 to 80% (figure 4-8).

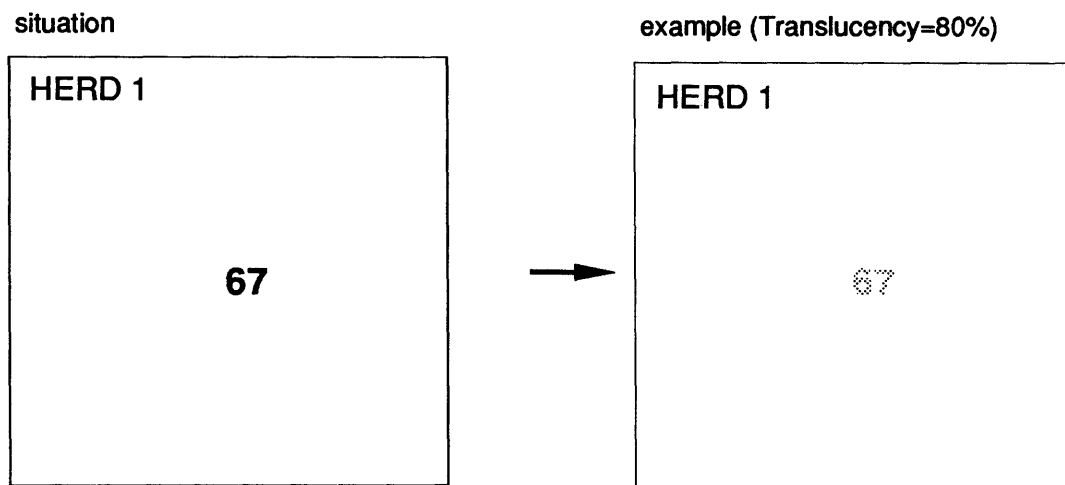


Figure 4-8 The first demonstration for "display-temperature."

Migration graphics

Now we need to show an example when some herd of mule deer is within 2 miles. First, we set this situation by placing "HERD 1" close to Temperature-1. Then, we demonstrate a solution to the second situation; The translucency of Temperature-1 is set to 30 % (figure 4-9).

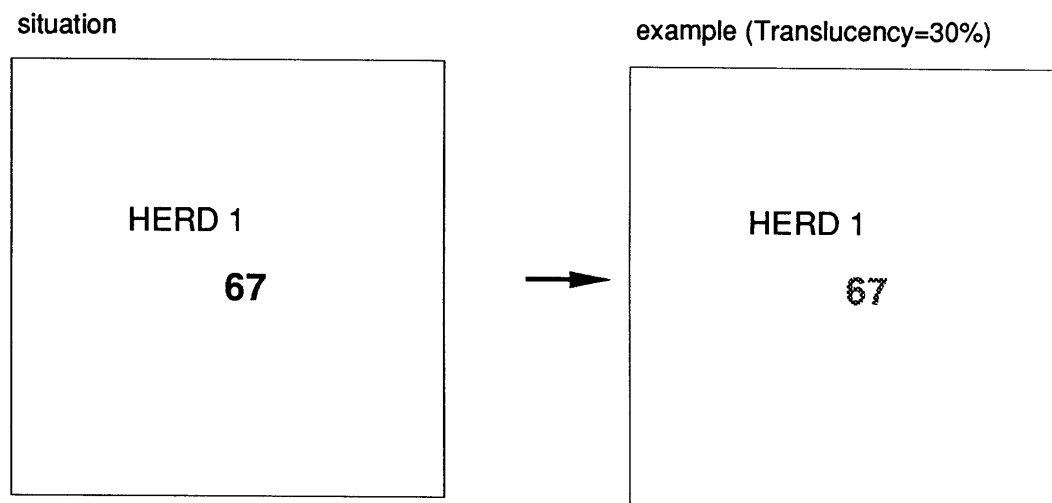


Figure 4-9 The second demonstration for "display-temperature."

The system tries to generalize a new example from the first one and asks :

This case matches the operation:

```
(send Temperature :set-trans 80).
```

However you demonstrated a different example

How do you distinguish

```
(send Temperature :set-trans 30) from
```

```
(send Temperature :set-trans 80)?
```

We explain that this is because there is some deer object within 2 miles:

```
=> (send Temperature  
      :any-object-within mule-deer 2).
```

Now the design procedure, "display-temperature", has been programmed to set the translucency of the temperature object based on the distance to the herd of deers. (Appendix C)

4.2.3 display of snow information

This section introduces the use of two graphical objects into one design procedure to visualize snow information. A picture of a snow cloud is used to represent the area covered by the cloud. Text is used to represent the amount of snow fall. The translucency of the picture is increased slightly when it overlaps with other objects, in order to make the information behind visible. The color of the text is set to Red when the amount of snowfall is large. The precise description of the design procedure, "display-snow", is the following: The translucency of the picture is set to 60% when it overlaps with a deer object. Otherwise the translucency of the picture is set to 30%. The color of the text is set to Red when the amount of snow fall is greater than 6 inches/hour. Otherwise the color of the text is set to Black.

We begin by setting the situation for a general case. Here, we manually set the value of information. We set the amount of snow fall to be 1 inch/hour. We demonstrate a design solution as follows: The

Migration graphics

translucency of a picture is set to 30% The color of the text is set to Black.
(figure 4-10)

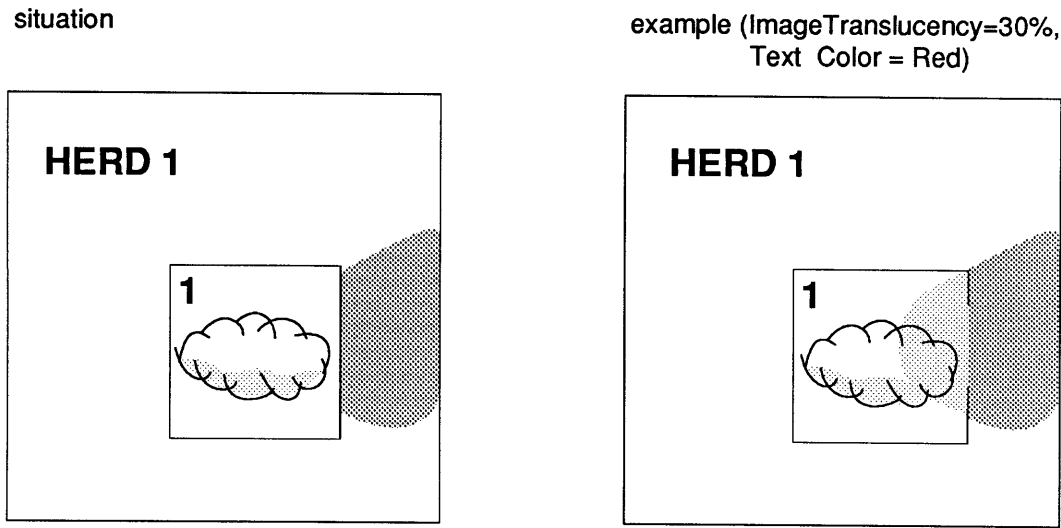


Figure 4-10 The first demonstration for "display-snow."

Now we set a new situation where the amount of snow fall is 7 inches/hour. We demonstrate a solution to the second situation. The translucency of the picture is set to be 60% so that "HERD 1" can be visible. The color of the text is set to Red since the amount of snow fall is more than 6 inches/hour. (figure 4-11)

Migration graphics

situation

example (ImageTranslucency=60%,
Text Color = Red)

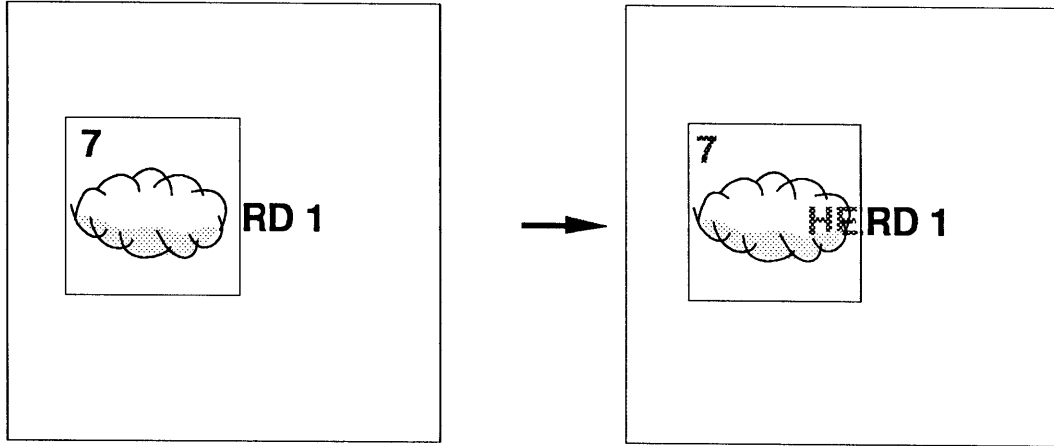


Figure 4-11 The second demonstration for "display-snow."

The system tries to generalize and asks:

This case matches the operation:

```
(send snow-text :set-color Black).
```

However you demonstrated a different example

How do you distinguish

```
(send snow-text :set-color Red) from
```

```
(send snow-text :set-color Black)?
```

We explain that this is because the amount of snow fall is greater than 6 inches/hour.

```
=> (> (send snow-text :snow-fall) 6)
```

Then, the system asks another question:

This case matches the operation:

Migration graphics

```
(send snow-picture :set-translucency 30).
```

However you demonstrated a different example

How do you distinguish

```
(send snow-picture :set-translucency 60) from
```

```
(send snow-picture :set-translucency 30)?
```

We explain that this is because the snow cloud is overlapping with the mule-deer object:

```
=> (send snow-picture :overlapping-to mule-deer).
```

The design procedure, "display-snow", has been programmed. Appendix C shows the result of applying all three design procedures concurrently.

CHAPTER 5

Conclusion

In this thesis, the Example-Based Graphical Programming System has been implemented. The object oriented design editor is built as an interface where the graphic designer shows examples. The system records the demonstration of design examples as a process of graphical editing. The generalizer then generalizes those examples into a computer program.

We have shown that the Example-Based Graphical Programming System is fairly easy to use for the designer, who is also a novice programmer, with minimal practice.

The system has been tested for the graphic design of migration graphics. The behaviors of graphical representation for dynamic information have been programmed by demonstrating design examples. This study showed that the user can quickly create a computer program using the system, as opposed to using more conventional programming languages.

5.1 Future work

In the current system, a lisp-based object oriented language is used for some of the designer-machine communication. Even though the system is visually oriented and easy to use, the designer has to learn simple statements to communicate with the system. A simple extension would be to use an English-like language. However, when using such a language, the designer always has to remember its syntax. Several different graphical languages have also been investigated. Graphical languages are known as intuitive interaction tools, but when we have to communicate a complex concept, such as conditionals or procedures, the language becomes unwieldy. This area needs to be explored.

The system also does not provide a static representation of the program. Therefore, it is hard to understand the program since it is automatically generated. Even though the system does not require the user to read and understand the programming language, if the language is easy to understand, such as Cusp* or HyperTalk, it will be useful for many applications. If we provide a static representation of the program, the designer may be able to learn the programming language from examples. The use of easy textual programming languages may also extend the

* Customer programming. A special English-like language provided by SmallStar [Halbert 81].

Conclusion

limitation of direct manipulation. Again, the designer does not need to learn programming to use the system, this is only a useful optional feature.

The designer rarely selects only good examples without making mistakes. The system currently does not support editing and debugging of the program. Editing capability is crucial in a graphic design program as well as in any other domains. In general, designing is a process of trial and evaluation. Adding an editing module to the system would allow the graphic designer to change his/her mind while showing examples. The most interesting area of research is to investigate editing by example.

The Example-Based Graphical Programming System can be implemented for various design applications, such as screen layout or corporate identity as well as the display of dynamic information. It is interesting to test the idea for other design problems.

The system has been tested only by a few graphic designers. Testing the system with more designers is also important to evaluate the potential of this technique as well as its limitation.

BIBLIOGRAPHY

- [Amari 87] Thomas R. Amari. "Automating the Design of Packaging Families Using PackIT, The Packager's Inferencing Tool," Masters Thesis, MIT, 1987.
- [Badshah 87] Alka G. Badshah. "GRID - Graphic Intelligence in Design: An Expert Layout System," Masters Thesis, MIT, 1987.
- [Charniak 84] Eugene Charniak, Drew McDermott. Introduction to Artificial Intelligence. Addison Wesley, 1984.
- [Dietterich 81] Thomas G. Dietterich, Ryszard S. Michalski. "Inductive Learning of Structural Descriptions: Evaluation Criteria and Comparative Review of Selected Methods," Artificial Intelligence 16, 1981.
- [Greenlee 88] Russell L. Greenlee. "From Sketch to Layout: Using Abstract Descriptions and Visual Properties to Generate Page Layout," Masters Thesis, MIT, 1988.
- [Halbert 81] Daniel C. Halbert. "An Example of Programming By Example," Masters Thesis, University of California, Berkeley and Berkeley and Xerox corporation Office Products Division, Palo Alto, CA., 1981.
- [Hayes-Roth 77] Frederich Hayes-Roth, John McDermott. "Knowledge Acquisition from Structural Descriptions," Proceedings of the 5th International Joint Conference on Artificial Intelligence, 1977. pp.356-362.

- [Lieberman 84] Henry Lieberman. "Video games by example," SigGraph Video Review (videotape) 12(1).
- [Lieberman 86] Henry Lieberman. "An Example Based Environment for Beginning Programmers," *Instructional Science* 14, Amsterdam, 1986. pp.277-292.
- [Lieberman 88] Henry Lieberman. "Design by Example," unpublished paper, 1988.
- [Myers 88] Brad A. Myers. *Creating User Interfaces by Demonstration*. Academic press, INC., San Diego, 1988.
- [Myers 86] Brad A. Myers. "Visual Programming, Programming by Example, and Program Visualization: A Taxonomy," *CHI'86 Proceedings*, 1986. pp.59-66.
- [Mitchell 81] Tom M. Mitchell. "Generalization as Search," *Reading in Artificial Intelligence*, Morgan Kaufmann Publishers Inc., 1981. pp517-546.
- [Reiss 86] Steven P. Reiss. "Displaying Program and Data Structures," *Technical Report No. CS-86-19*, Brown University, 1986.
- [Shu 88] Nan C. Shu. *Visual Programming*. Van Nostrand Reinhold, New York, 1988.
- [Stillings 87] Neil A. Stillings. "Artificial Intelligence: Search, Control, and learning," *Cognitive Science: An Introduction*, MIT Press, Cambridge. 1987. pp.171-213.

[Böcker 86] Heinz-Dieter Böcker, Gerhard Fischer. "The Enhancement of Understanding through Visual Representation," CHI'86 Proceedings, 1986. pp.44-50.

APPENDIX A

| DESIGN | PROCEDURE | PROJEC |
|-------------|----------------|--------|
| NEW ELEMENT | NEW | |
| REMOVE | LOAD | |
| REMOVE ALL | CLOSE | |
| ATTRIBUTES | SAVE | |
| INFORMATION | PRINT | |
| MOVE | LIST ALL | |
| SCALE | ADD ELEMENT | |
| PALETTE | REMOVE ELEMENT | |
| BACKGROUND | SHOW EXAMPLE | |
| | CANCEL | |
| | DONE | |
| | DEFINE | |
| | APPLY | |

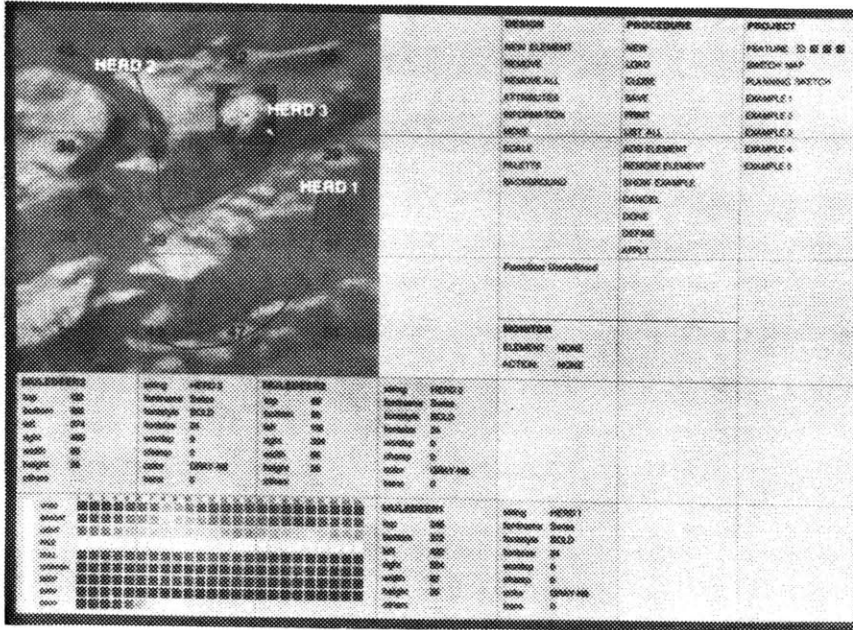
Menu

| TEMPERATURE16 | | | |
|---------------|-----|-----------|---------|
| top | 47 | string | 45 |
| bottom | 73 | fontname | Swiss |
| left | 62 | fontstyle | REGULAR |
| right | 90 | fontsize | 24 |
| width | 28 | wordsp | 0 |
| height | 26 | charsp | 0 |
| others | ... | color | GRAY-N0 |
| | | trans | 30 |

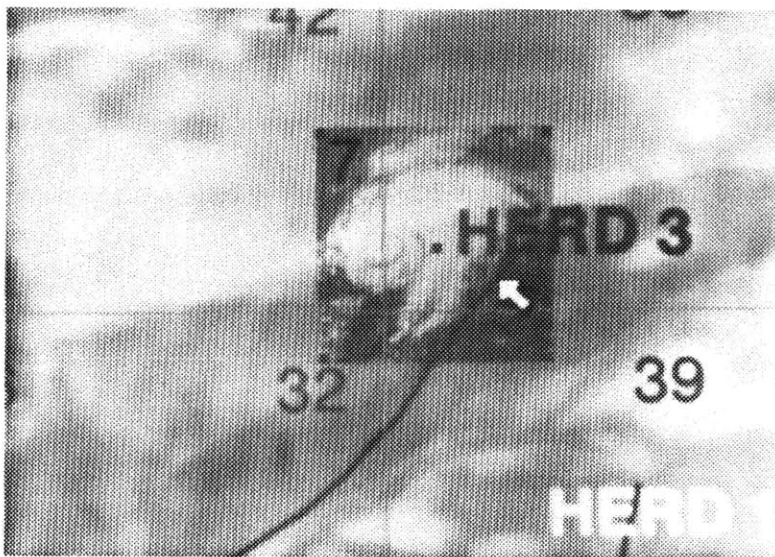
Attribute menu

APPENDIX B

The interface of the Example-Based Programming System for the dynamic display of mule deer migration.



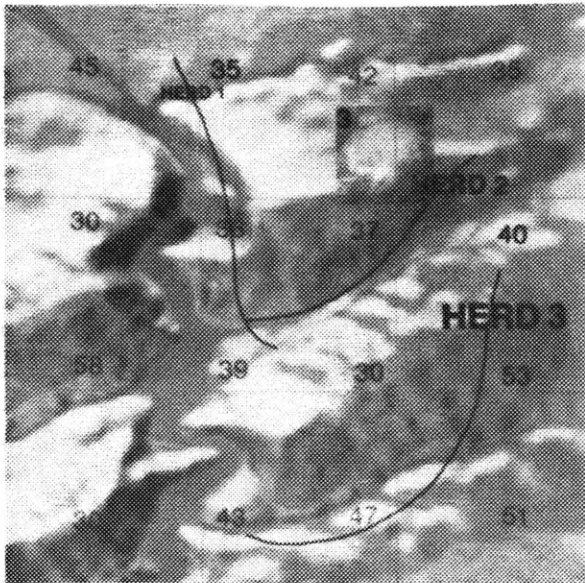
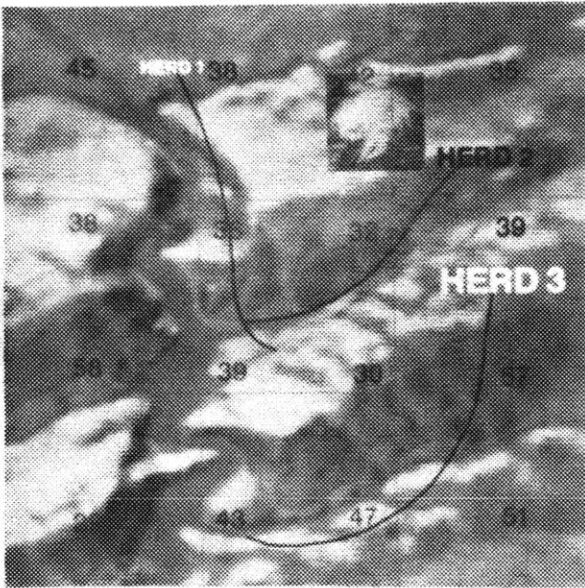
Display of the interface

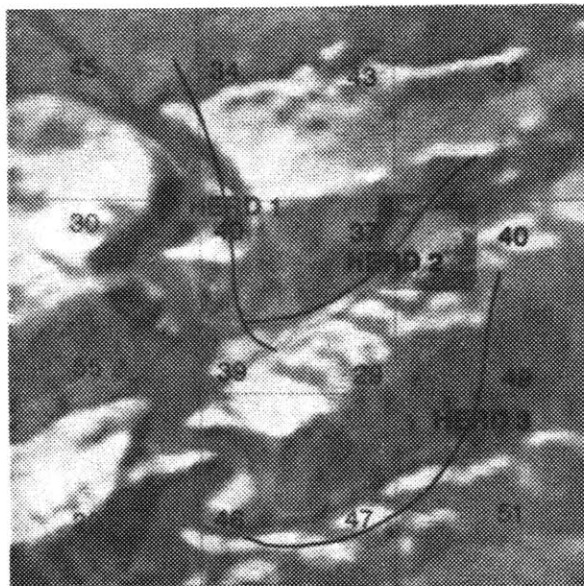
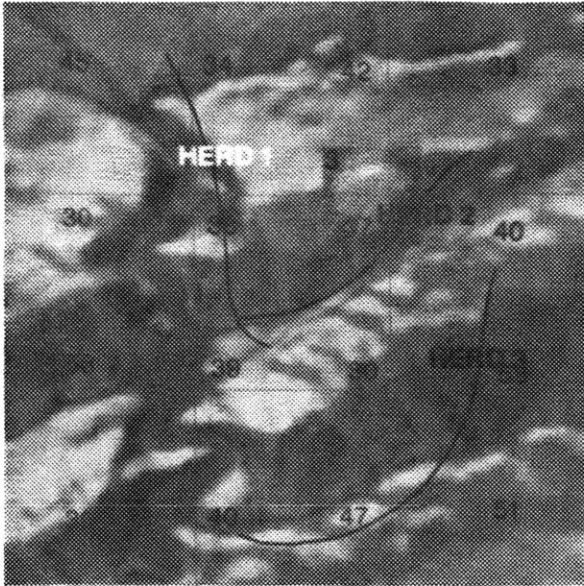


Graphical Editing

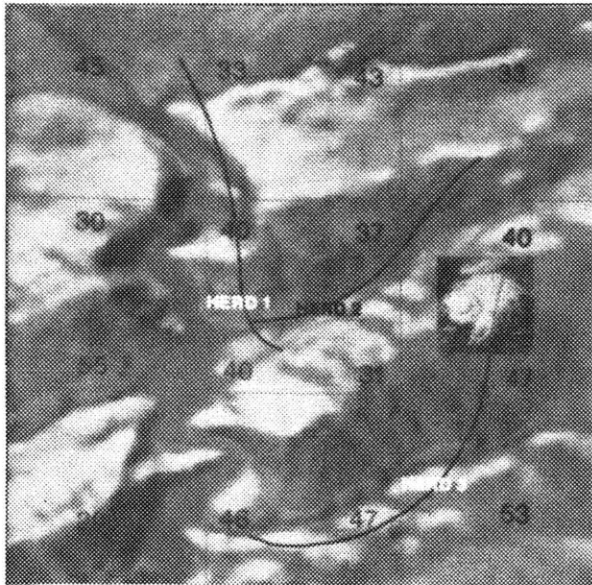
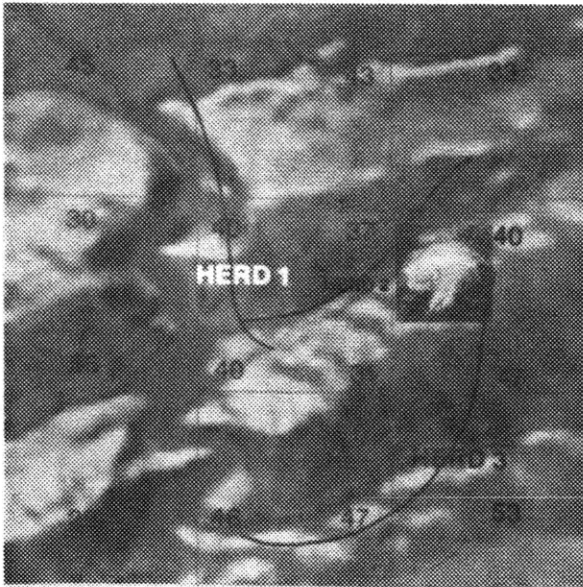
APPENDIX C

Design procedures programmed in chapter 4 are applied for the representation of simulated real-time information.





APPENDIX C



APPENDIX C

