Experiments

in

Interactive Map Retrieval

A thesis
submitted in partial fulfilment
of the requirements for the Degree of
Doctor of Philosophy in Computer Science
at the
University of Canterbury
by
Gregory C. Ewing

University of Canterbury
1992
Contents

Abstract ................................................................................................................................. 1

1 Introduction ....................................................................................................................... 3

2 A Map Browsing System .................................................................................................. 9
  2.1 User's View .................................................................................................................. 9
    2.1.1 Database ........................................................................................................... 9
    2.1.2 Browsing Functions .................................................................................... 11
  2.2 System Components .................................................................................................... 21
    2.2.1 Display Server .............................................................................................. 21
    2.2.2 User Interface ............................................................................................... 21
    2.2.3 Retrieval Engine .......................................................................................... 22
    2.2.4 Distribution of components across a network ............................................ 22
  2.3 Summary .................................................................................................................... 23

3 Graphical User Interfaces ................................................................................................. 25
  3.1 User interface metaphors .......................................................................................... 25
  3.2 Tools for building graphical user interfaces .......................................................... 26
    3.2.1 The separate development approach ......................................................... 26
    3.2.2 Tools for separate user interface development ........................................ 27
    3.2.3 The framework approach ........................................................................... 31
    Smalltalk .................................................................................................................. 32
    SunView .................................................................................................................. 32
    MacApp .................................................................................................................... 34
    InterViews ............................................................................................................... 36
  3.3 Display server systems .............................................................................................. 38
    3.3.1 X Windows ....................................................................................................... 39
    3.3.2 NeWS .............................................................................................................. 40
  3.4 Summary .................................................................................................................... 44

4 P: An Alternative to Handwritten PostScript .................................................................. 47
  4.1 Problems .................................................................................................................... 48
Contents

6.4.2 Output ............................................................................................................. 88
6.4.3 Scrolling and multiple views ............................................................................ 88
6.4.4 Summary ......................................................................................................... 91
6.5 Existing user interface customisation methods.................................................. 92
  6.5.1 SunView defaults file ...................................................................................... 92
  6.5.2 Macintosh resource files ............................................................................. 92
  6.5.3 X resources .................................................................................................. 93
  6.5.4 NeWS .......................................................................................................... 94
6.6 An architecture for a style-independent user interface system ...................... 95
6.7 Further benefits of a style-independent architecture .................................... 96
  6.7.1 Programming convenience ......................................................................... 97
  6.7.2 Application portability ................................................................................ 97
  6.7.3 User interface portability ............................................................................ 98
6.8 Designing a style-independent SAI ................................................................. 98
  6.8.1 Basic abstractions ....................................................................................... 99
  6.8.2 Specifying VObjects ................................................................................ 100
  6.8.3 Standard VObjects ................................................................................... 101
  6.8.4 Standard commands .................................................................................. 102
  6.8.5 Application-defined commands ................................................................ 103
  6.8.6 Automatic interface generation .................................................................. 104
6.9 Summary ............................................................................................................ 104

7 Database Design for Interactive Browsing ......................................................... 107
  7.1 Geographic data models .................................................................................. 107
    7.1.1 Spatial and non-spatial data ..................................................................... 107
    7.1.2 Raster vs. Vector representations .......................................................... 109
    7.1.3 Vector data models ................................................................................. 109
    7.1.4 Browsing-specific issues ......................................................................... 110
    7.1.5 Layers ...................................................................................................... 111
    7.1.6 Generalisation ......................................................................................... 111
    7.1.7 Multiple representations ......................................................................... 112
    7.1.8 Variable-resolution data structures .......................................................... 112
  7.2 A browsable geographic data model ............................................................... 114
    7.2.1 Terminology ............................................................................................ 114
    7.2.2 Entities and their representations ............................................................ 115
    7.2.3 Spatial objects .......................................................................................... 115
    7.2.4 Object sharing .......................................................................................... 117
10.1.1 The Point Quadtree ......................................................... 160
10.1.2 The PR Quadtree ............................................................... 160
10.1.3 Representing lines and areas using quadtrees ...................... 161
10.1.4 Quadtree-based spatial indexes ........................................... 162
10.1.5 Other methods related to the quadtree ............................... 163
10.2 Locational Keys ................................................................. 163
10.3 An experiment using locational keys .................................... 166
  10.3.1 Implementation .............................................................. 166
  10.3.2 Single key per object ...................................................... 167
  10.3.3 Multiple keys per object ................................................. 169
10.4 Conclusions .............................................................. 171

11 K-D Trees .............................................................................. 173
  11.1 The k-d tree ........................................................................ 173
  11.2 The experiment ................................................................. 174
    11.2.1 Pruning the search ......................................................... 176
    11.2.2 Tuning the algorithm ..................................................... 176
    11.2.3 Experimental results ..................................................... 176
  11.3 Conclusions .............................................................. 178

12 Paged Binary Trees ................................................................. 179
  12.1 Tree structures for secondary storage .................................. 179
    12.1.1 B-tree related methods ................................................. 179
  12.2 Paged binary trees .............................................................. 180
  12.3 Paged k-d tree ................................................................. 181
    12.3.1 Data structure ............................................................... 181
    12.3.2 Retrieval algorithm ...................................................... 182
    12.3.3 Choice of page size ...................................................... 182
    12.3.4 Experimental results ..................................................... 186
  12.4 Conclusions .............................................................. 188

13 The SKD Tree .............................................................. 189
  13.1 Matsuyama’s k-d tree .......................................................... 189
  13.2 The SKD tree ................................................................. 190
  13.3 An experiment using the skd tree ...................................... 191
  13.4 Conclusions .............................................................. 194

14 Using Custom Storage Structures ........................................ 195
14.1 Introduction ................................................................. 195
14.2 The custom data structure ........................................ 196
14.3 Experimental results .................................................. 196
14.4 Conclusions ............................................................... 197

15 Exploiting Locality of Reference ........................................ 199
15.1 Locality of reference in general .................................... 199
15.2 Locality of reference in map browsing ....................... 200
15.3 Cacheing and Prefetching .............................................. 200
  15.3.1 Cache architecture ................................................ 200
  15.3.2 Experimental results ............................................. 201
  15.3.3 Other reference patterns ....................................... 201
  15.3.4 Prefetching .......................................................... 203
  15.3.5 Replacement strategy ........................................... 204
15.4 Layer grouping ............................................................ 204
  15.4.1 Analysis ............................................................... 204
  15.4.2 Experimental results ............................................. 206
15.5 Conclusions ............................................................... 210

16 Conclusions ................................................................. 211
16.1 User interface ............................................................ 211
16.2 Database design ........................................................ 214

Appendix A: P Syntax and Semantics .................................... 217
  A.1 Syntax ................................................................. 217
  A.2 Semantics ............................................................. 218

Acknowledgements .......................................................... 221

References ................................................................. 223
Abstract

The thesis deals with the problem of constructing an interactive, visual browser for a large database of geographic data. Results are presented from work in two areas: one area is that of graphical user interfaces, the other is the structuring of a geographic database for rapid retrieval of information for interactive display.

The design is discussed of a user interface for a map browsing system. Proposals are then made about how a map database should appear to the user, and the functions that a map browsing system should provide. Observations are made about user interface systems in general. Particular reference is made to the difficulty, with present user interface systems, of ensuring consistency of the style, or "look and feel", of the user interface between applications, and of allowing the style to be customised to suit user preferences. An architecture for user interface systems is proposed in which applications are independent of the style of the user interface, so that the user may change the style at will. Suggestions are made as to how such a system might be realised.

The NeWS extensible window server, based on the PostScript graphics programming language, is examined. Some characteristics of PostScript are identified which make this language attractive for the task of displaying maps. Possibilities offered by NeWS for customisable user interfaces are explored.

To overcome the difficulties in writing large PostScript programs by hand, the author devised a new language P. A user interface toolkit for NeWS, written by the author in P, is described. Use of P made the task of writing the toolkit much easier. General observations are made about the design of such toolkits in the NeWS environment. Suggestions are made for improving the NeWS server-client communication model and support tools.

The second part of the thesis discusses the design of geographic databases. Some extensions are suggested to general geographic data models to support the specific requirements of interactive browsing. Results are presented from a set of experiments with different techniques of implementing these extensions.

Implementations are compared of several spatial indexing methods in a relational database environment. Locality of reference in map browsing is discussed. Experimental assessments are made of the following methods for exploiting this
locality: clustering, caching, and pre-fetching.

From the results of these experiments, conclusions are drawn about how to efficiently organise and index spatial data for interactive browsing in a conventional relational database system. Many of these conclusions are applicable in general to other forms of database management system.
Chapter 1

Introduction

This thesis is concerned with the design and implementation of interactive, graphical browsing systems for large databases of geographic information. These systems are becoming of interest due to a combination of recent trends and technological developments.

Geographic data is increasingly becoming available in machine-readable form. For example, in New Zealand, basic survey data is being stored in this form, and raster image data from remote sensing is being collected (LINZ 1986, Penny, Ewing, Pascoe and Forer 1987). This trend, together with the rapidly decreasing cost of hardware such as powerful graphics workstations and large-capacity disk storage, is directing more and more effort into the development of large geographic databases. In particular, computer cartography is becoming an increasingly attractive way of creating, and providing access to, map images.

Collecting geographic information is time-consuming and expensive. Consequently, a geographic database is a very valuable resource, and there is a strong incentive to share the database as much as possible. In New Zealand, for example, there is a project underway to assemble a single pool of data used by all Government departments which have need of geographic information.

Technology exists for such a pool of data to be made available to users throughout the country, each with a personal graphics workstation connected to a high-speed data network. As an example of the possibilities offered, in New Zealand the Department of Survey and Land Information "intends that the legal boundaries of any parcel of land in New Zealand will ultimately be able to be captured and reproduced at the touch of a button anywhere in the country, and will be up to the minute in the detail it shows" (Penny et al 1987).

In parallel with the appearance, at reasonable cost, of powerful, personal graphics workstations has been the increasing popularity of interactive, graphical user interfaces. These interfaces incorporate concepts such as multiple windows and
direct-manipulation metaphors, that make the capabilities of computer systems easy
to use and readily available to non-technical users. Applying these concepts to the
area of geographic information systems leads naturally to the development of powerful
and flexible interactive map browsers.

Successful construction of an interactive map browser requires solving problems in
two main areas: design and implementation of the user interface, and organisation of
the database to support interactive browsing.

In this thesis, Chapter 2 discusses the design of a user interface for interactive
map browsing. The way the database should appear to the user is described. It is
proposed that the database should appear as a large, seamless map made up of a
number of layers, and should be able to be viewed at any of a range of scales, or
viewing levels, with appropriate detail visible at each level. A set of basic functions
that a map browser should provide is defined. Additional functions that such a system
could also provide are suggested. A division of the browsing system into software
components, that can be distributed across a network in various ways, is proposed.

Chapter 3 reviews techniques that have been developed to ease the difficult task
of constructing applications that are to have highly interactive, graphical user
interfaces. Two main classes of tools are described, corresponding to two main user
interface metaphors, the conversational world and the model world. The concept of a
window server for use in a network environment is discussed, and the main features of
two window server systems are described: the widely-used X Windows system; and
Sun Microsystems’ NeWS extensible window server, based on Adobe’s PostScript
graphics programming language.

Chapters 4 and 5 present work that the author has done with the NeWS window
system. The author chose NeWS as a basis for the user interface of an experimental
map browsing system, because of the advantages of PostScript’s rich graphics model
for the application of map display. To provide facilities missing from the version of
NeWS then available, the author wrote a graphical user interface library as an
extension to NeWS.

While writing this library, the author found that making substantial extensions to
NeWS by means of handwritten PostScript code was a tedious and error-prone task.
Chapter 4 describes a new language, P, which the author designed to make the
facilities of PostScript available to the programmer in a much more convenient form.

Chapter 5 describes the main characteristics of the author’s user interface library,
and presents conclusions drawn from the experience of implementing it. Observations are made about the general design of user interface libraries for NeWS and the most appropriate form of communication between such a library and the client programs which use it. Shortcomings in the overall model of communication between the NeWS server and its clients are pointed out, and suggestions are made for the improvement of this model. Proposals are made for enhancements to the tools that NeWS provides for implementing server-client protocols, to support the improved model.

Chapter 6 discusses general issues of consistency of style, or "look and feel", between the user interfaces of different applications. It is shown that the design of current user interface toolkits results in the style of an application's user interface being largely fixed when the application is written. It is further shown that this style dependency comes about because of the nature of the application program interface (API) — that is, the boundary between the user interface toolkit and the applications which use it.

An alternative architecture for user interface systems is proposed, founded on the principle that the user should be in complete control of the user interface style of all the applications that he or she uses. This architecture is based on a standardised, style-independent API to be used by all applications. The beginnings of a design for such a style-independent API are presented, some problems that will be encountered in its further development are anticipated, and possible ways of overcoming these problems are outlined.

Interactive map browsing makes unique demands on the geographic database that are not found in other applications of GIS. To maintain acceptable interactive responsiveness, very fast retrievals of spatial data for a chosen geographic region are required. In addition, it must be possible to quickly retrieve data suitable for display at a wide range of viewing levels.

Chapter 7 discusses the design of geographic databases for interactive browsing, and presents in general terms a data model for a browsable geographic database. The model presented incorporates a number of extensions to general-purpose spatial data models, to support the unique requirements of interactive browsing.

To investigate techniques for implementing and using the data model described in Chapter 7, the author conducted a series of experiments. Chapter 8 sets out the overall objectives of these experiments, and describes the environment in which they were performed and the methodology employed. The main objectives were twofold: (1) to gain general insight into the kinds of data structures and algorithms required to
support interactive map browsing, and (2) to investigate whether spatial data held by a conventional relational database management system can be retrieved quickly enough for interactive display.

The experiments were conducted using a test database containing cadastral data supplied by the Christchurch Drainage Board and stored using the Ingres relational database system. Chapter 9 describes a preliminary series of experiments to find an optimum schema and set of database parameters for storing the spatial data, and to obtain benchmark timings for use as a control case to compare with further results. It is found that, once the required set of spatial objects is known, they can easily be retrieved from the relational database with sufficient speed for interactive use.

The flexibility and power of relational database systems makes them very attractive for storage of data generally, and geographic data is no exception. However, the one-dimensional indexing structures found in most relational database systems are not directly suitable for indexing spatial data, which is inherently multi-dimensional. One solution to this problem is to store spatial indexes, and perhaps the spatial data itself, outside the relational database. Another solution is to build extra software layers upon the relational database system to support spatial indexing.

Chapters 10 and 11 describe experiments with techniques which attempt to make as much use as possible of the relational database system’s query processing capabilities to implement spatial indexing. Chapter 10 investigates a technique due to Abel and Smith (1983, 1984, 1986) involving quadtree-based locational keys stored in a conventional one-dimensional index structure. Chapter 11 investigates the use of a k-d tree (Bentley 1975) stored in the relational database and searched breadth-first using database query statements.

Chapters 12 and 13 describe experiments with techniques aimed at avoiding the overhead associated with processing complex database queries, by performing more processing in software layers above the relational database system. Chapter 12 investigates the use of paged binary tree techniques (Knuth 1973) applied to the k-d tree. Chapter 13 investigates a variant of the k-d tree described by Ooi (1990) called the skd tree.

Chapter 14 describes an experiment to assess how much efficiency can be gained by sacrificing some of the flexibility of the relational database system and storing some or all of the spatial data and indexes outside the relational database. It is found that a substantial gain in retrieval speed can be made by storing the spatial index outside the relational database, and a smaller gain by doing the same with the spatial
data itself. Consequently, a good compromise would be to keep the spatial data in the relational database, retaining much of the flexibility, while storing the spatial indexes outside to improve retrieval speed.

Chapter 15 discusses locality of reference in map browsing. Patterns of behaviour that commonly occur during an interactive browsing session, such as zooming and scrolling, exhibit considerable locality of reference. Techniques for exploiting this locality are discussed, including caching and prefetching, and organising the database to take advantage of thematic relationships between layers. The results are presented of an experiment to assess the effectiveness of these techniques. It is found that the use of these techniques can substantially improve the interactive responsiveness of a map browser.

Chapter 16 summarises the results and conclusions presented in the preceding chapters. The main conclusions from the experiments conducted are, briefly:

(1) When using a conventional relational database system, out of the methods investigated the fastest retrieval by geographic area is provided by techniques such as those described in Chapters 12 and 13. These methods are based on standard spatial indexing structures and paged tree techniques. The relational database is used simply as a repository of data, and most of the processing is performed in software layers above the relational database system.

(2) If some of the flexibility of the relational database can be spared, considerable efficiency of retrieval can be gained by moving spatial indexes outside the relational database. By keeping the spatial data itself in the relational database, much of the flexibility can be retained with only a moderate effect on retrieval speed.

(3) Techniques for exploiting locality of reference can make considerable improvements to the interactive performance of a map browser. These techniques can be applied not only to relational databases but to geographic database systems in general.
Chapter 2

A Map Browsing System

We assume that an interactive map browsing system will be based on powerful, personal graphics workstations giving access to a typically large database of geographic data. The system should be able to quickly retrieve data representing entities of chosen types from a chosen geographic domain, and display it as a map at a chosen scale.

In this chapter, Section 2.1 discusses a map browsing system of this kind from the user's viewpoint: the structure of the database as seen by the user, and the functions provided to the user by the map browsing system. Section 2.2 discusses how such a system might be divided into software components and the way these components are interrelated.

2.1 User's View

The user's view of the system encompasses two main aspects: the logical structure of the database from the user's viewpoint; and the user interface of the browsing system in terms of the functions provided and the way that the user invokes these functions.

2.1.1 Database

To the user, the database consists of a single seamless domain, consisting of multiple layers of data, that may be viewed at different scales or viewing levels.

Seamless domain

Goodchild (1989) discusses the management of very large collections of geographic data, where "'Large' can be usefully defined as exceeding our current capability to deliver." He observes that, while it will probably be necessary to physically partition data by theme and geographic region, for administrative purposes and to speed retrieval, "...partitioning may be at least partially hidden from the user, who may value the ability to browse the database in an apparently seamless fashion."
Following this suggestion, the browsing system should make the database appear as a single, seamless map covering the entire geographic domain of interest, that the user may browse over without being hampered by artificial boundaries.

**Layers**

The database will contain objects representing geographic entities of many different types. At any given time, the user will most likely want to view only a small subset of the available objects. In fact, with a sufficiently rich database, showing all objects in a given region would result in a hopelessly cluttered display, so that a means of restricting the types of objects shown will be a necessity.

As discussed by Goodchild, it is common to divide objects into groups called *layers* based on their type. The term "layer" derives from the practice of drawing maps on transparent sheets which can be overlaid to create a composite map. Here, it simply means a set of geographic features on a particular theme.

The map browser should provide a means for the user to find out the available layers and select which layers are to be displayed. The database should be structured in a way that facilitates retrieval of objects from specific layers.

**Viewing levels**

As discussed by van Oosterom and van den Bos (1989), interactive browsing of a GIS requires that the data be displayable at several levels of detail. At the beginning of a browsing session, the user may be shown a small-scale, simplified map of the whole database. As the user zooms in on a selected portion of this map, more detail should progressively appear, until at some scale all the detail present in the database becomes visible.

The terms "spatial domain" and "layer" are well established in traditional cartography. For interactive mapping, we introduce a new term, *viewing level*, meaning a parameter that determines the amount of detail required in a displayed map.

Table 1 shows a hierarchy of viewing levels based on those suggested by Penny et al (Penny, Ewing and Pascoe 1989) for the New Zealand Digital Cadastral Database, showing examples of the kinds of features that would be revealed at each level.

In that paper, viewing levels were integers labelling a set of arbitrarily-chosen scale steps at which more detail would be displayed. It was observed, however, that
the scale steps chosen differed from one another by approximately a power of 10, suggesting the following definition of the viewing level parameter:

\[
\text{viewing \_ level} = - \log_{10} \text{scale}
\]  
(2.1)

By regarding the viewing level as a continuous parameter, the user is not restricted to a discrete set of display scales, nor are the steps in detail revelation restricted to a set of discrete levels.

There are other ways in which the viewing level parameter could be defined; for instance, the resolution of the display device might be taken into account, since this will influence the amount of detail that can be displayed. For the present purposes, however, we will use the definition of Eq. (2.1).

2.1.2 Browsing Functions

There are many functions that could be provided by a map browser, and many ways that these functions could be embodied in a user interface. This section lists the most basic functions that should be provided by any map browser, and an example of a user interface embodying these functions. Some suggestions are also given for further functions that could be incorporated into the map browsing framework.

**Basic functions**

The essential function provided by the map browser is that of selecting a geographic area of interest, retrieving spatial information about it, and displaying the information in the form of a map.

<table>
<thead>
<tr>
<th>Viewing level</th>
<th>Name</th>
<th>Display scale</th>
<th>Features displayed</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>National</td>
<td>1:10,000,000</td>
<td>Coastline</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 main cities</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>14 towns with title offices</td>
</tr>
<tr>
<td>6</td>
<td>Provincial</td>
<td>1:1,000,000</td>
<td>All major towns</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Major rivers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Railways</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Main highways</td>
</tr>
<tr>
<td>5</td>
<td>City or District</td>
<td>1:100,000</td>
<td>Main roads</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Some parks</td>
</tr>
<tr>
<td>4</td>
<td>Suburb or Town</td>
<td>1:10,000</td>
<td>All streets</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Names on main roads</td>
</tr>
<tr>
<td>3</td>
<td>Street or Farm</td>
<td>1:1000</td>
<td>Names on all streets</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Plan numbers</td>
</tr>
<tr>
<td>2</td>
<td>House lot</td>
<td>1:100</td>
<td>All details</td>
</tr>
</tbody>
</table>
From a simplified map of the whole domain covered by the database, the user should be able to select a region with the pointing device and obtain a more detailed view of it at a larger scale. The new view could be displayed in the same window, replacing the previous view, or a new window created.

Other basic functions that should be provided are:

* Zooming

Changing the scale of a map shown in a window.

* Panning or Scrolling

Moving the viewed region over the underlying map.

* Layer selection

Selecting which layers are to be shown in a given view, and the order in which they are to be overlaid.

* Window management

It is assumed that the map browser will be supported by a graphical user interface system that provides all of usual facilities for managing multiple windows, such as allowing the user to move and resize windows and change their overlapping order.

A user interface

Figure 2.1 shows a sequence of snapshots of the author’s map browsing system, CantaBrowse, developed to illustrate a possible way of making some of these functions available to the user.

The database shown was obtained from the New Zealand Department of Survey and Land Information (DoSLI), and contains cadastral data for a domain encompassing Lyttelton Harbour and a south-eastern portion of Christchurch. The data is divided into several layers, including “Roads”, “Coastline” and “Parcel Boundaries”.

Figure 2.1(a) shows the initial reference map that the system presents to the user. At this viewing level, only features from the “Roads” and “Coastlines” layers
are visible, and roads are displayed as single lines. A rectangular area has been chosen from it using the mouse.

In (b), the user has opened up the selected rectangle by double-clicking in it, and the system has created a new window showing the selected region at a larger scale. More detail is visible in this view; streets which were shown as single lines in the reference map are now shown as double lines.

In (c), the user has scrolled the view to the right a little by clicking on one of the scrolling arrows. The rectangle in the reference window has moved accordingly to reflect the new area being viewed by the window.

In (d), part of the second window has been selected and opened up to create a third window.

In (e), the user has initiated a dialogue for selecting layers to be viewed, and enabled the “Parcel boundaries” layer. After confirming the selection, (f) shows the window with parcel boundaries now displayed.

Other functions

Some other functions the map browser could provide include

- **Projections**

  The user could be given a choice of popular map projections, such as Mercator or polar, for displaying the retrieved data.

- **Navigation from spatial to non-spatial data**

  The user could select a feature with the pointing device and obtain non-spatial attribute data about it. For example, in the DCDB the user might point to a land parcel and find out its legal description, owner, area, etc.

- **Navigation from non-spatial to spatial data**

  The user could select an area to view by some non-spatial criterion. For instance the user might enter a street address and have the corresponding land parcel displayed.
Figure 2.1a
Reference map
Figure 2.1b
More detailed view of a selected region
Figure 2.1c
View after scrolling
Figure 2.1d
View after zooming a second time
Figure 2.1e
Dialogue box for selecting layers to view
Figure 2.11
View including another layer
• **Overlay**

A frequent operation on spatial data is that of overlaying one set of polygons on another to create a third set that consists of their intersection. A form of overlaying can be carried out using the facilities already described, by visually superimposing one map layer on another. For example, a layer representing areas of high rainfall might be displayed atop another representing catchment areas, and suitable sites for a new hydroelectric power station found by inspection.

For some applications, visual overlaying may be sufficient, while for others it may be necessary to calculate the intersection. For instance, to find the amount of water collected annually by the regions found in the previous example, it would be necessary to find the intersections of the two sets of polygons and calculate their area.

These and other manipulations of geographic data are outside the scope of the browsing system itself. However, interfaced to other modules of the GIS, the browser would provide an excellent tool for selecting the data to be operated upon, specifying the operation to be performed, and displaying the results.

• **Printing**

To create a permanent record of a displayed map, the contents of the window would be sent to a printer, or saved in a file for use by another application such as a desktop publishing system.

• **Annotation**

The user may wish to manually add symbols or labels and otherwise enhance the visual appearance of a map before saving or printing it. One way of providing these capabilities would be to make the output format compatible with that of some existing drawing program.

Alternatively, a set of drawing tools could be provided by the map browser to allow editing to be done without leaving the browsing environment. This would also allow provision of specialised editing functions relating to cartography that would not be found in a general-purpose drawing application.
2.2 System Components

The map browsing system can be divided into several components. In Figure 2.2, the system is depicted as consisting of two major components, the map browser itself, and the database. It is assumed that there will be a separate instance of the map browser running for each user, whereas the database and DBMS will be shared among users.

The map browser can be further subdivided into a display server, a user interface and a retrieval engine.

2.2.1 Display Server

In a network environment, it is common for access to graphical input and display facilities to be mediated by a display server running on the user’s workstation. The display server provides high-level, device-independent graphics functions to client applications running either on the workstation or another host on the network, and arbitrates access to the display by multiple clients.

Examples of display servers are the X Windows server (Schleifer and Gettys 1986) and Sun Microsystems’ NeWS (Gosling 1986; Sun 1987, 1988a). These display servers will be discussed in more detail in Chapter 3.

2.2.2 User Interface

The user interface is responsible for all the things the user sees: windows, menus, scroll bars, dialogue boxes and so forth. It translates user actions, such as mouse clicks in a window, into requests for the retrieval of data from particular layers and for a particular geographic region. It also takes the retrieved data and presents it to the user in the form of a map, carrying out any necessary coordinate transformations and selecting appropriate symbology.
2.2.3 Retrieval Engine

The retrieval engine is responsible for retrieving requested data from the database as efficiently as possible. The general form of request handled by the retrieval engine can be stated as

\[
\text{retrieve}(\text{level}, \text{layers}, \text{domain})
\]

where

- \textit{level} is the viewing level at which the data will be displayed,
- \textit{layers} is a set of layers, and
- \textit{domain} is a geographic region (assumed to be rectangular).

The contents of a window at any given time is defined by these three parameters.

Depending on the implementation of the database, the retrieval engine may carry out this request by simply issuing a few query statements to the DBMS, or it may carry out more extensive processing, such as looking up a spatial index.

2.2.4 Distribution of components across a network

It is assumed that the display server will run on the user's workstation, and that the database will reside on disks attached to one or more file servers on the network. The remaining components of the system can be distributed around the network in a variety of ways, some of which are depicted in Figure 2.3.

Figure 2.3a shows a configuration in which the user interface and retrieval engine of the map browser both reside on the user's workstation and communicate with the DBMS over the network.

If there is much communication between the retrieval engine and the DBMS, the retrieval engine could be moved closer to the DBMS as shown in Figure 2.3b. Using this configuration, only the data requested by the user need be transmitted over the network, any other traffic between the retrieval engine and DBMS being local to the file server.

Sometimes the user's "workstation" will be a dedicated machine, such as an X terminal, which runs a display server only, with application programs being run on a
Chapter 2 – A Map Browsing System

2.3 Summary

In this chapter, the broad characteristics of an interactive browsing system for geographic data has been outlined. A set of basic functions to be provided by such a system has been defined, and an example of a user interface embodying these

Figure 2.3
Ways of distributing components of the map browsing system across a network

separate “computing server” shared by many users. In these cases a configuration such as Figure 2.3c or d could be used.
functions presented. Further functions that could be provided by the system have also been suggested.

A division of the system into major software components has been proposed in such a way that the various components can be distributed over a network in a flexible manner to suit a variety of environmental conditions.
Chapter 3
Graphical User Interfaces

In recent years, the graphical user interface, or GUI, has become a very popular means for users to interact with a computer system. The GUI allows the user to interact with an application in a very direct way, and in terms relevant to the problem domain, leading to systems that are very easy to use.

Ease of use for the user, however, makes the programmer's task more difficult. Much effort is being put into finding ways of making it easier to construct applications with graphical user interfaces. This chapter is a review of the state of the art in this area.

Section 3.1 discusses two main metaphors for describing user interfaces, and Section 3.2 describes the main classes of tools that have been developed to aid the construction of user interfaces in each of these metaphors, and looks at the main features of some specific user interface toolkits. Section 3.3 examines the issue of graphical user interfaces in a network environment and looks at two network-oriented window systems, X Windows and NeWS.

3.1 User interface metaphors

Two main metaphors are useful for describing user interfaces, the conversational world and the model world (Hartson and Hix 1989).

In the conversational world, the user specifies what to do using a command language. This kind of interaction is known as a sequential dialogue, because it follows a prescribed sequence of actions and responses step by step. User actions in a sequential dialogue can include commands and data typed at a keyboard, clicking buttons with a mouse, and navigating through menus. Responses from the computer include display of prompts and data, and issuing of error messages.

In the model world, the user sees a collection of objects, graphically represented, which can be operated on directly by grabbing and moving them with a pointing device, or applying various tools to them. The term direct manipulation is applied to interfaces of this kind. In contrast to the single-threaded nature of a sequential dialogue, model
world dialogues tend to be *multithreaded* and *asynchronous*. There can be many interactions in progress at once, and the user may switch attention from one to another at any time.

The Xerox Star (Smith, Harslem, Irby and Kimball 1982), represents an early example of a model world interface. The Apple Lisa (Williams 1983) and Apple Macintosh (Williams 1984) were the first widely available systems based on the model world or "desktop" metaphor, and this kind of interface has become very widespread. Almost every personal computer or workstation now comes equipped with a graphical user interface based on a model world metaphor.

### 3.2 Tools for building graphical user interfaces

Two main approaches have been made to the problem of constructing applications with graphical user interfaces. These two approaches might be termed the *separate development* approach and the *framework* approach. Tools for building such applications can be divided into two classes according to the approach that they are based on.

#### 3.2.1 The separate development approach

The methodology of the separate development approach divides the application into two clearly delineated parts, the *dialogue* part and the *computational* part. Once the protocol between the two has been defined, development of each part can be carried out independently by different people using different tools and methods.

The concept of *dialogue independence* is central to this methodology. Hartson and Hix (1989, 1990) define dialogue independence as follows:

> "Dialogue independence is an approach in which design decisions affecting only the human-computer dialogue are isolated from those affecting only application system structure and computational software."

Hartson and Hix put forward many reasons for wanting dialogue independence, including

- Modularity - breaking a large system into relatively independent parts helps in dealing with complexity.
- Different methodologies and tools can be applied to the dialogue and computational parts. If the dialogue part can be specified and implemented without requiring programming, then non-programmers such as human
factors specialists and end users can more easily be involved in development of the user interface.

- Rapid prototyping - tools can be used to build mockups of proposed user interface designs and refine them iteratively.
- Multiple interfaces - several different user interfaces to an application could be provided to suit different hardware environments and user interface styles.

3.2.2 Tools for separate user interface development

Many tools have been developed to support the separate development methodology. All of these are based on the specification of the user interface in some formalism that is sufficiently precise to allow an implementation to be automatically generated from the specification.

Many of the formalisms used for specifying user interfaces are based on language theory. By analogy with the parsing of textual command languages, dialogue between the user and the application can be analysed into semantic, syntactic and lexical levels, and various schemes have been proposed for doing so (Foley and Wallace 1974; Foley 1980; Foley and van Dam 1982; Buxton 1983).

Jacob (1983) surveys work on two classes of formalism for specifying user interfaces: grammars expressed in BNF (Reisner 1981, Schneiderman 1982), and finite state machines (Parnas 1969, Woods 1970, Foley and Wallace 1974, Moran 1981). Other kinds of formalism that have been used include pushdown automata (Olsen 1984), and special purpose programming languages (Kasik 1982).

Some systems express the formalism using a textual language, while others provide an interactive editor for constructing the specification. The machine-readable form of the specification can be used in a variety of ways; typically it is interpreted by a run-time system, either for prototyping or in actual use, or it is compiled to produce user interface code for linking with the application.

A large number of user interface development systems have been built incorporating various combinations of these techniques. Some examples are CHIDE (Hardwick, Uejo, Spooner, Czechowski, Lohr and Sarachan 1991), SYNGRAPH (Olsen 1984), TIGER (Kasik 1982), FLAIR (Wong and Reid 1982), and COUSIN (Hayes and Szekely 1983). A survey of many more can be found in Hartson and Hix (1989).
Most of these systems deal only with conversational-world interfaces, where there is a single thread of dialogue, input and output are loosely coupled, and the user is kept at a considerable distance from the application semantics. Under these conditions it is relatively easy to separate the user interface and specify it in a formal way.

By contrast, formal specification of model world interfaces appears to be much more difficult. Partly this is because the formalisms used to describe sequential dialogues tend to become unwieldy when applied to multithreaded dialogues. Some progress has been made on this problem; for example, Jacob (1986) describes a formal framework in which a multithreaded dialogue is decomposed into multiple concurrent single-threaded dialogues, each of which is described by a separate state transition network.

A more fundamental problem is that, in a model world environment, the concept of dialogue independence is hard to pin down. The very purpose of a model world interface is to bring the user into closer contact with the application semantics, which seems to be directly opposed to the goals of dialogue independence.

Many authors testify to the difficulty of achieving dialogue independence in the model world (Hartson and Hix 1989, Hurley and Sibert 1989, Hartson 1989). The Macintosh Finder provides an example of the kind of dilemma encountered (Hartson 1989, Myers 1989). When the user is dragging a file icon and it passes over a folder icon, the folder is highlighted to remind the user that releasing the file at that point will move it into the folder. If it passes over another file icon, however, no highlighting occurs.

If the highlighting is to be handled entirely by the user interface, it must know something about the semantics of files and folders, possibly compromising dialogue independence. On the other hand, if the user interface knows nothing about these semantics, continual communication with the computational part of the application is required during the dragging operation, making the protocol between the two more complicated.

On a more abstract level, we can ask which parts of the Finder constitute its dialogue-independent functionality, and which belong to its user interface. It could be argued that the purpose of the Finder is to provide a way for the user to view and manipulate the file system, and the way files and directories are represented - as windows containing icons, lists of names, or something else - is strictly a property of the user interface.
Alternatively, it could be argued that representing files and directories as icons, and letting the user drag them around, is what the Finder is all about. In other words, the Finder is essentially just a user interface to the file system, performing little or no computation.

Both of these viewpoints lead to the same conclusion: that almost all of the Finder resides on the user interface side of the division, and very little on the computational side. Also, some semantics clearly must be present in the user interface. The dialogue part must either know about files and folders, or know that it needs to know, so that it can find out from the computational part when necessary.

Many of the commonly found applications with direct manipulation interfaces, such as word processors and drawing editors, share with the Finder the property that they are mostly user interface. If one attempts to draw a clear boundary between dialogue and computation in these applications, one finds that – compared to the dialogue part – the purely computational part is very small.

Other applications, such as spreadsheets and database management systems, have computational parts that are more substantial. Nevertheless, the user interfaces to these applications that are typically provided in a model world environment make up a large proportion of the code and functionality.

Hartson (1989) acknowledges the need for some semantics in the user interface. He divides an application into three parts - input (I), display (D), and computation (C) (Figure 3.1). Ideally, I and D would belong exclusively to the dialogue part and C to the computational part. He recognises, however, that there is need for some computation in the dialogue part, represented in Figure 3.2 by including some C and D with I, and some C with D.

He then describes the problem of achieving dialogue independence as one of finding, in any particular case, the right amount of computation to include in the dialogue, so that communication between the two parts is minimised, while not bringing excessive semantics into the dialogue component.

The main difference between applications with conversational world versus model world interfaces is this: Conversational world interfaces tend to be very simple in structure, and have very little demand for computation. That computation which they do require can usually be supplied in a "prepackaged" form, such as simple text editing, or by borrowing a simple semantic function from the computational part, such as for validation of an input value. Of the total functionality provided by the application,
only a small proportion is in the dialogue component. Given these characteristics, it is relatively easy to devise a means of declaratively describing the dialogue part and its interaction with the computational part.

A model world application, by contrast, has a dialogue component that is much larger, and much more computationally demanding. A substantial fraction of the application’s functionality is often implemented in the dialogue component. Moreover, not only is there a heavy need for computation in the dialogue, but the kinds of computation required are as diverse as the applications themselves.

It follows that any formalism or tool for specifying or implementing the dialogue component of a model-world application requires the power of a programming language. Indeed, all of the widely-used tools for constructing model world interfaces are, in fact, based on general-purpose programming languages.

A particularly successful class of these tools, based on object-oriented programming languages, will be examined in the next section.
3.2.3 The framework approach

The techniques of object-oriented programming are well suited to the task of implementing model-world dialogues. Entities with which the user interacts are easily seen as objects in the object-oriented sense, to which input events such as mouse clicks and keystrokes are directed. Each object responds to the events it receives independently of the state of other objects, allowing multithreaded dialogues to be handled in a natural way.

The object-oriented toolkit or framework is a library of user interface components, structured in an object-oriented fashion, from which the programmer can construct the dialogue component of an application. Typically the toolkit provides a set of objects which implement windows, menus, buttons and so forth in some particular style, and a framework which ties them together and takes care of marshalling input events to the relevant objects. The programmer can either use these objects as supplied, or create customised versions of them, by means of inheritance and specialisation, for the application’s unique requirements.
Usually the toolkit, the dialogue component of the application, and the computational component of the application are written in the same language. There is no sharp boundary between dialogue and computation enforced by the methodology or tools, although dialogue independence principles can often be discerned in the structure of the toolkit and its interface with the application.

The remainder of this section will briefly describe the main features of a selection of object-oriented user interface frameworks, to convey the general flavour of these systems and how they are used.

**Smalltalk**

The Smalltalk-80 system (Tesler 1981, Goldberg and Robson 1983, Krasner and Pope 1988) can be regarded as a single, large object-oriented user interface toolkit. In Smalltalk, applications are not separate entities; rather, one tailors the system for an application by adding new classes of object designed for the task at hand.

The Smalltalk framework is based on three object classes called the Model, the View and the Controller, connected as shown in Figure 3.3. The View and Controller are dialogue objects, with the View handling graphical output and the Controller handling input. The Model is a computational object, embodying the application’s data structures and associated operations.

Views are arranged in a hierarchy. The example in Figure 3.4 is taken from Krasner and Pope (1988), and illustrates how simple dialogue objects can be composed to form more complex ones. The classes StandardSystemView, SwitchView and Button are standard, systems-supplied ones, whereas Counter and CounterView have been defined by the programmer by subclassing the generic Model and View classes. (The Controller object associated with each View is not shown.)

Figure 3.5 shows a fragment of the Smalltalk code which sets up the object hierarchy of Figure 3.4.

**SunView**

SunView (Sun 1988b) is a graphical user interface toolkit supplied by Sun Microsystems for use on their workstations. It consists of a run-time library linked with application programs, supported by a small amount of code in the Unix kernel to handle low-level input and output.
The SunView library is designed for use with C programs. Although C does not directly support object-oriented programming, SunView is structured to some extent in an object-oriented way. A fixed set of "classes" of user interface components is provided, which may be assembled in certain ways. Figure 3.6 depicts the SunView class hierarchy and shows how objects can be combined.

A part of SunView called the notifier handles input in an object-oriented way. The application program may associate a callback procedure with an event type and an object. The notifier will call the procedure when an input event of that type is directed to that object.
"Create the main model"

aCounter <- Counter new.

"Create auxiliary models used in the dialogue"

incrButton <- Button newOff.
incrButton onAction: [ aCounterView model increment ].
decrButton <- Button newOff.
decrButton onAction: [ aCounterView model decrement ].

"Create and assemble the views"

topView <- StandardSystemView new label: 'Counter'.
aCounterView <- CounterView new model: aCounter.
topView addSubView: aCounterView
  in: (0.4@0 extent 0.6@1)
  borderWidth: 0.

incrSwitchView <- SwitchView new model: incrButton.
incrSwitchView label: ('+' asDisplayText form magnifyBy: 2@2).
topView addSubView: incrSwitchView
  in: (0@0 extent: 0.4@0.5)
  borderWidth: (0@0 extent: 2@1).

decrSwitchView <- SwitchView new model: decrButton.
decrSwitchView label: ('-' asDisplayText form magnifyBy: 2@2).
topView addSubView: decrSwitchView
  in: (0@0.5 extent: 0.4@0.5)
  borderWidth: (0@1 extent: 2@0).

Figure 3.5
Smalltalk code for creating the view hierarchy of Figure 3.4.

Typically an application will create the user interface objects it needs, attach callback procedures to them, and then pass control to the notifier, which loops reading input events and calling the appropriate callback procedures. Figure 3.7 shows part of the code of a simple SunView application and the appearance of the window it creates.

MacApp

MacApp (Schmucker 1986a,b) is an object-oriented user interface toolkit for the Macintosh. It is written in Object Pascal, a dialect of Pascal extended with object-oriented facilities. Object Pascal was developed jointly by Apple Computer and Niklaus Wirth (Schmucker 1986b).

MacApp consists of a collection of classes ("object types" in Object Pascal terms) which, together with the Macintosh Toolbox code in ROM, implement all the
standard functionality of the Macintosh user interface. On its own, MacApp constitutes the shell of a complete Macintosh application.

The programmer uses MacApp by extending it with new object types that implement application-specific user interface and computational functions. In this respect MacApp is akin to Smalltalk, and differs somewhat from SunView: the SunView library cannot be extended by the programmer in the same object-oriented style.

Figure 3.8 illustrates the MacApp class hierarchy (in MacApp there is a convention of beginning object type identifiers with the letter “T”) and an example of how MacApp objects are assembled to form a user interface.
A simple SunView application (abridged)

```c
main() {

    frame = window_create(NULL, FRAME,
           FRAME_LABEL, "hello_world_panel",
           0);

    panel = window_create(frame, PANEL, WIN_FONT, bold, 0);

    panel_create_item(panel, PANEL_MESSAGE,
           PANEL_LABEL_STRING, "Push button to quit.", 0);

    panel_create_item(panel, PANEL_BUTTON,
           PANEL_LABEL_IMAGE, panel_button_image(panel, "Good-bye", 0, 0),
           PANEL_NOTIFY_PROC, quit_proc,
           0);

    window_fit(panel);
    window_fit(frame);
    window_main_loop(frame); /* Enter the notifier */
}

static void quit_proc() {
    /* Called when the quit button is pressed */
    window_destroy(frame);
}
```

Figure 3.7a
Part of the code of a simple SunView application (from Sun 1988b page 39)

<table>
<thead>
<tr>
<th>hello_world_panel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push button to quit. Goodye</td>
</tr>
</tbody>
</table>

Figure 3.7b
Window created by the code of Figure 3.7a

**InterViews**

InterViews (Linton, Vlissides and Calder a,b) is an object-oriented user interface library written in C++ (Stroustrup 1986). Of the frameworks described here, InterViews appears to be particularly flexible and easy to use.
Chapter 3 - Graphical User Interfaces

Figure 3.8a
Part of the MacApp class hierarchy

Figure 3.8b
Assembling MacApp objects
Class *Interactor* provides a base class for a comprehensive set of user interface objects, such as buttons, menus, scrollbars and text boxes. There are also several types of Interactor which can contain other Interactors, such as *HBox* and *VBox* which arrange their contents horizontally or vertically, and *Frame* for providing a margin or border around another Interactor.

Dialogue independence principles can be seen in the division of objects into two groups, *views* and *subjects*, with views being dialogue objects and subjects being computational objects. A view thus correspond to Smalltalk View-Controller pair, and a subject to a Smalltalk Model.

View-subject constructs are also used to good effect within the user interface framework. For instance, a *Button* is a view with a *ButtonState* as its subject; interlocked groups of buttons are implemented by a set of *Button* objects viewing the same *ButtonState*.

As another example, scrolling is implemented by attaching a *Perspective* object to a view. Scroll bars are view objects having the Perspective as a subject; interacting with either scroll bar changes the Perspective and thus causes the view to scroll. This arrangement permits considerable flexibility. For instance, a different type of scrolling device can be substituted for the scroll bars, or even used in addition to them, without any change to the view object being scrolled.

### 3.3 Display server systems

With the growth in popularity of personal workstations attached to a network of computing resources, it has become desirable to be able to run applications with graphical user interfaces on machines other than the workstation where the graphical input and output devices reside.

Usually this is achieved by running a *display server* program on the workstation to which *client* programs can connect through the network. The display server provides a device-independent protocol for clients to perform graphical input and output, and coordinates access to the screen by simultaneously running clients.

This section describes the main features of two display server systems, the X Windows system and Sun Microsystems' NeWS. The NeWS system will be discussed in particular detail to provide background for the next two chapters.
3.3.1 X Windows

The X Windows system arose out of Project Athena, which was begun in 1984 at the Massachusetts Institute of Technology (MIT). In 1988 an association of computer vendors, the X Consortium, was formed to support the development of X. Since then, X has been adopted by many hardware and software vendors, and has become regarded as a de facto standard.

The X Windows system comprises several layers of software as depicted in Figure 3.9. At the bottom level is the X server, which runs on the user’s workstation and provides a standard set of graphical input and output operations.

The server maintains a collection of windows. Each window is a region of the screen within which drawing operations can be performed, clipped to the window boundary. Windows may overlap, can contain subwindows, and may be made sensitive to input.

Client programs communicate with the server over the network using a standard protocol called the X protocol. The client side of the X protocol is implemented by a library called Xlib which contains a routine for each of the operations that the client can ask the server to perform.

Clients may use Xlib directly, but because the facilities it provides are low-level, its use is tedious for all but the simplest user interfaces. Most clients make use of another library, built on top of Xlib, providing higher-level facilities.

The X toolkit intrinsics library, or Xt, provides a framework for constructing object-oriented toolkits in C. Several libraries based on Xt are available that provide user interface components such as scrollable windows, menus, buttons and so forth, in a way similar to the object-oriented toolkits already discussed. These libraries differ mainly in the style or “look and feel” of user interfaces constructed with them.

Other user interface toolkits are also available that are not based on Xt. For example there is an implementation of the InterViews library (described above) that is built directly upon Xlib.
3.3.2 NeWS

The *Network extensible Window System*, or NeWS (Sun 1987, 1988a) was developed by Sun Microsystems in response to some perceived shortcomings in the design of the X window system, as discussed in detail by Rosenthal (1986), Gosling (1986) and Sun (1987).

The main difference between X and NeWS is that whereas the services provided by the X server are fixed\(^1\), the NeWS server is *extensible*. NeWS is based on the PostScript graphics programming language defined by John Warnock and Charles Geschke at Adobe Systems (Adobe 1985a,b).

The NeWS server contains a PostScript interpreter, and client programs send commands to the server in the form of pieces of PostScript code to be elaborated. Since PostScript is a Turing-equivalent programming language, procedure definitions can be sent to the server, allowing clients to extend the facilities of the server dynamically according to their needs.

---

\(^1\)In X11 there is a method specified for making extensions to the X protocol and X server, but it is much less flexible and dynamic than that of NeWS, and its use is very rare.
Some advantages of using PostScript as a communication and extension language are:

(1) The PostScript graphics model is very high-level and powerful, and is completely device-independent, allowing complex images to be described easily, and advantage to be taken of advanced graphics hardware. In contrast, the X graphics model is rather low-level and hardware-oriented.

(2) Each client can effectively define its own communication protocol with the server, allowing very efficient utilisation of communication bandwidth between server and client. There are two ways in which bandwidth can be saved:

- By economy of expression. For example, telling the X server to draw a grid requires sending the coordinates of the endpoints of every line. Telling the NeWS server to do the same thing requires only a short, simple loop.

- By eliminating the need to communicate at all. Interaction techniques such as dragging require much low-level communication between the X server and a client. In NeWS, code to handle the feedback can be downloaded into the server and executed there, with only the final result being sent back to the client.

These savings in communication have the potential to improve interactive responsiveness in environments where network bandwidth is a scarce resource.

(3) A comprehensive body of user interface library code can be loaded into the server and shared between all clients. The centralisation and dynamic nature of this library opens up possibilities for customising user interface features in a far more flexible way than could be achieved in X.

*NeWS extensions to PostScript*

PostScript was originally designed for output-only single-tasking devices such as printers. NeWS adds some concepts, data types and operators to the language to make it more suitable for use as the basis of a display server. The most important new types are the *canvas*, the *event* and the *process*. 
• **Canvases**

A *canvas* plays the role of the PostScript “current page” for graphics output. Canvases may be of arbitrary shape and overlap, and form a hierarchy, similar to windows in the *X* server.

• **Processes**

The NeWS server manages a collection of “lightweight processes”, or separate threads of execution of PostScript code. Each process has its own context comprising an execution stack, an operand stack and a dictionary stack. The *fork* operator creates a new process which executes in a copy of the current context. Processes can communicate by sending *events* to each other (see below) or using shared variables guarded by *monitor* objects.

• **Events**

Events are created in response to input actions such as mouse clicks, mouse motion and key presses. Flexible means are provided for processes to express interest in and receive various classes of events. A process can also create an event and send it to itself or another process.

*Client-server communication*

When a client first makes connection with the server, a process in the server is awoken which creates a new process to handle communication with that client.

The connection between server and client is a bi-directional data stream. The client requests the server to perform an operation by writing a fragment of PostScript code to its end of the stream. At the server, the client’s PostScript process runs in a loop, reading fragments of PostScript from the server end of the stream and executing them.

Communication in the other direction is in the form of messages written to the data stream by PostScript processes in the server, and read and acted upon by the client. Each message comprises a *tag* defined by the programmer identifying the type of message, possibly followed by one or more data items. Facilities are provided for encoding and sending tags and data values, and for receiving and decoding them in the client.
Typically a client will create one or more extra PostScript processes for receiving user input and either handling it or relaying it to the client. Figure 3.10 depicts communication paths between the server, a client, and a pair of PostScript processes belonging to the client.

**Tools for constructing client-server interfaces**

NeWS comes with a tool called `cps` to aid in the specification and implementation of interfaces between clients and the server. Cps takes a file of procedure definitions which associate a C procedure name with a fragment of PostScript. Cps generates a set of macros which may be called from a C program.

Figure 3.11 shows an example of a cps definition, illustrating three basic kinds of cps procedures. The first kind is a procedure which returns no result; calling it simply causes the corresponding PostScript code to be sent to the server, with arguments substituted as appropriate.

The second kind is a function which returns a result. After sending the PostScript, the macro will wait until a message is received back from the server containing the result. The part after the "=>" specifies the tag value which the result message will have, the data items which will follow it, and the arguments in which their values will be returned. In the body, the operators `tagprint` and `typedprint` are used to transmit the result.

The third kind is a polling function used to receive other messages, such as responses to input events, sent by code in the server. If a message with the given tag value has been received when the function is called, it will return `true`, otherwise it will
return false. Typically the main loop of a client repeatedly calls each of its polling functions in turn until one of them returns true and then acts on the message received.

3.4 Summary

User interface metaphors can be divided into two classes, conversational world and model world. Conversational world interfaces are usually simple enough for an application to be divided into dialogue and computational parts in accordance with the principles of dialogue independence. The dialogue part and its interaction with the computational part are easily specified using a declarative formalism that can be used to automatically derive an implementation.

In contrast, model-world interfaces are more elaborate and interact much more closely with the computational parts of the application, making them difficult to specify formally in a way that does not involve programming. Most successful systems for building model-world user interfaces are based on an object-oriented programming language, and provide a library of dialogue and computational objects which the application programmer uses and extends to construct an application.

In a network environment, it is often desirable to interact graphically with applications running on remote machines. To accomplish this, a display server is run on the user’s workstation, and client programs running on the same or different machines communicate with the server.
Two very different approaches to the design of display server systems are exemplified by the X Windows system and the NeWS system. The X server provides a fixed set of low-level graphics facilities to client programs; more complex facilities are provided by user interface libraries linked with each application.

The NeWS server, in contrast, is extensible. Based on the PostScript graphics programming language, NeWS provides much higher level graphics facilities than X. More importantly, it can be dynamically extended by downloading PostScript procedures, allowing clients to define their own application-specific communication protocol to make the most efficient use of network bandwidth.

NeWS encourages much of the user interface library code, that in X would be linked with clients, to be centralised in the NeWS server and shared between all clients. This opens up interesting possibilities for user interface customisation by modifying or replacing the interface code.
Chapter 4

P: An Alternative to Handwritten PostScript

When the map browser project was begun, the only graphical user interface environments readily available were SunView and NeWS. SunView had the advantage of being well established and supported, but the facilities provided for rendering images were rudimentary. The “pixrect” library on which SunView is based contains routines for drawing single-pixel wide straight line segments, bitmap images, and horizontal text in a fixed set of sizes. There is no direct support for drawing curved shapes or arbitrary filled areas.

Being based on PostScript, NeWS offers a much richer graphics environment in which straight and curved lines of any width and style and filled areas of any shape may be easily specified, and images can be scaled and rotated. Text is treated the same way as other images, allowing text to be drawn in any size and at any angle.

These characteristics make the PostScript graphics model very attractive for mapping applications. Furthermore, the use of PostScript allows the possibility of using the same description of a map to generate the screen display and produce a printed copy. For these reasons, NeWS was chosen as a basis for the user interface of the map browser.

The versions of NeWS available at the time (NeWS 1.0, later 1.1) were in an early stage of development. A user interface library called LiteWin was supplied, written in PostScript, providing some of the facilities desirable to an application programmer, but not all. Some of the missing facilities were essential for the intended mapping application.

Consideration was given to extending LiteWin in order to supply the missing facilities. However, it was found that LiteWin was written in a style that did not make for either easy understanding or straightforward extension. It appeared that it would be easier to start again from scratch, and build a library with extensibility in mind from the outset.

The author’s experience with writing this library brought to light the difficulty of
writing large amounts of PostScript code by hand. The PostScript language was designed for use as a device-independent page description language for laser printers and other high-resolution graphical output devices. It was intended mainly as a standard intermediate form to be generated by other programs such as word processors and desktop publishing systems. It has a very simple, postfix syntax that is easy for programs to generate and parse, but not so easy for people to write and read.

This chapter discusses some of the shortcomings of PostScript as a programming language for people, and describes a new language, P, which the author has devised to overcome them. The language was designed primarily for use with NeWS, but it is suitable for any application that requires manually-written PostScript code.

The reader is assumed to have some familiarity with the PostScript language.

4.1 Problems

The main difficulty encountered in writing PostScript programs manually is the problem of visualising and keeping track of the state of the stack over long sequences of operations. This is made worse by the fact that there is nothing to ensure that the correct number and type of parameters are passed to a procedure, or that the stack is left in the correct state on return.

Consider the following pseudo-code procedure to find the roots of a quadratic equation. It takes the three coefficients as parameters and returns a two-element array containing the roots. (For simplicity it is assumed that the roots are real.)

```plaintext
procedure qroots(a,b,c)
    local d,r1,r2
    d := sqrt(b*b-4*a*c)
    r1 := (-b+d)/(2*a)
    r2 := (-b-d)/(2*a)
    return r1,r2
```

Providing named parameters and local variables in PostScript is cumbersome, as it involves pushing a new dictionary onto the dictionary stack, popping the parameters (in reverse order) and storing them in the dictionary, and making definitions for any local variables. A direct translation of the above algorithm into PostScript might be:

```postscript
/qroots { 6 dict begin % a b c => [root1 root2] /c exch def % Pop and store parameters /b exch def /a exch def /d b b mul 4 a c mul mul sub sqrt def % Calculate determinant /r1 b neg d add 2 a mul div def % First root
```
Because of the tedious procedure required to declare named parameters and local variables, and the run-time overhead incurred, one is tempted to dispense with these, simply leaving parameters and local variables on the stack and using \texttt{index} to reference them. Using this technique, \texttt{qroots} can be rewritten in a more efficient, albeit less readable, style as follows:

\begin{verbatim}
/qroots { 1 index 2 index mul 4 5 index 3 index mul mul sub sqrt [2 index neg 1 index add 2 4 index mul div 4 index neg 3 index sub 2 6 index mul div] 5 -1 roll 4 (pop) repeat } def
\end{verbatim}

4.2 Solutions

The problems of programming in PostScript have much in common with those of assembly language. The assembly language programmer typically has a stack which can be manipulated with great freedom, and a set of very loosely-typed operators and data structures. This freedom both permits great efficiency and allows tremendous scope for obscure errors.

Very effective solutions to the problems of assembly-language programming have been in use for some time in the form of high-level programming languages. Clearly a higher-level language was needed that could be translated easily into PostScript.
4.2.1 An alternative syntax

The author considered the possibility of translating an existing high-level language, such as C or Pascal, into PostScript. However, each of these languages has its own set of abstractions, which cannot always be mapped easily or efficiently onto those of PostScript. The intention was not to supplant the control and data structures of PostScript, but to provide an easier and clearer means of expressing them.

To that end, a new language, P, was designed, mainly for the purpose of writing toolkits and applications for NeWS, although it can be used in any situation requiring hand-written PostScript code. A translator converts P source into PostScript source, which can then be interfaced with a NeWS application program using Sun's cps utility as described in Chapter 3.

The control and data structures of P are the same as those of PostScript, but the syntax is more traditional and Algol-like. Some of the main features of the P syntax are listed below. A concise summary can be found in Appendix A; for more detailed information, see (Ewing 1988).

• Expressions involving arithmetic, relational and assignment operators are written in the usual infix style. There are two assignment operators which correspond to the PostScript def and store operators:

\[
\begin{align*}
a & := 5 & \text{/* def */} \\
a & := b*(c+d) & \text{/* store */} 
\end{align*}
\]

• A conventional procedure calling syntax is used, with the procedure name followed by the parameters in parentheses:

\[
grobulate(f,g)
\]

If there are no parameters, the parentheses may be omitted.

• The conventional indexing notation may be applied to arrays, strings and dictionaries:

\[
a[i] := b[j]
\]

A variant of this notation is used to denote subranges. The following extracts \( k \) elements of \( b \) starting at \( j \) and places them in \( a \) starting at \( i \):

\[
a[i@] := b[j@k]
\]

• Parameters and local variables may be declared at the beginning of any code
block, and the appropriate dictionary-manipulating code is automatically generated. A procedure is declared by defining a name as a code block with parameters. The following declares a procedure with two parameters a, b and two local variables c, d:

```
grobulate ::= { |a,b|  
   local c,d;  
   ...  
 }
```

- All the PostScript control structures are provided with a traditional syntax, for example:

```
if a>b then {  
c:=a  
} else {  
c:=b  
}
```

- The iteration control structures make use of a block with parameters, for example:

```
for 1 to 10 do [|i|]  
  /* Do something with i */  
}
```

- NeWS makes extensive use of object-oriented programming techniques. P provides a class-defining construct which allows the definition of class variables and methods, instance variables and methods, and inheritance from a superclass. Methods are invoked by means of a dot-notation. The statement

```
window.set_title("Hello world")
```

invokes the method set_title of the object window with the parameter "Hello World".

- Occasionally something needs to be done in PostScript that cannot be expressed easily using an infix expression syntax. On these occasions it is possible to "escape" from P into raw PostScript. Since any P statement can be an expression, any sequence of constants and symbols may be written separated by semicolons, and will generate an exactly equivalent sequence of PostScript tokens.

For instance, the PostScript operator stringwidth returns two results on the stack, the x and y components of the string's dimension. Often only the x component is wanted; the following P fragment directly expresses the PostScript idiom employed for this:

```
s; stringwidth; pop  /* Get the x-component of the width of s */
```
To allow any valid PostScript name to be included in a P program, a sequence of characters enclosed in ‘...’ (a backquote and a single quote) is treated syntactically as an identifier.

4.3 Examples

4.3.1 qroots

Here is a P version of the qroots procedure presented earlier.

```
qroots ::= \{a,b,c\}  /* Parameters */
local d,rl,r2;  /* Local variables */
d := sqrt(b*b-4*a*c);
rl := (-b+d)/(2*a);
r2 := (-b-d)/(2*a);
[rl,r2]  /* Return value */
```

This P version is obviously much more readable than either of the PostScript versions. Although the current implementation of P does not prevent the programmer from misusing the stack or passing incorrect parameters to a procedure, the improved syntax makes it much less likely that this will be done accidentally.

4.3.2 Elliptical Arcs

The following program is taken from the example on page 137 of the PostScript Language Tutorial and Cookbook (Adobe 1985a). It defines a procedure for drawing elliptical arcs and gives an example of its use.

PostScript version:

```
ellipsedict 8 dict def
ellipsedict /mtrx matrix put

/ellipse {
  ellipsedict begin
  /endangle exch def
  /startangle exch def
  /yrad exch def
  /xrad exch def
  /y exch def
  /x exch def
  /savematrix mtrx currentmatrix def
  x y translate
  xrad yrad scale
  0 1 startangle endangle arc
  savematrix setmatrix
  end
} def

newpath
```
400 72 144 0 360 ellipse
stroke

The following P version is not quite a literal translation, since the use of P's named parameter mechanism makes the creation of a globally-stored dictionary for local variables unnecessary.

```
ellipse :={ |x,y,xrad,yrad,startangle,endangle |
    local savematrix=currentmatrix(matrix);
    translate(x,y);
    scale(xrad,yrad);
    arc(O,O,1,startangle,endangle);
    setmatrix(savematrix);
};

newpath;
ellipse(144,400,72,144,0,360);
stroke
```

4.4 Experience so far

Using the Unix parser-generating tools lex and yacc, the P translator was easily implemented on a Sun workstation, mainly due to the very straightforward mapping between P constructs and PostScript constructs. The author has written a considerable amount of code in P, including an experimental user interface library for NeWS. This experience has indicated that P is very successful at making the power of PostScript easily available to the programmer.

4.5 Future potential

There are several enhancements that could be made to the language, its current implementation, and the supporting environment.

4.5.1 Type checking

More protection against programmer errors could be provided by checking the number and type of parameters passed to procedures. With the current definition of the language this would require run-time type checking, because of the dynamic typing inherited from the underlying PostScript execution model. Run-time type checking might be made a translator option which could be turned off once a program had been debugged.

Alternatively, static typing could be introduced into the language so that more comprehensive type checking could be carried out at translation time. This would require an extensive revision of the language design and make the translator more complex.
4.5.2 Implementation

Named parameters and local variables could be implemented more efficiently. The present scheme of using dictionaries makes for easy translation but not for efficient execution. It also has potential to cause difficulty with some PostScript implementations where the memory manager is unable to automatically reclaim memory. A more intelligent translator would leave local variables on the stack and reference them using the \texttt{index} operator, which would solve both problems.

4.5.3 Debugging support

NeWS needs a run-time source-level debugging environment, allowing code to be traced, breakpoints inserted and variables examined during execution. The PostScript debugger supplied with NeWS 1.1 goes some way towards providing these facilities, but it is not a source-level debugger in the sense of relating errors and breakpoints to code and variables in the original source file.

To implement such a debugger, information is required at run-time about the source file and its relationship to the object code being executed. No such information is at present available to the NeWS debugger, partly because there is no provision to collect it, and partly because there is little structural information present in PostScript code that would be of use.

By contrast, P source code contains much more structural information, and the P translator provides a convenient point at which to collect and analyse this information. The P translator should be able to provide enough information to support a debugging system with capabilities comparable to, for instance, the \texttt{dbx} debugger in Berkely Unix.

4.6 Summary

A language has been designed to make the facilities of the PostScript graphics language available to programmers in an easy-to-use form. The language has been implemented and used to write a user interface toolkit for NeWS. This experience has shown it to be an effective tool for creating understandable and error-free programs. There is potential for future developments of the language and implementation to provide even more protection against error, and to support a more comprehensive debugging environment than would be possible for PostScript alone.
Chapter 5

A User Interface Toolkit for NeWS

This chapter describes the Mockintosh, a user interface toolkit for NeWS which the author developed and used to construct the user interface of one version of his experimental map browsing system.

Sections 5.1 and 5.2 describes the general structure of the toolkit from the user's and programmer's viewpoints and shows some examples of its use. Section 5.3 presents conclusions that the author has drawn from the experience of building and using the toolkit, and makes some suggestions for further development.

5.1 The user's view

The author chose the name Mockintosh because of its visual similarity to the Macintosh user interface (Apple 1985). The Macintosh interface was chosen as a model because it is familiar to many people and, in the author's opinion, it is graphically appealing.

The main parts seen by the user are the desktop, the menu bar and a collection of windows. Within a window may appear various standard user interface components as well as application-specific displays and controls.

The desktop is a rectangular region upon which windows appear. Windows may overlap and may be moved and resized by the user by means of controls provided around the window border.

Above the desktop is the menu bar, a strip containing the titles of pulldown menus. Clicking on a menu title causes the menu to appear so that a command may be chosen from it.

In the Macintosh, the menu bar is associated with the currently running application, and (usually) changes only when switching applications. To fit in better with the multitasking environment of Unix, the Mockintosh associates a separate set
of menus with each window; when a window is made active, its menus are installed in
the menu bar at the top of the screen.

There are two kinds of windows, ordinary windows and dialogues. An ordinary
window usually contains a viewport for displaying application data, possibly
associated with scroll bars for changing the viewed area. Dialogue windows usually
contain a set of controls, such as pushbuttons, check boxes, radio buttons and text
entry boxes.

Dialogues may be modal or non-modal. A modal dialogue requires that the user
deal with and dismiss it before switching to another activity, whereas non-modal
dialogues do not.

Figure 5.1 depicts a typical Mockintosh display illustrating many of the standard
components.

5.2 The programmer's view

From the application programmer's viewpoint, the Mockintosh consists of an
object-oriented server-side library of user interface components written in P, and a
client-side library written in C that provides access to the server-side components
and mediates communication between the server and client. Figure 5.2 illustrates the
overall structure of the Mockintosh and one of its clients.

5.2.1 The class hierarchy

Figure 5.3 shows the class hierarchy of the server-side objects, of which the most
important are described briefly here.

Class “Gadget”

Gadget is the ancestor class of all objects which can appear on the screen. Each
Gadget has its own canvas and a lightweight process to handle input events directed
to its canvas. A Gadget may have subgadgets attached to it; the canvas of a subgadget
is a child canvas of its parent Gadget's canvas, so that a subgadget is clipped to the
boundary of its parent.

Class “Control”

Control is an abstract class providing a basis for various items found in dialogue
boxes and elsewhere, such as buttons, check boxes and text entry areas. A generic
Control has a title, a value and a highlighting state, although not all Controls use all of
these attributes. A Control also has a client, a Gadget which is notified when the control is manipulated by the user. If the client object is null, a message is sent over the communication link to the application.

Figure 5.4 illustrates the appearance of various kinds of controls.
Figure 5.2
Overall structure of the Mockintosh and a client

Figure 5.3
Mockintosh Class Hierarchy

Figure 5.4
Types of Controls
Class "Button"

Button is a descendant of Control that provides behaviour common to various types of button. When a Button is clicked, it is highlighted as long as the mouse remains within it. If the mouse is released within the button, its client is notified, otherwise the click is ignored.

Subclasses of Button include PushButton, CheckBox and RadioButton. Radio buttons come in interlocked groups managed by the class RadioButtons.

Class "Label"

A Label is a Control used to provide labelling text in a dialogue box, and is completely passive.

Class "Slider"

A Slider is identical to a ScrollBar but has the message protocol of a Control so that it may be used as an item in a dialogue window.

Classes "View" and "Port"

These two classes cooperate to provide a scrollable area in which the application may display an image. The View ties together the Port and (optionally) a horizontal and vertical scroll bar, and takes care of all the details involved in scrolling.

By default, class Port sends all mouse, keyboard and damage events to the application for it to handle. If some or all of these events are to be handled in the server, a subclass of Port must be created.

Classes "TextEditPort" and "TextEditView"

A TextEditPort provides basic mouse-oriented text editing facilities supporting cut-and-paste operations. A TextEditView provides a scrollable text editing area.

Classes "Menu" and "MenuBar"

These two classes cooperate to provide a menu bar of pull-down menus. A Menu is created from a list of items, consisting of the name of the item and, optionally, an action to be performed when the item is selected. If no action is specified, a message is sent to the application.
A MenuBar is created from a list of Menus. In addition to the menus supplied, a standard system menu named "*" is added at the left of each menu bar. This menu is analogous to the "Apple" menu on the Macintosh in that its purpose is to include various system-related commands that should always be available. Currently it contains one item, the "Shutdown" command for shutting down the entire Mockintosh. Other things that it might profitably contain include a list of frequently-used applications that the user can configure.

Class "ScrollBar"

A ScrollBar normally has a View or a Slider as its client. When the user manipulates the scroll bar, a message is sent to its client Gadget, or if none, over the network to the client application. Figure 5.5 illustrates ScrollBars.

Classes "Window" and "Dialogue"

A Window provides a standard Macintosh-style window, with optionally a title bar, a "close box" for closing the window, and a "grow box" for changing its size. Typically the application will attach one or more other Gadgets to the window, such as a View.

A Dialogue is a descendant of Window specialised for implementing dialogue boxes. When a Dialogue is created, it may be given a list of item specifications describing Controls to be created and attached to the Dialogue.

5.2.2 Composing objects

Mockintosh objects are designed to be "plugged together" to construct more complex user interface components. Figure 5.6 is an exploded view of a typical window with a scrollable viewing area, showing how it is constructed from a Window and a View; the View is in turn composed of a Port and two ScrollBars. Figure 5.7 shows a P procedure that could be used to create this window.
5.2.3 The C Library

To make the Mockintosh library more easily usable from an application written in C, a library of C routines was created to provide interfaces to the standard Mockintosh facilities. The C library consists of three parts, the message routines, the menu and dialogue routines and the notifier.
Experiments in Interactive Map Retrieval

Figure 5.7

P procedure to construct the window of Figure 5.6

```p
MakeMapWindow := {
  local win, view, port;

  /* Create the window and set its size and position */
  win := Window.new(['TitleBar', 'CloseBox', 'GrowBox']);
  win.Shape(100, 100, 400, 300);

  /* Create the view */
  view := View.new(win, ['HScrollBar', 'VScrollBar', 'GrowBox']);

  /* Create the port and add it to the view */
  port := Port.new(view);
  view.SetPort(port);

  /* Make the window visible */
  win.Show;

  /* Return the window */
  win
}
```

Figure 5.8

Example of a message-sending stub and associated cps macro definition

```p
/* C message routine */
Size(id, w, h) int id, w, h; {
  ps_Size(id, w, h);
}

/* CPS definition */
cdef ps_Size(id, w, h) w h /Size id IDToHandle send
```

5.2.4 Message routines

This section consists of a set of stub routines, one for each message understood by a Mockintosh object. Each stub invokes a cps macro that sends the appropriate PostScript fragment to the NeWS server.

So that the application can refer to objects residing in the server, each Gadget is assigned a unique id number upon creation. Two PostScript procedures, IDToHandle and HandleToID, are used by message routines to convert between ID numbers and the objects they represent.

Figure 5.8 shows a typical stub, the cps macro it calls.
int MakeExampleWindow() {
    int winID, viewID, portID;
    winID = NewWindow
        (WIN_TITLE_BAR+WIN_CLOSE_BOX+WIN_GROW_BOX);
    Shape(winID, 100, 100, 300, 400);
    viewID = NewView(winID,
        VIEW_H_SCROLL_BAR+VIEW_V_SCROLL_BAR+
        VIEW_GROW_BOX);
    portID = NewPort(viewID)
    SetPort(winID, portID);
    Show(winID);
    return winID;
}

Figure 5.9
C equivalent of the procedure in Figure 5.7

int myMenuBar;

myMenuBarID = CreateMenuBar(
    CreateMenu("File",
        "New", "Open", "Close", "Save", "Save As...", "Quit", 0),
    CreateMenu("Edit",
        "Undo", "Cut", "Copy", "Paste", "Clear", 0),
    0);

Figure 5.10
Creating a list of menus from C

Figure 5.9 shows a C procedure that does the same as the P procedure of Figure 5.7, but using the message routines.

5.2.5 Menu and dialogue routines

The C library contains procedures for creating menus and dialogues which take variable argument lists describing the required items.

A menu is created using CreateMenu, which takes a title string and list of strings containing the item names. A set of menus is assembled into a menu bar with CreateMenuBar, which takes a list of menu IDs. Figure 5.10 illustrates the use of these routines.

A dialogue box is created by CreateDialogue, which takes a list of pointers to structures describing dialogue items and returns a pointer to a dialogue structure. Other routines can be used to present the dialogue either modally or non-modally and
access the state of each of its items. Each item can also be associated with a variable in the C program. The initial setting of the control is taken from the variable, and when the dialogue is dismissed, the control’s setting is automatically copied back to the variable.

Figure 5.11 illustrates the use of the dialogue routines to create and present a dialogue for editing a title string.

5.2.6 The Notifier

Based on the facility in SunView of the same name, the notifier is the part of the Mockintosh C library concerned with handling messages received from the server. Its purpose is to hide the details of the cps input-handling mechanism and to provide a convenient way of dispatching input events to the appropriate handling routines.

Event messages handled by the notifier are made up of four parts, the event type, the gadget ID, the selectors and the parameters. The event type corresponds to the cps packet tag. The gadget ID is the ID number of the gadget which sent the message. The selectors are integers used to further subclassify the event, and the parameters contain any further relevant information.

When a message is received, the notifier locates a handler for it based on the message’s type, gadget ID and selectors. The application installs handlers using the On procedure, which is called as follows:

\[
\text{On}(\text{type}, \text{id}, \text{sel1}, \ldots, \text{seln}, \text{handler}, \text{info})
\]

where type is the event type, id is the gadget ID, sel1...seln are the selector values, handler is the procedure to be called in response to the event, and info is a pointer to any data structure that the handler needs to do its work. When a matching event is received, the handler is called as:

\[
\text{handler}(\text{id}, \text{info}, \text{params})
\]

where params is a pointer to the parameters of the event.

For example, selecting a menu item causes an event of type MenuSelection to be sent, with the ID number of the window owning the menu bar as a parameter. The event has two selectors, the menu number within the menu bar and the item number within the menu.
char titleTemp[MAX_TITLE_LENGTH];
Dialogue *editTitleDialogue;

/* This procedure would be called during initialisation
to create the editing dialogue */

MakeTitleEditDialogue() {
    editTitleDialogue = CreateDialogue(
        Label(10,10,100,16,"Title:"),
        TextEditView(10,20,300,20,0,
            titleTemp,MAX_TITLE_LENGTH),
        PushButton(10,50,40,20,"OK"),
        PushButton(250,50,40,20,"Cancel"),
        0);
}

/* This procedure would be called each time a title
was to be edited */

EditTitle(title) char *title; {
    int buttonHit;
    /* Put the title to be edited into temporary
    storage */ strcpy(titleTemp,title);
    /* Present the dialogue */
    buttonHit = ModalDialogue(editTitleDialogue);
    /* If the OK button was pressed, return the new
    value */
    if (buttonHit == 3) strcpy(title,titleTemp);
}

Use of the dialogue routines to construct and present the dialogue of Figure 5.11a

Figure 5.11a
Construction of a dialogue from standard components

Figure 5.11b
Use of the dialogue routines to construct and present the dialogue of Figure 5.11a

Suppose winInfo points to a data structure in the C program associated with the
window. Then, the following call could be used to install a handler for item saveItem of
menu fileMenu of window winID:

On(MenuSelection,winID,fileMenu,saveItem,SaveHandler,winInfo)
The procedure `SaveHandler` would be declared as:

```c
SaveHandler(id,info) int id; MyWinInfo *winInfo; {
    /* Code to handle the event */
}
```

(The `params` argument to the handler is not used in this case and has been omitted.)

Provision is made for application programs to extend the notifier to handle application-defined event types.

5.3 Conclusions

5.3.1 General impressions

Implementation of the server-side framework was relatively straightforward, although various bugs and dubious features of NeWS intruded at times. The interactive, object-oriented environment, together with the P language, proved to be a convenient way to develop user interface code.

As originally conceived, the Mockintosh was to have been usable at a basic level without the programmer having to write any server extensions, or to know much about NeWS and PostScript beyond how to use the basic graphics facilities. However, experience in writing a practical Mockintosh application (the map browser) revealed that the facilities provided are at too low a level, particularly when direct-manipulation style operations are wanted. For the sake of efficiency and general tidiness, it is usually desirable to make at least a few server extensions to support the application.

For example, the map browser required the user to be able to select a region by dragging a selection marquee over it, displayed as a dotted rectangle, and “open” it by double-clicking. Using the basic Mockintosh facilities, it would have been necessary for the application to receive the raw mouse events and take care of providing the necessary feedback itself. This would have been inefficient and likely to result in unacceptably bad responsiveness.

Instead, the approach taken was to define a new descendant of the Port class called MapPort, with methods to handle the mouse events and provide the appropriate feedback. The currently selected rectangle is kept as part of the MapPort’s state, and only when the user finally double-clicks to open the region is a message sent to the client. On the output side, methods were added to MapPort for drawing the various kinds of objects found in the map database, such as lines, circles and text blocks. To
support these extensions, the necessary cps procedures were created for accessing them, and the notifier extended to handle a new message type.

Given that the application programmer will usually be writing server extensions, the C interface to the Mockintosh is of doubtful value.

The purpose of the message routines is to provide access to the basic Mockintosh facilities without the need to know about cps; but if new classes and messages are being defined in P, the programmer will need to be familiar with cps in order to invoke them.

The dialogue routines were intended to provide a way of constructing and using dialogues easily without becoming involved in P programming. If the programmer is already writing P routines for other purposes, however, the dialogue routines do not make the process of dialogue construction much easier than writing the corresponding P code. Also, the mechanism used in the dialogue routines is not easily extensible, so that if a user-defined item is required in a dialogue, it will be necessary to use P anyway.

The notifier is perhaps the clumsiest part of the system. Its purpose is to provide an object-oriented way of handling messages from the server, by associating C procedures with particular messages and object IDs. It achieves this reasonably well for the predefined Mockintosh messages, and can be extended to handle new messages provided by the programmer. However, the extension mechanism is unwieldy, requiring the programmer to understand some of the notifier's internals, and allowing considerable scope for error. Furthermore, unlike the other parts of the C library, there is no choice about its use, since various parts of the basic system rely on it being present and handling all the input. The programmer wanting to define his own messages thus has no option but to extend the notifier.

The author now believes that the best way to solve the problems inherent in the Mockintosh C library is to discard it altogether. Most of the message routines are useful only for assembling window structures from their component gadgets using C code, a task that can be achieved just as easily, and more efficiently communication-wise, using a custom P routine called by a cps procedure. Similarly, the predefined input messages are used for reporting low-level events such as mouse clicks and keystrokes, which in most cases the programmer will want to handle in the server and communicate with the client in higher-level, application-specific terms.
Viewed in this way, the low-level communication facilities predefined in the Mockintosh at present are largely superfluous and even undesirable. They are superfluous because the most natural and efficient point at which to divide the application between the client and server results in the communication between the two being almost entirely in application-specific terms. They are undesirable because they do not significantly help, and sometimes hinder, the creation of application-specific communication facilities.

A further conclusion that the author has drawn from experience with the Mockintosh is that the facilities provided in NeWS for constructing application-specific interfaces between server and client could be much improved. As will be discussed further below, the communication model between the NeWS server and its clients is not well matched to the needs of interactive applications, and the programmer could be given much more help constructing server-client interfaces.

5.3.2 Processes

Attaching a separate lightweight process to each Gadget was done because of similar techniques used in various example programs supplied with NeWS, which seemed to provide a convenient way of dispatching events to the appropriate handling code. It was also intended to facilitate association of a graphics context with each object, via the process’s current graphics state.

However, since the methods of a Gadget are frequently called from a process belonging to another Gadget, it was found necessary in most cases for methods that draw on the Gadget’s canvas to explicitly set up the graphics state each time. As a result, the graphics state of the Gadget’s process could only be taken advantage of in rare cases.

The proliferation of processes also caused other problems. Whenever a new Gadget is created, it must "express interest" in all the event types it needs to handle. This involves creating a template object for each event type of interest, resulting in a considerable overhead when creating a complex object such as a dialogue box with many subgadgets.

The unpredictability of which process will execute a given method can lead to subtle bugs if a method makes unwarranted assumptions about the environment in which it will be executed. The order in which things happen can become difficult to control. Certain facilities which require global management of event distribution, such
as modal dialogues, become awkward to implement. Generally, things tended to become more complicated than they should be.

The Mockintosh would probably be simpler in many places if there were a single process for each client that was responsible for event collection and distribution. This process would express interest in all events for all canvases belonging to the client, and send messages to the appropriate Gadgets when events arrived for their canvases.

Such a single-process structure would greatly simplify the interactions between the various parts of the system, and completely eliminate all the process synchronisation problems. It would also allow facilities such as modal dialogues to be implemented more easily.

When a modal dialogue is active, the user must be prevented from manipulating any of the other windows belonging to the application. Effectively this means that mouse and keyboard events for other windows must be ignored. Other types of events, however, such as the “damage” events that occur when part of a window is uncovered, must still be responded to. A single event-handling process would provide a convenient point at which to carry out event filtering of this sort.

5.3.3 Graphics states

Since an object may be sent messages at any time by any process, any method which draws into its Gadget’s canvas must remember to set up the appropriate graphics state beforehand, and restore the previous one afterwards.

Because this operation is so common, class Gadget was provided with a method called Draw that takes an executable array to be executed in the Gadget’s graphics context. Draw first saves the current graphics state, sets the current canvas to the Gadget’s canvas, and then calls Setup, a method which may be overridden to set up the coordinate system in a special way if desired. Draw then executes the array passed to it, and restores the old graphics state before returning.

The Draw method makes correct management of the graphics state much easier, but it could be made easier still. An object conceptually encapsulates state and behaviour, and the graphics context of a Gadget is part of its state, so one would like to be able to simply send a Gadget a message, and have the corresponding method automatically execute in the right graphics context, just as it automatically executes in the right name scope.
This could be implemented by making a simple modification to the P class package. Instead of the send operator invoking a method directly, it would do so via a method of class Object, called, say, execmethod. Class Object would implement execmethod by looking up and executing the name of the method, just as send does ordinarily. Other classes, however, could override execmethod to provide special behaviour whenever any message was received. In particular, class Gadget could use it to set up the graphics state.

On the negative side, the graphics setup code would then be executed every time a message was received, whether the corresponding method needed it or not. Experimentation would be necessary to find out whether the extra overhead could be tolerated.

5.3.4 Communication tools

Writing two programs that communicate with each other is always harder than writing a single program, and a particularly bothersome aspect is that of designing and implementing a communication protocol. In the case of the NeWS server and its clients, the cps tool provides some aid, but it can still be tedious and error-prone. There is nothing to ensure that the parameters passed to a cps procedure match what is expected at the other end. Tag numbers for messages from the server to the client must be manually assigned, and their uniqueness and consistent use enforced by the programmer. Mechanical details like these should be taken care of by the tools.

5.3.5 The communication model

The model of communication between the NeWS server and a client seems inappropriate to the task for which it is used. In an event-driven application, the user interface is in control, receiving input events from the user and sending commands to the application.

The most logical communication model to use for this is a remote procedure call or RPC model. In the RPC model, the server provides a set of services which the client accesses by calling procedures. Each procedure is a stub which sends a message to the server identifying which procedure is to be invoked along with its parameters, and waits for a reply. The server runs in a loop, receiving requests from its clients, performing the corresponding operations and sending back the results.

Tools exist which will take a set of RPC interface definitions, comprising the headers of the required procedures along with their parameter and result types, and
generate the corresponding communication code for the server and client. Implementation of the server program is completed by filling in the bodies of the procedures, and the client program is provided with a library of procedures that may be called just like any other.

In terms of the RPC model, the event-handling process in the server runs in a loop. When it receives an event, it makes a remote procedure call to a routine in the client, which performs the appropriate action.

Remote procedure calls are also needed in the other direction. For instance, suppose the server decides that part of a window needs to be redrawn and performs a remote procedure call to a drawing routine in the client. In turn, the client routine makes remote procedure calls to the server to carry out drawing operations.

Figure 5.12 shows the flow of control and communication involved in this case. It can be seen that the system as a whole has a single thread of control which hops back and forth between server and client, and that communication between the two is symmetrical: either side may request the other to perform an operation.

The interfaces generated by cps can be thought of as remote procedure calls from the client to the NeWS server. This is not true in the other direction, however - messages sent from the server to the client must be explicitly decoded and acted upon.

Nevertheless, NeWS clients normally build an RPC-like mechanism of their own in the reverse direction, either by means of a custom-written event loop or some other mechanism like the Mockintosh notifier. If the communication model were more symmetrical, it would fit the actual usage more closely, and make the client programmer's job much easier.

This could be accomplished by a tool like cps, but in reverse - let us call it psc. A definition file would be given to psc containing the headers of C functions to be callable from PostScript. The output of psc would be a C file containing the appropriate event loop and dispatching code, and a PostScript file containing stubs for sending the relevant messages.

5.3.6 Data structures

Another facet of the communication problem is that of referencing data structures. In the Mockintosh, the client needs to be able to specify objects on the server in order to send messages to them. For this purpose, each Gadget is assigned a unique ID
number on creation, by which the client can refer to it. Two P routines are provided, HandleToID and IDToHandle, for converting between ID numbers and object handles on the server. These routines are used extensively by both the predefinedcps interfaces and those created by the application programmer.

Although this scheme works reasonably well, it places the burden of remembering to make the appropriate conversions on the programmer, and is another of the mechanical tasks of protocol implementation that could and should be handled by the tools.

With the symmetrical RPC protocol discussed above, the problem also arises in the other direction. If, in the hypothetical application illustrated in Figure 5.12, there
are many instances of Foo along with corresponding data structures in the client, a parameter is required to the RepairFoo() call to indicate which Foo the client should redraw.

To accommodate this, our proposed psc tool could be extended to allow the declaration of (at least the names of) C data structures, pointers to which should conceptually be passable across the RPC interface. Declaring a parameter or result of an RPC procedure as one of these pointers would result in the appropriate mapping operations automatically being inserted into the communication code by psc.

Figure 5.13 shows how a definition file for psc might appear, incorporating the features discussed so far. Here it is assumed that psc takes over the functions of the existing cps as well, so that both sides of the interface can be specified in a single definition.

5.3.7 Extending object-orientedness into the client

In object-oriented toolkits like Smalltalk, MacApp and InterViews, there are no artificial boundaries between the user interface and the application. The two are written in the same language, and an object-oriented style is, or can be, used throughout.
So far we have been considering the case where the client part of the application is written in ordinary C. The server code, however, communicates with the client most naturally in an object-oriented way. If the client were written in an object-oriented language, interfacing between server and client could be made considerably tidier.

Figure 5.14 shows how an object-oriented interface between C++ and P might be specified. The interface could be implemented by means of "proxy" objects representing actual objects on the other side of the interface.

For each C++ object to which the server can send a message, a "proxy" P object would also be created on the server, to provide something for the server routines to send messages to. Each method of the C++ object would have a corresponding proxy method defined for the P object.

The proxy method would send a message over the network identifying the target object, the method and any parameters. Interface code in the client (generated by psc) would call the appropriate method of the C++ object. Similar arrangements would apply in the other direction.

5.3.8 Summary

In hindsight, the application program interface of the Mockintosh could be made simpler by encouraging the programmer to make application-specific extensions in P and construct a custom communication protocol suited to the task at hand. The internal
structure of the Mockintosh would be simplified if a single process for each client were responsible for collecting input events and distributing them to user interface objects.

Management of the graphics states of each user interface object would be easier if messages sent to an object automatically executed in the correct graphics context. This could be easily arranged by means of a simple modification to the mechanism used in NeWS for implementing method calls.

The communication model between the NeWS server and its clients would be better suited for its intended task if it were made more symmetrical, so that a remote procedure call metaphor could be used in both directions. Proposals have been made for extensions to the tools provided in NeWS for constructing client-server interfaces, to support this improved communication model, and automate some mechanical tasks that are currently left to the programmer.

When the client is written in an object-oriented language, objects in the client should be able to send messages to objects in the server and vice versa. Further extensions to the communication tools have been proposed to facilitate the specification of this kind of communication protocol and the automation of its implementation.
Chapter 6

Style Independence

Consistency between the user interfaces of different applications makes an important contribution to the ease of use of a computer system. If the user can invoke the same operations in different applications in the same way, then there is less for the user to remember, and knowledge gained from using one application can be applied to others. This observation applies to user interfaces of any sort, whether graphical or not, but has been brought into sharp focus in recent times by the success of systems such as the Apple Macintosh where consistency is a primary goal of the user interface design.

Consistency is, however, difficult to maintain when using hardware and software obtained from diverse sources. This chapter examines the issue of user interface consistency from a global perspective, pointing out ways in which the current generation of user interface systems support, or fail to support, consistency between user interfaces of different applications.

In this chapter, the author proposes that user interface systems be designed with the goal of style independence: the user should be able to choose a user interface style according to personal preferences, and have all the applications he or she uses automatically conform to that style. Some criteria that a style-independent system needs to meet are then discussed, and some suggestions made for further work in this area.

6.1 User interface styles

The style, or "look and feel", of a user interface is a set of conventions concerning the appearance and behaviour of certain interface elements that are consistent across applications. Some styles in common use are the Macintosh style, the "MIT style" of X Windows, and proposed standards such as OpenLook (Probst 1988, Sun 1990) and Motif (Deininger and Fernandez 1990).

To maintain consistency, it is important that all applications used by a given user have the same style of user interface. Frequently-performed actions become
Experiments in Interactive Map Retrieval

6.2 Style independence

Current tools for constructing applications with graphical user interfaces result in a situation where the user interface style comes with the application, as shown in Figure 6.1. Although some customisation of the user interface is often allowed for, the basic style of user interface is determined when the application is chosen.

Given the abundance of user interface styles, this means that either the user’s choice of applications is restricted, or the user must put up with using applications with differing styles of interface, often side by side at the same time - a very undesirable arrangement.

The problem could be solved by standardising on a single user interface style to be used by all application developers. But reaching universal agreement on anything is difficult, and reaching agreement on something as complex, subtle and subjective as a user interface style seems unlikely in the foreseeable future. In any case, given the amount of personal preference involved, standardising on a single style is probably undesirable.

A much better solution would be if the user interface style came with the user, as shown in Figure 6.2. In this scheme, a standard application program interface, or API, is used by all applications to communicate with a user interface library. The user automatic, and adjustment to a different style in which things are accomplished in subtly different ways can be difficult.

Figure 6.1
Current scheme: User interface comes with the application program
selects a library implementing the style of the user’s choice, and through it may communicate with any application in that style.

For this scheme to work, the API must be style independent. That is, it must abstract all the details that differ between the various user interface styles that the system is to cope with.

The problem has therefore been shifted from standardising on a user interface style to standardising on a style-independent API. Before one can be standardised on, though, at least one style-independent API must be designed.

6.3 Previous work on style independence

Little seems to have been published which directly addresses the issue of style independence, particularly with regard to direct-manipulation interfaces.

Feldman and Rogers (1982) describe a style-independent system for sequential-dialogue interfaces. An abstract interaction handler (AIH) communicates with applications in an intermediate language consisting of abstract interaction tasks connected by a state transition network. A set of style modules implement the interaction tasks in terms of particular interaction techniques.

Paddon and Spaziani (1989) address the problem of creating applications that can be used from a variety of kinds of devices ranging from bitmapped graphics displays down to simple text terminals. Restricting themselves to textually-oriented
Experiments in Interactive Map Retrieval

applications, they describe an adaptable text interface (ATI) based on a set of abstract object types which permit varying degrees of elaboration in their implementation depending on the capabilities of the device in use.

An interesting feature of Paddon and Spaziani's system is their handling of layout. The size of items within a window is dependent on the device being used, leading to the possibility of items overlapping. To solve this, the user is allowed to rearrange items interactively at run time, and to save the resulting layout for later use.

Goldschlager and Coldwell discuss an alternative way of structuring user interface systems and their applications, which is document-centred rather than application-centred. Instead of being "owned" by a particular application, documents exist independently and are operated on by many small, specialised applications that can be fitted together much like Unix tools.

Goldschlager and Coldwell's documents consist of a hierarchy of data objects that are displayed in a corresponding hierarchy of windows. Associated with each "active region" of a window there is an application which handles user interaction with the data object displayed in that region. Applications communicate by passing messages up and down the window hierarchy.

User interface objects such as scroll bars, menus etc. are implemented by small applications that interact with the user and send appropriate messages to other applications. All interaction with the user is isolated in the window manager and these user interface applications, so that in principle the user interface style could be changed by modifying these parts of the system.

6.4 Style dependencies in current user interface toolkits

Most user interface toolkits in common use are designed in such a way that the dialogue parts of the application need detailed knowledge about the structure and layout of the user interface, something which is often very style-dependent.

To illustrate this point, we examine the implementation of a simple dialogue box using two different styles of interaction with some typical user interface toolkits. The dialogue is one that might be found in a map browsing system for selecting map layers to be viewed.

The dialogue box will display a list of layer names together with an indication of whether each layer is currently selected, and provide a set of operations for selecting and deselecting layers, and rearranging the order in which they will be overlaid.
Figure 6.3a illustrates the dialogue as it might appear in a system like Smalltalk or SunView where commands are typically issued by means of a pop-up menu. Figure 6.3b shows an alternative style more likely to be found on the Macintosh where the commands are invoked by means of a set of buttons.

Figure 6.4a shows part of the Smalltalk implementation of the interface of Figure 6.3a. An application-defined class LayerList models the list of layers as a collection of strings and provides a set of methods for operating on it. When sent the message browse, the LayerList creates an instance of the standard class SelectionInListView,
Experiments in Interactive Map Retrieval

Figure 6.4a
Smalltalk objects implementing the interface of Figure 6.3a

![Diagram](image)

Figure 6.4b
Alternative Smalltalk interface using buttons and corresponding user interface objects

parameterised with the names of methods for accessing the list and obtaining a menu of commands. The LayerList also creates an instance of StandardSystemView (which implements the top-level window frame and title) and places the list view inside it.

Figure 6.4a shows the relationships between the various objects. Figure 6.5a shows the source code of two methods of LayerList: the *browse* method which creates the user interface objects, and the *menu* method which creates the menu that the SelectionInListView will pop up.

Figures 6.4b and 6.5b show an alternative implementation in Smalltalk using an interface more like that of Figure 6.3b. Comparing these two implementations, it can be seen that the dialogue-related code is heavily dependent on the interaction techniques that are to be used (pop-up menu or set of buttons) and the layout of the relevant user interface objects.
Chapter 6 – Style Independence

!LayerList methodsFor: 'browsing'!

browse
  "Open a browser on this layer list"
  | topView listView |

  "Create the top level view"
  topView ← StandardSystemView new label: 'Layers'.
  topView borderWidth: 1.
  topView minimumSize: 200@100.

  "Create the list view"
  listView ← SelectionInListView on: self
    printItems: true onElement: false
    aspect: #selection change: #selection:
    list: #list menu: #menu initialSelection: #initialSelection.

  "Plug them together"
  topView addSubView: listView
    in: (O@0 extent: 1@1) borderWidth: 1.

  "Show them to the user"
  topView controller open


Figure 6.5a

Part of the Smalltalk code for creating and assembling the user interface objects

This kind of dependency between the user interface style and the application code is not unique to Smalltalk, but is typical of object-oriented user interface toolkits. Figure 6.6 shows implementations of these two interfaces using SunView, and Figure 6.7 using InterViews.

In all of these, the programmer is responsible for selecting the interaction techniques to be used and writing code to create the necessary dialogue objects and assemble them into the desired layout. Clearly, any substantial change to the user interface style will require many pieces of application code to be searched for and changed.
"Open a Style 2 browser on this layer list (button controlled)"

| topView listView numButtons buttonPos buttonSize button buttonView buttonNames buttonActions buttonWidth buttonHeight |

"Create the top level view"

```
topView ← StandardSystemView new label: 'Layers'.
topView borderWidth: 1.
topView minimumSize: 200@150.
```

"Create the list view"

```
listView ← SelectionInListView on: self
   printItems: true oneltem: false
   aspect: #selection change: #selection:
   list: #Iist menu: nil initialSelection: #initialSelection.
topView addSubView: listView
   in: (0@0 extent: 0.75@1) borderWidth: 1.
```

"Create the buttons"

```
buttonNames ← #("Show" 'Hide' 'Top' 'Raise' 'Lower' 'Bottom' 'OK' 'Cancel').
buttonActions ←
   #(showSelection hideSelection selectionToTop
      raiseSelection lowerSelection selectionToBottom
      acceptEdits cancelEdits).
numButtons ← buttonNames size.
buttonWidth ← 0.25.
buttonHeight ← (1/numButtons).
buttonSize ← buttonWidth@buttonHeight.
buttonPos ← 0.75@0.
```

```
1 to: numButtons do: [:i |
   button ← Button newOff.
   button onAction: [
      ((i > 6) | (selection ~= nil))
      ifTrue: [self perform: (buttonActions at: i)].
      (i >= 7) ifTrue: [topView controller close]
   ].
   buttonView ← SwitchView new model:button.
   buttonView label: ((buttonNames at: i) asDisplayText form).
   topView addSubView: buttonView
   in: (buttonPos extent: buttonSize) borderWidth: 1.
   buttonPos ← buttonPos + (0@buttonHeight).
   ]
```

"Show the lot to the user"

```
topView controller open
```

---

**Figure 6.5b**

Smalltalk code for creating the interface of Figure 6.5a
6.4.1 Input

In these examples, most of the style dependencies arise because the application needs to know the particular ways that the user will enter commands.

In direct-manipulation interfaces, most input actions consist of selecting objects and specifying commands to be carried out on those objects. Often the commands are themselves represented by objects, such as buttons, menu items, and tool icons, which the user selects. In fact, Phillips and Apperley (1990) surveyed user interaction tasks in several Macintosh applications and concluded that all the interactions they studied could be decomposed into selection sub-tasks.

In current object-oriented toolkits, commands tend to be conveyed in terms of which user interface objects were selected, such as “item n of menu m chosen” or “button b pressed”. A means of associating command objects with semantic procedures is often provided, but it is up to the application to decide the mapping between commands and the specific user interface objects which represent them.
def int (*actions[]()) {
    show_layer, hide_layer, 0,
    layer_to_top, raise_layer, lower_layer, layer_to_bottom, 0,
    accept_edits, cancel_edits
};

layer_list_event_proc(item, event)
    Panel_item item; Event *event;
    if (event_action(event) == MS_RIGHT && event_is_down(event)) {
        int command = (int) menu_show(menu, panel, event, 0);
        (*actions[command - 1])();
    }
}

main() {
    frame = window_create(NULL, FRAME,
        FRAME_LABEL, "Layers",
        0);

    panel = window_create(frame, PANEL,
        WIN_ROWS, 20,
        WIN_COLUMNS, 40,
        0);

    layer_list = panel_create_item(panel, PANEL_CHOICE,
        PANEL_FEEDBACK, PANEL_INVERTED,
        PANEL_LAYOUT, PANEL_VERTICAL,
        0);

    menu = menu_create(MENU_STRINGS,
        "Show", "Hide",
        "------",
        "Top", "Raise", "Lower", "Bottom",
        "-------",
        "Accept", "Cancel",
        0, 0);

    panel_set(layer_list,
        PANEL_EVENT_PROC, layer_list_event_proc,
        0);

    window_fit(frame);

    window_main_loop(frame);
}

Figure 6.6b
SunView code implementing the menu-style dialogue

The choice of user interface objects to represent commands can depend heavily on the user interface style. In some styles, there is a certain distinguished set of commands for which very strict guidelines are laid down concerning the interaction techniques for invoking them. For instance, Figure 6.8 shows the standard File and Edit menus as specified in the Macintosh User Interface Guidelines (Apple 1985). All
Chapter 6 – Style Independence

Figure 6.6c
SunView code implementing the button-style dialogue

applications which use these commands are expected to have menus which are as similar to these as possible.

Other user interface styles often have similar sets of distinguished commands, but invoked in different ways. For example, Smalltalk has a very similar set of standard text-editing commands, invoked from a pop-up menu over the window concerned. SunView uses a set of specially-labelled function keys, and some X styles use various buttons of a multi-button mouse.

Another area of differing input conventions concerns ways of selecting objects to be operated upon. The use of a mouse click to select an individual object is fairly universal. Methods of extending the selection variously involve using different mouse buttons, mouse buttons in conjunction with modifier keys, or mouse gestures such as dragging. These kinds of input actions tend to be conveyed to the application in quite
low-level terms, such as "right mouse button down with shift key", leaving the application to work out their (style-dependent) meaning.

6.4.2 Output

Style dependencies can also arise on the output side, often in subtle ways. In the example of Figure 6.3a, '+' and '-' characters have been arbitrarily used to mark shown and hidden layers, in the absence of any particular convention in Smalltalk for such purposes. In contrast, Macintosh applications often use a check mark as shown in Figure 6.3b.

6.4.3 Scrolling and multiple views

Scrolling is a very common feature of graphical user interfaces, and almost every user interface style has a convention regarding the apparatus used for it. The
/* Derive a class from MenuItem to call a procedure when the item is chosen */

class Command : public MenuItem {
    public:
    Command(char *name, void (*procedure)());
    /* ... */
};

/* Derive a class from StringBrowser to hold the list of layers and manage a pop-up menu */

class LayerList : public StringBrowser {
    public:
    LayerList(Menu *);
    /* ... */
};

/* Construct and present the dialogue */

CreateDialogue() {
    /* Create the menu */
    Menu *menu = new PopupMenu();
    menu->Include(new Command("Show", ShowLayer));
    menu->Include(new Command("Hide", HideLayer));
    menu->Include(new MenuItem(new HBorder()));
    menu->Include(new Command("Top", LayerToTop));
    menu->Include(new Command("Raise", RaiseLayer));
    menu->Include(new Command("Lower", LowerLayer));
    menu->Include(new Command("Bottom", LayerToBottom));
    menu->Include(new MenuItem(new HBorder()));
    menu->Include(new Command("Accept", AcceptEdits));
    menu->Include(new Command("Cancel", CancelEdits));

    /* Create the layer list */
    list = new LayerList(menu);

    /* Frame it and show it to the user */
    world->Insert(new MarginFrame(new Frame(list), 8));
}

Figure 6.7b
InterViews code for constructing the menu-style dialogue

appearance and functioning of this apparatus varies considerably between one style and another.

Being so common, almost every user interface toolkit provides support for scrolling in some form. In some toolkits, the scrolling apparatus is almost completely invisible to the application; in MacApp, for example, a TFrame is created with options specifying scrollability, and thereafter the creation, assembly and management of scroll bars is handled entirely by the toolkit.
Experiments in Interactive Map Retrieval

/* Derive a class from PushButton to call a
   procedure when the button is pressed */

class CmdButton : public PushButton {
  public:
    CmdButton(char *name, void (*procedure)());
  /* ... */
}

/* Derive a class from StringBrowser to hold the
   list of layers */

class LayerList : public StringBrowser {
  public:
    LayerList();
  /* ... */
};

/* Construct and present the dialogue */

CreateDialogue() {
  /* Create the buttons */
  Box *buttons = new VBox();
  buttons->Insert(new CmdButton("Show", ShowLayer));
  buttons->Insert(new VGlue(gap));
  buttons->Insert(new CmdButton("Hide", HideLayer));
  buttons->Insert(new VGlue(2*gap));
  buttons->Insert(new CmdButton("Top", LayerToTop));
  buttons->Insert(new VGlue(gap));
  buttons->Insert(new CmdButton("Raise", RaiseLayer));
  buttons->Insert(new VGlue(gap));
  buttons->Insert(new CmdButton("Lower", LowerLayer));
  buttons->Insert(new VGlue(gap));
  buttons->Insert(new CmdButton("Bottom", LayerToBottom));
  buttons->Insert(new VGlue(2*gap));
  buttons->Insert(new CmdButton("Accept", AcceptEdits));
  buttons->Insert(new VGlue(gap));
  buttons->Insert(new CmdButton("Cancel", CancelEdits));

  /* Create the layer list */
  list = new LayerList();

  /* Assemble and show to user */
  Box *box = new HBox
    (new Frame(new MarginFrame(list, 4)),
    new HGlue(gap),
    buttons);
  world->Insert(new Frame(new MarginFrame(box, 8)));
automatically). Here, style-dependent information, such as the placement of scroll bars, becomes embedded in the application code.

A related facility often provided is the ability to create multiple views of the same or different parts of a document. Conventions differ on how this is done - splitting panes, creating new windows - and the interaction techniques used for creating extra views.

Toolkits often provide at least some support for multiple views. In Smalltalk, a Model may have multiple View-Controller pairs attached to it, with all Views notified automatically whenever the Model changes. Similarly, in InterViews a Subject can have multiple Views, and in MacApp a TDocument can have multiple TViews, each of which appears in a different TFrame. However, in all these cases, style-dependent code is required in the application to provide the user with a command for creating a new view, and presenting the view in an appropriate way - such as by splitting an existing view, or creating a new window.

6.4.4 Summary

In summary, existing object-oriented user interface toolkits for model-world user interfaces are designed in such a way that applications which use them inevitably contain much code that is heavily dependent on a particular user interface style. These style dependencies arise either because style-dependent features are designed into the toolkit and reflected in the facilities provided to the programmer, or because of

<table>
<thead>
<tr>
<th>File</th>
<th>Edit</th>
</tr>
</thead>
<tbody>
<tr>
<td>New %N</td>
<td>Undo %Z</td>
</tr>
<tr>
<td>Open... %O</td>
<td></td>
</tr>
<tr>
<td>Close</td>
<td>Cut %H</td>
</tr>
<tr>
<td>Save %S</td>
<td>Copy %C</td>
</tr>
<tr>
<td>Save As...</td>
<td>Paste %U</td>
</tr>
<tr>
<td>Page Setup...</td>
<td>Clear</td>
</tr>
<tr>
<td>Print</td>
<td></td>
</tr>
<tr>
<td>Quit %Q</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.8
Standard Macintosh "File" and "Edit" menus
style-dependent choices that the programmer is required to make when writing the application.

6.5 Existing user interface customisation methods

Many user interface systems provide a means for the user to customise the user interface style to some degree. Usually this takes the form of a configuration file which specifies values for a set of options. The extent to which the user interface style can be modified in this way depends on the range and flexibility of the options provided.

6.5.1 SunView defaults file

In the SunView system, a per-user defaults file is used to hold a collection of option settings relating to both individual applications and the system as a whole. Some of the system-wide settings that may be customised are:

- Fonts to be used for various purposes
- Attributes of pop-up menus, such as margin width, and presence or absence of a shadow
- Size and placement of scroll bars
- Background pattern
- Mappings between function keys and standard editing and window management commands

The extent of customisability provided by these sorts of options is very restricted. For instance, a SunView user can change which side of a window pane its scroll bar appears on, but only because the designers of SunView thought to include it as an option. It is impossible to change the appearance and functioning of scroll bars sufficiently to make them work like, for instance, Macintosh scroll bars.

6.5.2 Macintosh resource files

In the Macintosh system, a file can contain a collection of resources. A resource is a block of data with an associated type, represented by a 4-character name. An application contains resources of type CODE which hold executable code, together with other resources used by the application. Many of the visible features of the application's user interface are controlled by these resources, and there exist tools for
editing resources which allow these features to be customised without changing any application code. For example:

- All of the character strings used to label windows, menu items, buttons and so forth are (in a well-written application) stored in resources, so that they can be translated into another natural language.

- Resources of type MENU and DLOG control the layout of menus and dialogue boxes.

- Special types of code resource control the appearance and behaviour of certain standard user interface components. WDEF resources define the appearance of window frames and the way windows are moved and resized. MDEF resources define the appearance and behaviour of menus. CDEF resources define the appearance and behaviour of controls such as buttons and scroll bars.

There is a system resource file shared by all applications containing WDEF, MDEF and CDEF resources for the standard kinds of windows, menus and controls found in the Macintosh user interface. By replacing these resources, there is considerable scope for making system-wide changes to the user interface style. Since these resources contain executable code, there are few limits to the changes that could be made to the appearance or behaviour of the components they define.

However, there are still limits to the kinds of changes that can be made to the user interface as a whole. While all scroll bars could be made to look and behave like SunView ones by writing an appropriate CDEF function and installing it in the system resource file, there is no way that they could be made to appear on the left side of a window instead of the right on a system-wide basis, because the code for positioning scroll bars is embedded in each application.

6.5.3 X resources

The X Windows system uses an arrangement of option settings which is similar in some ways to that of SunView. In X these option settings are referred to as "resources"\(^1\).

X resources are structured around the hierarchy of objects making up the user interface of an application, or widget hierarchy (in Xt-based toolkits, a user interface

\(^1\) The X usage of the term “resource” has nothing to do with the Macintosh usage.
object is termed a "widget"). Each widget has a name and a set of options that the user can specify. An X resource consists of a sequence of widget names tracing a path down the widget hierarchy to the widget concerned, the name of an option, and a value for the option.

The set of options which may be modified in an X application is typically much larger than in SunView. Most widgets allow their size, colour, font and position within their parent widget to be specified, plus many other options dependent on the type of widget. There are also options which control the bindings between input events and actions that the widget can perform. These kinds of options provide quite a wide scope for customising an application's user interface.

System-wide changes can often be made, at least among applications which use the same user interface toolkit. Each type of widget has a class name, and wildcards may be used in widget pathnames, allowing options to be specified that apply to any occurrence of a given type of widget in any application.

Although the X resources system is very flexible, there are still limits to the kinds of customisations that can be made. A widget can be positioned anywhere within its parent but cannot be moved to a different parent or changed into a different kind of widget. A command invoked by a button could not be moved into a menu, for instance. Essentially, no structural changes can be made to the user interface, only cosmetic ones.

6.5.4 NeWS

Of all the user interface systems studied here, NeWS appears to offer the greatest promise in terms of user interface style customisation. The NeWS server provides a place to install a user interface toolkit which is shared by all applications and provides unlimited scope for customisation, even incrementally and dynamically.

However, as we have seen, it is not sufficient simply to be able to change or even totally replace the user interface toolkit. Unless the toolkit's API is very carefully designed, style dependencies become embedded in the applications themselves, severely limiting the changes which can be effected by modifying the toolkit.

So, although NeWS as it stands does not solve the problem of providing a style-independent user interface system, the NeWS environment promises to be an excellent one in which to carry out further research in this area.
6.6 An architecture for a style-independent user interface system

At first sight, the concepts of style independence and dialogue independence appear to be very similar. Both have the goal of separating an application into two parts, with a well-defined interface between the two, in such a way that each part may be substantially changed without affecting the other.

The two concepts are, however, quite distinct. Dialogue independence is concerned with finding, for each application, a division of that application into a dialogue component and a computational component, such that the dialogue component encompasses the implementation of the whole user interface of that application.

The dialogue and computational components communicate through what might be called the dialogue-computation interface, or DCI, as shown in Figure 6.9. The DCI is a private, internal interface uniquely designed for that particular application. The nature of the DCI depends on the kinds of computation performed by the application, and on whatever methodologies and tools are used to construct the dialogue component. Since the dialogue and computational components are designed to go with each other and nothing else, there is no need for the DCIs of different applications to be compatible, or even bear any resemblance to each other.

Within the dialogue component, there will be some parts which handle interaction goals which are “standard” in the sense that the user interface style sets down guidelines for the interaction techniques that are to be used for them, to ensure consistency between applications. This part of the dialogue component will be termed the standard dialogue component.

Other parts of the dialogue component will be concerned with interaction goals for which there is no guideline provided by the style, so that the dialogue designer is free to choose or invent an interaction technique for use in that particular circumstance. This part of the dialogue component will be termed the custom dialogue component.
As we have seen, current toolkits for constructing the dialogue components of applications were designed with only one user interface style in mind. As a result, the standard and custom dialogue components of applications constructed with these toolkits are intertwined, as depicted in Figure 6.10.

If the aim of a style-independent environment for applications is to be achieved, it will be necessary to separate the standard and custom dialogue components. Figure 6.11 shows a system architecture in which all the standard dialogue components have been factored out and placed in a single module, shared by all applications. This module isolates all the style-dependent parts in one place, and can therefore be called the style module. By replacing this module, the user interface style is changed immediately and consistently for all applications which use it.

Between the style module and the application there is a new interface: the style-application interface, or SAI. The SAI is both style-independent and application-independent; it is style-independent because it connects many styles with each application, and it is application-independent because it connects many applications with each style.

The key difference between style independence and dialogue independence can now be seen. Dialogue independence is concerned with defining an internal interface within each application, and deals with all of the application's dialogue requirements. Style independence is concerned with defining an external interface used by all applications, and deals with only a part of the dialogue: that which must be consistent across all applications.

### 6.7 Further benefits of a style-independent architecture

The system architecture of Figure 6.11, if it could be realised, would confer several extra and far-reaching benefits beyond the ability to simply change user interface styles.
6.7.1 Programming convenience

By isolating all of the style-dependent parts of the dialogues of all applications, and implementing them once for each style, the dialogue developer is both aided in ensuring consistency between applications, and spared the need to re-implement these in each application.

6.7.2 Application portability

A common lament is that many of the powerful, interactive applications now available are tied to a particular brand of hardware; Macintosh applications, for example, are intimately connected with the proprietary Macintosh system software, and can be adapted to other systems only with great difficulty.

There are two problems involved in porting such applications to a new environment. One is simply getting it to work, and the other is the fact that elements of the user interface style of the old environment will often be buried deeply in the application, and may be very difficult to dig out and change.

An API that was both style and application independent would solve both of these problems, resulting in applications that could be moved easily to any environment, and used with any user interface style once there.
6.7.3 User interface portability

Figure 6.11 depicts the SAI as having two levels at which the application can interact. The high-level part interfaces the application to the style module, and consists of abstractions of the various standard interaction goals for which the style module supplies interaction techniques.

For the custom dialogue component of the application, a lower-level interface will be necessary, across which will pass uninterpreted input events and simple output primitives. The application can then build its own interaction techniques on this interface, for interaction goals which the high-level interface does not support.

Considering that the style module will itself have need of low-level input and output, it is logical to consider whether it can be made to use the same low-level interface as is provided to the application through the SAI, as shown in Figure 6.12. Since the SAI, by definition, is standardised, this has the immediate and very considerable benefit that all of the style modules are now portable.

The only hardware-dependent code in the system is that between the input and display hardware and the low-level SAI. By implementing this small part on a new piece of hardware, all the applications and style modules immediately become usable.

6.8 Designing a style-independent SAI

An ideal style-independent SAI would need to be flexible enough to cope with any set of guidelines for any user interface style existing now or in the future. Since the range of possible user interface styles is unbounded, this is most likely impossible.
A more reasonable goal would be to take the interface styles most widely used at present, and try to find a set of abstractions which can accommodate them all. These abstractions should be made as general as possible, to allow room for user interface styles to evolve.

A further desirable goal is that the SAI should be as simple as possible. Current user interface toolkits tend to be very complicated, with documentation often running to hundreds of pages, and involving many interrelated concepts which must be understood before the toolkit can be used effectively. Keeping the abstractions of the SAI as few and as easily described as possible would seem to be an important goal.

It is beyond the scope of this thesis to develop a full specification for a style-independent API. The remainder of this section will briefly discuss the main features that such an API might have and some of the problems that will be encountered, and then suggest how they might be tackled.

6.8.1 Basic abstractions

With current object-oriented toolkits, the programmer defines a set of application-dependent objects together with a set of operations that may be performed upon them. Then, a network of user interface objects is constructed to provide visible representations of these objects and the means for the user to invoke operations on them.

As we have seen, the structure of the user interface network is heavily dependent on the user interface style. So, the first requirement of our style-independent SAI will be to completely hide its structure from the application programmer.

Let us define a visible object, or VObject, as the basic abstraction of the SAI. Every object belonging to the application that is to have a visible representation will be implemented by means of a VObject.

Each VObject will have a set of operations associated with it. In particular, every VObject will have a Draw operation which is invoked by the user interface system whenever the object needs to be drawn on the screen. Other operations will represent commands that may be entered by the user.

There will be two kinds of VObject, standard and application-defined. Standard VObjects will be used to represent objects that are found in many applications, and which various user interface styles define specific ways of displaying and interacting
Experiments in Interactive Map Retrieval

class Layer : public VObject {

    interface:
    void Draw(); /* Determines its appearance */
    void Show(); /* Operations the user may invoke */
    void Hide();

    private:
    boolean shown;
    char *name;
    /* any other private data and methods */
}

Figure 6.13
C++-like specification of an application-defined VObject

with. For objects that are unique to an individual application, a generic VObject will be provided that the application programmer can adapt.

There will also be standard and application-defined operations on VObjects. Again, standard operations will be ones that are found in many applications, and the current style will determine the interaction techniques used to invoke them.

6.8.2 Specifying VObjects

To keep the application programmer's task simple, it is desirable that the description of the application's interface with the system be simple and brief, and as compatible as possible with the language in which the application is being written.

Since a VObject is conceptually just an application object which happens to be visible to the user, a possible technique would be to use the language's own syntax for defining data types as a basis for the specification language. Then, the programmer would need only to write a single definition for a given object that would be usable by both the user interface system and the application program itself.

Figure 6.13 shows an example of how such a specification might appear using a C++-like specification language. It is identical to a normal C++ class declaration, with the addition of an interface section containing methods which represent actions that the user or user interface system may invoke. A preprocessor could be used to derive a standard C++ header file from this specification.

An important characteristic of this description is that it specifies only the operations which may be performed on the object (and, in the case of an application-defined VObject, its appearance). The specification does not in any way restrict how the user is to invoke these operations. The user interface system is responsible for
selecting user interface objects to represent these commands, according to the style being used, and arranging them appropriately.

The user interface system is also responsible for deciding where and how the VObject itself is to be shown to the user, how scrolling is to be performed, and how multiple views are created and arranged.

6.8.3 Standard VObjects

Certain kinds of primitive data objects frequently need to be shown to the user, and often the user must be allowed to enter or edit their values. Objects of this sort include character strings, numbers, booleans and enumerated types.

User interface styles usually have standard interface objects for representing these kinds of data items. For instance, the Macintosh has text boxes for editing strings and numbers, check boxes for booleans, and radio button groups or pop-up menus for enumerated types.

Standard VObjects such as VString, VInteger, VBoolean, etc., could be provided for these data types, and the user interface system could easily infer the appropriate interface object to use according to the current style.

As well as primitive objects, certain kinds of composite objects are also frequently found. Dialogue boxes, for instance, often consist mostly of a collection of standard primitive interface objects, and are used to allow the user to edit a collection of attributes.

Figure 6.14 shows how a dialogue might be specified in terms of a C++ structure of data items that the user is to be able to edit. The intention is for the user interface system to infer that a dialogue box is appropriate and construct one from a collection of user interface objects corresponding to the fields of the structure. The specification does not define the layout of the items within the dialogue box, since this is likely to be influenced by the style being used.

Other kinds of composite items for which VObjects could be defined include one and two dimensional arrays (such as a list of file names, or array of spreadsheet cells) and collections of objects that may be positioned arbitrarily (such as file icons in the Macintosh Finder). The corresponding VObjects would define operations such as selecting, moving, inserting and deleting items, which would be implemented according to the current style.
struct LineStyle : VObject {
    interface:
    VInteger lineWidth;
    VBoolean doubleLine;
}

Figure 6.14
A C++ structure from which a dialogue box could be inferred

6.8.4 Standard commands

One of the main problems in designing a style-independent API concerns the selection of appropriate means for the user to invoke commands. As mentioned earlier, some user interface styles set down strict guidelines concerning a certain set of commands. These commands and their semantics will need to be defined as a part of the API, so that they can be recognised and handled correctly.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Macintosh</th>
<th>SunView (textedit)</th>
<th>OpenLook (Sun textedit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open file</td>
<td>Items under &quot;File&quot; pulldown menu</td>
<td>Items in popup menu over window pane</td>
<td>Items under &quot;File&quot; pulldown menu</td>
</tr>
<tr>
<td>Close file</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Save under existing name</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Save under new name</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quit application</td>
<td></td>
<td>Item in window frame popup menu</td>
<td></td>
</tr>
<tr>
<td>Cut</td>
<td>Items under &quot;Edit&quot; pulldown menu</td>
<td>Function keys</td>
<td>Items under &quot;Edit&quot; pulldown menu</td>
</tr>
<tr>
<td>Copy</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Paste</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clear</td>
<td></td>
<td>No equivalent</td>
<td></td>
</tr>
<tr>
<td>Select</td>
<td>Click</td>
<td>Left-click</td>
<td></td>
</tr>
<tr>
<td>Select range</td>
<td>Click and drag</td>
<td>No equivalent</td>
<td></td>
</tr>
<tr>
<td>Deselect</td>
<td>Click in empty space</td>
<td>Click in empty space</td>
<td></td>
</tr>
<tr>
<td>Extend selection</td>
<td>Shift-click</td>
<td>Middle-click</td>
<td></td>
</tr>
</tbody>
</table>

Anticipation of every possible command that may need special treatment in every possible user interface style is obviously impossible, so it will be necessary to choose some set of styles to use as a working basis. Fortunately, the distinguished commands found in most of the commonly used user interface styles at present are reasonably similar. Table 6.1 lists some of these commands and their implementation in a few styles.
Some pairs of user interface styles have commands which are similar but whose semantics do not exactly match; in these cases it may be necessary to break some operations down into more primitive ones. For example, the Macintosh "Paste" command deletes any existing selection before inserting the contents of the clipboard, whereas the SunView "Paste" command does not\(^2\). In this case the Macintosh "Paste" operation could be resolved into "delete selection" followed by "insert contents of clipboard".

6.8.5 Application-defined commands

A more difficult problem arises with application-specific commands. The style being used will have some influence over the way these commands are implemented. For example, in Smalltalk, pop-up menus are used heavily for entering commands of many sorts, whereas in the Macintosh they are used much less often and for more restricted purposes.

A command that would be found in a pop-up menu in Smalltalk would, in a Macintosh application, be more likely to be found in a pull-down menu in the menu bar, or invoked by pressing a button. But the choice between a menu command or a button, or some other technique, remains open. The best choice in any given case will depend on factors such as how frequently the command is used, how similar commands are handled in other applications, and the personal preferences of the user.

So we have the problem that the mapping between commands and interaction techniques cannot be built into the application, and yet also cannot be completely resolved simply by knowing the user interface style. Its full resolution will depend on many aesthetic and ergonomic considerations.

The problem could be addressed by associating a configuration file with the application, describing the structure of a suitably designed user interface for it in a particular style.

But this would require a configuration file per application per user interface style. Creating these configuration files would place a considerable burden on either the application developer, who would need to supply a set of configuration files with each

\(^2\)Although SunView provides another kind of selection, called a "pending delete" selection, which behaves like a Macintosh selection with respect to the "Paste" command.
application, or the user, who would need to construct a configuration file for each application acquired.

Furthermore, changing to an entirely new and unanticipated style would result in none of the user's applications working until new configuration files had been obtained or constructed for them.

6.8.6 Automatic interface generation

A better approach would be to automatically generate a user interface based on the set of operations associated with each VObject. In this respect the system would be somewhat akin to MIKE (Olsen 1986) which generates a menu-based interface from a description of an application's semantic procedures, or the Control-Panel Interface (Fisher and Joy 1987) which generates dialogue boxes.

A more general system than these would be required, based on heuristics that capture some of the content of the guidelines for the user interface style being used. The heuristics could be guided by programmer-supplied "hints" concerning things such as the relative frequency with which commands are used.

The automatically-generated interface might not be ideal, but it would serve as a starting point. Then, a means could be provided for the user to tailor the interface to overcome any shortcomings and satisfy personal preferences. The tailoring mechanism could take the form of a configuration file, or better yet, the interface could be made interactively editable in a direct-manipulation fashion, like Hypercard (Goodman 1987) or HYPE (Read and Smith 1991).

6.9 Summary

Style independence in user interface systems has the advantages of promoting consistency between applications from diverse sources and enhancing portability of applications and user interfaces. However, the design of current object-oriented user interface toolkits is such that style-independent applications are difficult or impossible to create.

To allow the development of style-independent applications, it will be necessary to define a standard application program interface that makes no assumptions about the user interface style.

The beginnings of a design for such an interface have been presented. The suggested design is based on abstract data types whose instances are to be visible to
the user. Some difficulties that will need to be overcome have been discussed, and suggestions have been made as to how these difficulties might be addressed.
This chapter discusses issues involved in the design of a geographic database that will support interactive, visual browsing.

Section 7.1 covers issues relating to the design of the data model, or high-level view of the data. Some general issues are introduced, followed by some that are specific to visual browsing.

In Section 7.2, a data model for browsable geographic data is developed that addresses the issues introduced in Section 7.1. The model described will be used as a reference model for discussion in following chapters.

7.1 Geographic data models

A geographic information system holds a collection of information about geographic entities. In work done by the US National Committee on Cartographic Data Standards, the term entity has been defined as "a real world phenomenon that is not subdivided into phenomena of the same kind".

A geographic entity is one which has a location on the earth’s surface. Geographic entities may be tangible ones such as roads and bridges, or artificially defined ones such as census districts or parcels of land. A GIS may also handle information that varies continuously from point to point, such as elevation or climatic conditions.

The NCCDS also defines two other terms, object and feature. An object is a piece of digital information in a GIS which represents all or part of an entity. A feature refers collectively to an entity and the objects representing it (Figure 7.0).

7.1.1 Spatial and non-spatial data

Most pieces of geographic data can be classified into two types:
(1) *Spatial* data, which describes the physical location, size and shape of entities and the topological relationships between them, or data that varies continuously over the surface of the earth.

(2) *Non-spatial* data, describing any other characteristics of entities that are of interest to the user of a GIS.

Non-spatial data is sometimes referred to as *attribute* data, although entities could be regarded as having a set of attributes of which some are spatial and others are non-spatial.

Mostly, non-spatial data consists of a simple set of attributes such as the name of a landmark or the area of a land parcel. Standard database management techniques are adequate for dealing with this type of data. Beyond this, little can be said in general about non-spatial data. The appropriate data model to use for non-spatial data in any given GIS will depend on the circumstances, and will often need to be modified and extended to cope with changes in requirements.

In contrast, spatial data has a very fixed and regular structure. All of the spatial attributes of an entity can be represented by a small number of geometric primitives, such as points, lines and areas, combined in certain predefined ways, so that there is little if any need for flexibility in the spatial data model.

Furthermore, operations performed on spatial data often rely on properties such as distance, adjacency and containment that are uniquely spatial in character. Currently available database management systems do not directly support the representation or use of these properties. As a result, the subject of storage and manipulation of spatial data poses unique challenges.
The main focus of this thesis will be on the spatial data component, with particular regard to the problem of interactive retrieval.

7.1.2 Raster vs. Vector representations

There are two main ways of representing spatial data, which are known as the raster and vector models (Feuchtwanger and Poiker 1986).

The raster model is suited to the representation of information which varies from point to point, such as soil type or average rainfall. The area is subdivided into a regular grid of cells, usually rectangular, although sometimes other shapes such as triangles or hexagons are used. For each cell, a representative value of the datum is recorded, such as the average value over the cell, or the value at the centre of the cell.

The vector model is suited to the representation of entities with sharply defined boundaries, such as roads, lakes and land masses, and particularly for artificial entities such land parcels and suburbs. The vector model represents entities in terms of geometric primitives - points for 0-dimensional entities like mountain peaks, lines for 1-dimensional entities like roads and rivers, and polygons for 2-dimensional entities like land parcels.

It is beyond the scope of this discussion to consider the advantages and disadvantages of one model or the other for various applications. It should be clear that the two models are complementary rather than competitive, each being suited to a particular domain of application.

The remainder of this thesis will concentrate mostly on the vector model, as it is the one most suited to the kinds of features found in cadastral data: entities such as land parcels and street boundaries are most naturally represented by lines and polygons.

7.1.3 Vector data models

A large amount of literature has been published concerning vector data models in GIS. A few of the more significant contributions are cited here to illustrate the progress that has been made.

Early spatial data models were very simple and designed mainly around the requirements of the input stage (Peucker and Chrisman 1975). Little topological information about the spatial relationships between entities was explicitly
represented and had to be calculated when needed. The term “spaghetti” is sometimes applied to this kind of model.

More modern approaches represent spatial data in the form of a topological network of nodes, lines and polygons. An example is the POLYVRT structure described by Peucker and Chrisman. In POLYVRT the basic unit of data storage is the *chain*, which is an unbroken line made up of a series of points. Each chain has a *node* at each end, and is stored with pointers to the two *polygons* on either side. Each polygon has a list of chains making up its boundary. Polygons adjacent to a given polygon can be found by going through the chains common to their boundaries.

Most vector-type spatial data structures can be seen as variations on this topological network theme. Shapiro and Haralick (1980) describe a data structure which they call “relational”, although “network” would be more in keeping with today’s terminology. They show how their structure can be used to mimic a variety of others, such as DIME (Cooke and Maxfield 1967), POLYVRT, and GEOGRAF (Peucker and Chrisman 1975). They present schemes for answering various queries using their data structure.

Hagan (1981) describes an implementation of a POLYVRT-like structure using a CODASYL network database. The structure consists of two levels, a lower level containing spatial data objects, and a higher level which ties the spatial objects together to represent real-world entities.

Palimaka, Halustchak and Walker (1986) describe a GIS divided into spatial and non-spatial components, with the non-spatial data held in a relational database. They use a quadtree (Klinger and Dyer 1976) as a spatial index to facilitate retrieval of data by geographic area.

### 7.1.4 Browsing-specific issues

Efficient implementation of the browsing operations introduced in Chapter 2 will require some specialised support from the data structure. Provision will need to be made for:

1. the assortment of entities into layers,
2. viewing the data at different scales or viewing levels, and
3. retrieval by geographic area.
7.1.5 Layers

A *layer* as defined in Chapter 2 can be thought of as a set of entities defined by a predicate involving non-spatial attributes. For instance, a "Main highways" layer might be defined as the set of entities whose *type* attribute equals "Road" and whose *class* attribute equals "Major".

As such, no special support for the concept of a layer is necessary besides ensuring that the relevant attributes are stored somewhere in the database. For interactive browsing, however, layer membership will be one of the primary criteria for selecting entities, so that some form of indexing by layer would seem to be desirable.

Each layer can be regarded as a predefined query involving non-spatial attributes. These queries can be pre-executed and the results stored for later use. Information about each layer defined can be kept in a *layer catalogue* from which the user selects layers of interest.

7.1.6 Generalisation

The need to display the same entities at a wide range of viewing levels leads into the area of cartographic *generalisation*, or the process of simplifying a set of data to generate maps at different scales (Monmonier 1982). Generalisation involves three main operations:

1. Feature selection: Choosing the most important features and retaining them, while discarding the remainder to avoid clutter.

2. Feature simplification:
   - Smoothing out minor bumps and wiggles in line features while retaining their visually dominant characteristics.
   - Changing from one form of symbology to another, such as from a double to a single line for a road, or from an irregular area to a dot for a town or city.
   - Exaggerating the size of small but important features to make them visible.

3. Feature placement: Features that are too close together to display in their true positions at the chosen scale may need to be moved apart.
Parts of the generalisation process can be automated; for example, various algorithms exist for line reduction (Douglas and Peucker 1973, Palmer 1983, Muller 1987). However, the interactions between the three kinds of operations, and the aesthetic judgements involved, make the generalisation process difficult to automate fully, perhaps impossible.

Even with suitable algorithms, it is unlikely that data stored at the most detailed resolution could be retrieved and generalised quickly enough "on the fly" to allow interactive browsing without unacceptable delays.

Two ways of addressing this problem are: (1) to store multiple representations of the data at different resolutions, and (2) to use variable-resolution data structures.

7.1.7 Multiple representations

The database is assumed to be modified relatively infrequently, so a reasonable approach is to pre-generalise the data, creating alternative representations for display at a variety of smaller scales. Since the amount of data required to represent a small-scale map is much less than that for a large-scale map of the same area, the extra storage space required should be minimal.

Van Oosterom and van der Bos (1989) suggest a rule of "constant picture information density" for displayed maps: the total amount of information displayed on one screen should be about the same for all scales.

If we adopt this principle, then a scale reduction of $r$ results in a data reduction of $r^2$. So, for example, a scale ratio of 10 between representations would result in each representation requiring $1/100$ of the storage space of the next most detailed one. A stack of any number of these extra representations could therefore be stored at an overhead of less than 1.2%.

7.1.8 Variable-resolution data structures

An alternative to storing multiple representations of the data is to store a single representation in a variable-resolution data structure. Such a structure has the property that a variety of representative subsets of the data can be retrieved from it at a cost proportional to the size of the subset required.

An example of a variable-resolution data structure for vector data is the striptree (Ballard 1981) for storing line features represented as a sequence of points.
The striptree is a binary tree. The root node approximates the whole line with a straight line segment between its two endpoints, together with a bounding rectangle which just encloses all the points in the line (Figure 7.1a).

The line is then split in two at the point which is farthest from the line connecting the two endpoints, and the two halves are recursively approximated in the same way (Figure 7.1b) until only straight line segments remain.

The distance $h$ between the centre line and the extreme point of each strip a measure of how good an approximation is obtained by replacing that strip with a straight line. So, given some tolerance $\delta$, the tree can be descended until nodes are reached whose $h$ is no greater than $\delta$, and the resulting approximation will deviate by no more than $\delta$ from the actual line.
In fact, the approximation obtained is the same as that which would result from applying the Douglas-Peucker line reduction algorithm (Douglas and Peucker 1973) to the original data with a tolerance of $\delta$.

The striptree also has other properties which are of use in a GIS. Given a search window, the tree can be efficiently searched to find strips whose bounding rectangles intersect the window, so the tree can be used as a form of spatial index. A variety of other operations can also be performed quickly on lines represented as striptrees, by making use of intersection information between bounding rectangles (Samet 1990).

7.2 A browsable geographic data model

In this section, a data model will be developed for a geographic database designed to support interactive browsing, to be used as a backdrop for further discussion. The model will be described in fairly general terms. The details of some parts will be left unspecified at this stage, either because they are not relevant to the problem of interactive browsing, or because they may be elaborated in a variety of ways which will be discussed later.

7.2.1 Terminology

Many terms in the field of GIS are used by different authors in slightly different ways. The following definitions are based where possible on the proposed U.S. Spatial Data Transfer Standard (SDTS 1990).

*Entity*
A real world phenomenon.

*Object*
A digital representation of all or part of an entity.

*Feature*
A entity and its object representation.

*Point*
A 0-dimensional object that specifies a geometric location.

*Node*
A 0-dimensional object that is a topological junction or end point.

*Line*
A 1-dimensional object.

*Arc*
A locus of points that forms a curve defined by a mathematical function.
Chapter 7 – Database Design for Interactive Browsing

7.2.2 Entities and their representations

Each entity will be represented in the database by an object, part of which will specify the spatial attributes of the entity, with the remainder specifying non-spatial attributes. Figure 7.2 shows the relationship between entities, spatial objects and non-spatial attributes.

7.2.3 Spatial objects

The SDTS distinguishes two kinds of spatial information: geometric (specifying location and shape) and topological (specifying connectivity between objects). Of these two, only geometric information is strictly necessary for the purpose of displaying maps. Having topological information available is, however, of great advantage when performing other common operations, and many geographic databases include both kinds of spatial data.

As remarked earlier, spatial data models tend to be variations on the same theme. Figure 7.3 shows one way of modelling geometric data. This diagram is intended to convey that a spatial object can be either a point, line or area, and that lines are defined in terms of a sequence of connected points, while areas are defined by their boundaries, each of which is a closed sequence of lines.
Being purely geometrical, this model does not include any explicit relationships between different spatial objects.

Topological information is often represented by organising the geometric objects into a topological network of nodes connected by lines. Figure 7.4 shows a possible way of augmenting the model of Figure 7.3 to include topological information of this sort.

For the resulting network to be topologically consistent, chains must not be allowed to intersect; so, provision has been made in Figure 7.4 for a line object to be defined in terms of a collection of chains.

There are many other ways in which a topological data model could be arranged.
Figure 7.5 shows an alternative way of representing areas. The chains in the network partition the plane into a set of non-overlapping *simple areas* from which more complex areas can be built up.

This scheme has the advantage that it provides a simple and unambiguous way of representing disconnected areas or areas with holes. If an area is represented in terms of boundary rings, questions arise as to the meaning of nested or intersecting rings; resolving these questions usually requires making *ad hoc* definitions or restrictions. Representation in terms of simple areas avoids all such questions and provides a unique representation for any given area.

On the other hand, the model of Figure 7.5 may be less desirable than that of Figure 7.4 for some operations. For instance, to display the boundary of an area defined in terms of simple areas, it would be necessary to first find all the simple areas involved, then find all the chains adjacent to these simple areas, and then to eliminate those chains for which the areas on both sides are both part of the original area, to leave the chains forming the actual boundary.

By contrast, the model of Figure 7.4 allows the required chains to be retrieved directly and displayed without any further processing. So, it may be a better choice for interactive browsing, where retrieval for display is the main concern.

### 7.2.4 Object sharing

Often a given spatial object will form part of the description of more than one entity. There are two ways in which this sharing of spatial objects may arise.

![Spatial Object Diagram](image-url)
A hierarchy of entities sharing spatial data

1. An entity is defined in terms of other entities. For example, a city block might be defined as a collection of land parcels, each of which is represented spatially by an area object. Each area object then describes both a Parcel and part of a City Block. This situation can be modelled by arranging entities in a hierarchy as shown in Figure 7.6.

2. Two otherwise separate entities share part of their spatial definition. For example, two adjacent land parcels will have part of their boundary in common. Or, a certain land parcel might be legally defined so that part of its boundary coincides with a river; if the river changes course, the boundary should change as well.

These situations are easily accommodated within the spatial data models of Figures 7.4 or 7.5, as illustrated in Figure 7.7.

7.2.5 Layers, features and representations

The basic query executed by the map browser involves three keys: the viewing level, the layers required, and the spatial domain required. It is assumed that there will be many more layers in the database than the user will want to view at any one time, so it is sensible to use the layer as the most significant key for narrowing down the scope of the search.

Figure 7.8 shows a data model for incorporating layers into the database. This model allows for layers to be organised into a hierarchy of groups and indexed by a layer catalogue. The catalogue contains descriptive information about each layer to help the user in selecting layers to view.
Each layer consists of a collection of symbols, or visual representations of entities. Each entity may be represented by several alternative symbols, designed for display at different viewing levels. Each symbol is associated with a spatial object that defines the symbol's appearance. For the most detailed representation, this will be the base object defining the entity.

Having selected data by layer, it remains to carry out the selections by domain and viewing level. The internal organisation chosen for layers will depend on which of

---

1 In part; some characteristics of symbols, such as colour, shading, line width, and so forth should be under the control of the user. These are not stored in the database, except perhaps as default values.
Experiments in Interactive Map Retrieval

these remaining keys is used as the more significant key, and which as the less significant.

Selection by domain will require the use of a spatial index, represented by the shaded items in Figure 7.8. A wide variety of spatial indexing structures could be used, some of which will be examined in later chapters. For now, the form of spatial index used will be left unspecified.

If the domain is used as the more significant key, then each index entry will consist of a collection of symbols representing the same entity at different resolutions, from which one representation is chosen based on the viewing level.

This arrangement has the disadvantage that, at high viewing levels (small scales) much irrelevant data may need to be sorted through, since relatively few entities in the layer will be represented at that level.

Figure 7.9 shows an alternative arrangement, in which the viewing level is used as the more significant key to select one of several representations of an entire layer, followed by selection of symbols using the domain as the less significant key. Each layer representation would contain at most one symbol representing each entity, and all the symbols in a given layer representation would be of a similar resolution.
Under this arrangement, layer representations for viewing at small scale contain only symbols suitable for display at that scale, and therefore are relatively small.

As well as making high-viewing-level retrievals more efficient, less storage space is required, since information about the resolution of symbols only needs to be stored for each layer representation, rather than for each symbol.

Another possible approach is to combine indexing by domain and viewing level, by using a variable-resolution data structure. Under this scheme, the domain and viewing level would be regarded together as a single key and used to search a tree structure, with the domain determining the branches taken, and the viewing level determining the depth to which the search descended. Low-resolution symbols would be stored near the root, and high-resolution symbols near the leaves.

7.2.6 Retrieving parts of a symbol

Some entities may cover a large area and require a large amount of data to represent them. For example, the coastline of New Zealand at the most detailed level required would contain many thousands of points. When viewing the coastline at this level, however, only a small portion of it will be in view at any moment.

If the coastline is stored as a single spatial object, and represented by a single symbol in the relevant layer, it will be necessary to retrieve the whole object in order to display even a small part of it, which is clearly not very efficient.
This problem can be overcome by splitting up the coastline into many smaller objects, each represented by a separate symbol. Looking up the layer’s spatial index would finds only those parts of the coastline which intersect the search window.

A more elegant solution is to store the spatial objects themselves in a spatially-organised structure, such as a quadtree or striptree, to enable retrieval of those parts of objects intersecting the search window. Use of a variable-resolution data structure such as the striptree would have the added benefit of allowing the same objects to be used for a wide range of viewing levels.

7.2.7 Data clustering

To maximise speed of retrieval, it is desirable to arrange for data that is frequently retrieved together to be clustered together on the storage device. Here, this means that spatial objects associated with symbols in a given layer should be clustered by spatial location, so that symbols which are nearby geographically are likely to be stored together.

The same techniques used for spatial indexing can also be used to achieve spatial clustering; for example, space can be subdivided in a quadtree fashion until cells contain a small enough amount of data to store in a single disk block.

However, symbols belonging to different layers may have parts of their spatial definition in common, as would arise for instance in Figure 7.7 if “Rivers” and “Parcels” were assigned to different layers.

If cross-layer object sharing occurs frequently, conflicting clustering requirements are likely to arise. For instance, to maximise retrieval speed of “Rivers”, chain C2 in Figure 7.7 should be stored with other river data for the same area, whereas to maximise retrieval speed of “Parcels” it should be stored with the other parcel data for that area.

The best way to resolve this problem will depend on whether the layers involved are more often viewed together or separately. If they are usually viewed together, their data can be combined for the purpose of spatial clustering. It may be advantageous to do so in this case even if there is no sharing between the layers.

If the layers are usually viewed separately, it will be better to separate the data which is not shared, and to cluster the shared data with that of the layer which is viewed more often.
Another approach is to store multiple copies of the shared data, so that the features of each layer have their own, private set of spatial objects that can be clustered together without affecting other layers. This approach would be a tradeoff between retrieval speed and increased storage space and redundancy.

If every layer were to have its own set of spatial objects, with no sharing between layers at all, then the spatial indexing structure of each layer could be replaced by a spatial data structure, holding the spatial data itself instead of pointers to another part of the database. The same data structure would then provide both the spatial indexing and clustering functions.

7.2.8 Summary

The various parts of the data model described so far are pieced together in Figure 7.10. The data model can be seen as comprising two main parts, a general-purpose part and a browsing-specific part. The general-purpose part consists of geographic entities, the spatial data defining them, and any other non-spatial data associated with them; the role of this part could be filled by any existing geographic database.

The browsing-specific part extends the general-purpose database with structures designed to support interactive browsing. Entities are organised into a collection of layers, wherein they are represented by symbols. A layer catalogue and a set of spatial indexes facilitate retrieval of features by layer and geographic area.

An entity may be represented by more than one symbol, designed for display at different scales. Each symbol is associated with a spatial object, either the entity's defining object or a simplified one derived from it by cartographic generalisation.

7.3 Conclusions

An extension to a general-purpose geographic data model to support interactive browsing has been proposed. The model allows for the organisation of entities into layers, different visual representations of entities at different scales or viewing levels, and the efficient retrieval of spatial data by layer, geographic area and viewing level.

Issues of data sharing and data clustering have been discussed, particularly in regard to their implications for rapid spatial retrieval, and the tradeoffs that may be required between retrieval speed, storage space and redundancy.
Alternative elaborations of parts of the data model have been briefly explored. The main area requiring further investigation is how best to index features by layer, geographic area and viewing level to achieve the most efficient interactive retrieval.
Chapter 8

Introduction to Experiments

This and the following chapters describe a set of experiments that were conducted with the aim of comparing a variety of spatial data storage, indexing and retrieval methods with regard to their suitability for interactive browsing.

This chapter sets out the overall objectives of the experiments, describes the hardware and software environment in which the experiments were performed, and discusses the methodology employed. Detailed descriptions of each of the experiments and the conclusions drawn will be presented in the following chapters.

8.1 Experimental Objectives

The relational database has proven to be a successful and powerful tool for storing and manipulating data in a very flexible manner. Hence, it is logical to consider the use of an RDBMS for storing and manipulating geographic data.

However, conventional relational database systems provide storage and indexing structures suited only to one-dimensional data (ISAM indexes, B-trees, etc.). These facilities are not directly suitable for handling the spatial components of geographic data. Spatial data is inherently multidimensional, and the operations frequently applied to it (such as containment, intersection and proximity testing) are intimately tied to its spatial character.

The result is that a conventional RDBMS, while very well suited for the non-spatial components of geographic data, does not provide facilities which directly support the manipulation of spatial data.

There are three main approaches to handling geographic data involving a relational database. Briefly, they are:

(1) Geographic data is commonly divided into spatial and non-spatial parts, with the non-spatial data being held in a relational database and the spatial data (or at
least the spatial indexes) being stored in separate, custom-designed data structures. An example is the ARC/INFO system (Dangermond 1983, ESRI 1985).

(2) Another approach is the use of an extensible RDBMS, such as Postgres (Stonebraker and Rowe 1986, Stonebraker, Rowe and Hirohama 1990) enhanced with special storage structures and operations designed specifically for spatial data (for example, van Oosterom and Vijlbrief 1991). A number of other extensible database management systems are reviewed by Batory and Mannimo (1986).

(3) A third approach is the use of a conventional RDBMS to implement spatial indexing structures such as k-d trees, quadtrees and R-trees. In this approach, the facilities provided by the underlying RDBMS are supplemented by further software layers built upon them. For example, Abel and Smith (1983) describe a method of using a standard B-tree structure to provide a spatial index.

Much of the work presently being done on GIS is aimed at providing a general "geographic query language", along the lines of existing relational query languages such as SQL.

These languages include predicates such as "contains" and "intersects" for operating on spatial data, allowing the user to formulate complex queries involving spatial and non-spatial data in a uniform way (Abel and Smith 1986, Ooi 1990). In such a system, generality and flexibility are the primary requirements, speed being a secondary consideration.

Supporting an interactive map browser, on the other hand, imposes a different set of requirements. There is essentially only one type of spatial query to handle, the window query, so that a general spatial query processor is not needed. Furthermore, speed is of vital importance if adequate interactive responsiveness is to be achieved.

The overall objective of the set of experiments described here was to gain insight into the kinds of data structures and algorithms required to support an interactive map browser. Most of the work was done using a conventional RDBMS, because the flexibility of the relational database allowed a wide variety of data structures and algorithms to be tried out quickly and easily.

This flexibility comes at a price in efficiency, and it is possible that a relational DBMS may introduce too much overhead to be usable in a practical browsing system. Nevertheless, much of what has been learned is of a general nature, and could be applied to the design of browsers for geographic information systems based on other types of DBMS.
A secondary objective was to find out whether a relational DBMS could, in fact, be used to support a practical map browser. Even in a browsing system, the advantages of the relational database are still desirable for updating the data and using the database for other purposes. It would be very convenient if the same database could support all activities.

8.2 Experimental environment

This section describes the test database and the hardware and software used to carry out the experiments.

8.2.1 The data

It was desired to experiment with data that was being used in an actual application, so a sample of geographic data was obtained from the Christchurch Drainage Board. It consists of cadastral information, locations of stormwater and sewer pipes, and other related information for some selected regions of the Christchurch area.

As supplied, the data consisted of four types of objects:

- Point
- Circle
- Curve - a sequence of connected line segments and circular arcs
- Text block

Attribute information associated with each object was used to separate objects into layers according to the type of entity they represent. The main layers include:

- Street outlines
- Property boundaries
- Building outlines
- Locations of stormwater and sewer pipes
- Locations of manholes

A number of other layers were included, such as street names, street address numbers and legal descriptions. Table 8.1 shows a complete list of layers.

The data supplied by the Drainage Board was augmented with a small amount of data from other sources to provide a small-scale map of the Christchurch area
showing main streets, rivers, major parks, and urban boundaries. This map was used as a reference map for the browser.

<table>
<thead>
<tr>
<th>Table 8.1 - Database Layers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Drainage Board data</strong></td>
</tr>
<tr>
<td>Building outlines</td>
</tr>
<tr>
<td>Easements</td>
</tr>
<tr>
<td>Foulwater sewers</td>
</tr>
<tr>
<td>House numbers</td>
</tr>
<tr>
<td>Legal descriptions</td>
</tr>
<tr>
<td>Private foulwater drains</td>
</tr>
<tr>
<td>Private stormwater drains</td>
</tr>
<tr>
<td>Rivers</td>
</tr>
<tr>
<td>Section boundaries</td>
</tr>
<tr>
<td>Sewer manholes</td>
</tr>
<tr>
<td>Stormwater manholes</td>
</tr>
<tr>
<td>Stormwater sewers</td>
</tr>
<tr>
<td>Street names</td>
</tr>
<tr>
<td>Street outlines</td>
</tr>
</tbody>
</table>

8.2.2 The schema

The data was converted to a relational form and stored in a relational database. The initial database schema was derived in a straightforward manner from the structure of the original data. No provision was made for topological information, because the original data only contained geometric information, and no topological information was required for the experiments. Nevertheless, the schema could easily be extended to include topological information along the lines discussed in Chapter 7.

Figure 8.1 depicts the relations used and the connections between them. Central to the database are the six relations `object`, `point`, `circle`, `curve`, `textblock` and `textline`. Each object is assigned a unique object ID, and has a tuple in the `object` relation which specifies the type of object (POINT, CIRCLE, CURVE or TEXT) together with its minimum bounding rectangle.

Each layer has a tuple in the `layer` relation which names the layer and associates a unique layer ID with it. Each layer has one or more `representations` (as discussed in Chapter 7). The `reps` relation contains a tuple for each representation of each layer, specifying the range of viewing levels for which objects in the representation are suitable. Each object belongs to one representation, identified by the `repid` field of its `object` tuple.
The four types of spatial object are held in the \textit{point}, \textit{circle}, \textit{curve}, \textit{textblock} and \textit{textline} relations. The \textit{point} and \textit{circle} relations are reasonably self-explanatory. The \textit{curve} relation contains lists of endpoints which are understood to be connected by straight line segments or circular arcs. The \textit{seq} field contains sequence numbers to preserve the ordering of points within a given \textit{curve} object (all points belonging to one
curve object have the same objid). The bulge field contains either zero for a straight line segment, or information which specifies the radius of an arc.

The textblock relation contains a tuple for each text object, specifying the box within which the text should appear, how it should be justified within the box, and the angle of rotation of the box. The box may contain one or more lines of text, and these are held in the textline relation, one tuple per line.

The object relation also has a source ID field, which is used to tag each object with information about the source from which it was obtained, for administrative purposes. The source relation associates a source ID with a name.

8.2.3 Databases constructed

To facilitate testing new algorithms, two other databases were constructed, having the same schema but containing subsets of the data in the full database. Table 8.2 lists some statistics concerning the three databases. Database 3 contains all the data; database 2 contains data for a small subregion, and database 1 contains data for the same subregion but from selected layers only.

Although the sample data supplied contained no point objects, provision was made for them in the schema to accommodate possible future data acquisitions.

<table>
<thead>
<tr>
<th>Database</th>
<th>Objects</th>
<th>Points</th>
<th>Curves</th>
<th>Number of...</th>
<th>Circles</th>
<th>Text Blocks</th>
<th>Text Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Points in Curve objects</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>53</td>
<td>0</td>
<td>93</td>
<td>58</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>720</td>
<td>0</td>
<td>508</td>
<td>2674</td>
<td>101</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>3</td>
<td>22915</td>
<td>0</td>
<td>19221</td>
<td>56616</td>
<td>1930</td>
<td>727</td>
<td>728</td>
</tr>
</tbody>
</table>

8.2.4 Hardware Environment

The experiments were carried out using a network of Sun SPARCStations connected by Ethernet. Two main machines were involved in the experiments, a file server with large disk capacity shared by all users of the network, and a workstation with local disks and a colour display. Figure 8.2 shows the configuration of the relevant part of the network.
8.2.5 Software Environment

The map browser was structured along the lines of Figure 2.2, with the user interface and retrieval engine combined into a single program.

The first version of the user interface was implemented using NeWS and the Mockintosh user interface library described in Chapter 5. Later the user interface was re-implemented for X11 using the InterViews library referred to in Chapter 3, so that it could be used with a wider variety of workstations and graphics terminals.

The browser was structured so that a variety of retrieval engines could be installed for the purpose of evaluating different retrieval techniques.

For the DBMS, the Ingres relational database system was used. Two versions of Ingres were available, a freeware version called University Ingres, or UIngres (Stonebraker, Wong and Kreps 1976; Kalash, Rodgin, Fong and Anton 1986), and a commercial version from Relational Technology Inc. (RTI Ingres, or Commercial Ingres).

UIngres had the advantages that it was relatively small and self-contained, required no elaborate installation procedures, was generally well-behaved, and the source code was available to help in tracking down any problems.
On the other hand, RTI Ingres offered some facilities, such as B-tree indexes, that UIngres lacked, and also appeared to be more effective at query optimisation in some cases.

RTI Ingres supports two query languages, SQL and Quel, whereas UIngres supports only Quel. So that the map browser could be used with either DBMS, Quel was used exclusively. Where possible, experiments were performed using both UIngres and RTI Ingres so that the results could be compared.

### 8.3 Experimental Design

In each experiment, the aim was to assess the effect on some performance index of some change to the system, such as a change in database structure, retrieval algorithm or network configuration. To do this, the same set of retrieval operations were performed before and after the change, and the resulting values of the performance index compared.

To ensure repeatability of the retrieval operations, a benchmark script was used. The script was created by instrumenting the map browser to record sequences of retrieval operations made by a user during an interactive session.

#### 8.3.1 Performance indices

Ultimately, the aim is to minimise response time to the user, so the most direct and reliable performance index would be the interactive response time $T_R$, which could be defined as the time between the user issuing the retrieval command, and the last piece of data being displayed. $T_R$ also has the advantage of being easy to measure.

#### 8.3.2 Secondary factors

Ferrari (1978) classifies factors affecting the outcome of an experiment into primary and secondary factors. Primary factors are those whose effect the experimenter is interested in studying, whereas secondary factors are any other factors that may substantially affect the experiment and therefore must be taken into account.

A problem with using $T_R$ as a performance index is that it depends critically on many secondary factors that are difficult to measure or control, such as

- Contention for CPU time, physical memory and disk access by other processes running on the same machine.
• The time taken to service requests to the file server, which will be affected by both activity on the file server and general traffic on the network.

• The effect of contention on patterns of cacheing activity in various memory buffers and kernel file caches.

The effects of some of these secondary factors could be isolated by breaking down the retrieval time into components due to the map browser and other processes. However, the Unix system does not allow such a detailed breakdown to be obtained easily, if at all.

The amount of CPU time used by individual processes can be measured, and a count of the number of block I/O requests made by a process can be obtained. However, it is not feasible to determine how much of the time spent waiting for I/O operations is due to contention from the I/O of other processes. Furthermore, when a file server is involved, there is no way of distinguishing between file server requests from different sources.

Even if a suitably detailed breakdown of retrieval time were available, effects due to cacheing would still remain. Retrievals performed on a lightly loaded system may be faster due to the availability of more memory for file cacheing.

Other ways of dealing with the effects of secondary factors on $T_R$ include:

(1) Controlling the environment:

Arrange for dedicated machines on which to run all parts of the system, including the file server, on an isolated network. Although this eliminates many effects due to contention, it does not take into account the effects on cacheing. Results obtained under single-user conditions might degrade more than expected in the face of contention, due to lower cache hit rates.

(2) Averaging:

Perform the experiments a large number of times, under a wide variety of workload conditions, and average the results. This method is potentially the most accurate, since by its nature it takes all factors into account. However, depending on how long it takes to perform an experimental run, the time taken to perform such comprehensive testing maybe prohibitive.

(3) Find an alternative performance index.
An alternative performance index might be

\[ I = k_1 T_{CPU} + k_2 N_{I/O} \]

where \( T_{CPU} \) is the total CPU time used by the map browser back end and the DBMS, and \( N_{I/O} \) is the total number of block I/O requests made. The constants \( k_1 \) and \( k_2 \) would be chosen according to the relative contribution of these two factors to total response time.

For instance, we could set \( k_1 = 1 \) and \( k_2 = T_b \), where \( T_b \) is the average time taken to perform a block I/O operation under various workloads, determined by experiment. By adjusting \( k_1 \) we could also take into account the effect of different processor speeds and architectures between different machines in the system.

As well as time-related measures, it may be desirable to measure other performance indices, to take into account issues that may be overlooked in concentrating solely on response time. For instance, a retrieval algorithm might achieve excellent response time at the expense of requiring very large amounts of memory, suggesting that memory requirements should also be considered when evaluating a retrieval strategy.

Methods that might be used to measure memory requirements:

- Predict by calculation, where possible, from knowledge of the algorithm.
- Measure average working set size.
- Measure the page fault rate.

8.3.3 Procedure adopted

To provide a controlled environment, where possible experiments were carried out using the configuration shown in Figure 8.3, with all components residing on a single, dedicated machine. With no other users competing for any of the resources, a simple measurement of \( T_R \) can be used as a reasonably reliable performance index.
In addition, a number of runs of each experiment were performed, so that an average could be taken, and the variance between runs used to gauge the reliability of the results.

These evaluation methods proved to be adequate for the purpose of the experiments, which was to gain insights into the relative merits of different storage and retrieval techniques, and to draw conclusions on good practices.

### 8.3.4 Benchmarks used

The benchmark scripts were designed to model a typical user action, that of repeatedly zooming in on smaller subregions to find a feature of interest. Two benchmarks were constructed, A and B. Benchmark A was further subdivided into three cases, A1 to A3, each tailored for one of the three test databases.

#### Table 8.3a

<table>
<thead>
<tr>
<th>Level</th>
<th>minx</th>
<th>miny</th>
<th>maxx</th>
<th>maxy</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>2473.292</td>
<td>5745.729</td>
<td>2473.524</td>
<td>5746.088</td>
</tr>
<tr>
<td>2.3</td>
<td>2473.400</td>
<td>5745.795</td>
<td>2473.439</td>
<td>5745.830</td>
</tr>
<tr>
<td>1.5</td>
<td>2473.409</td>
<td>5745.800</td>
<td>2473.416</td>
<td>5745.806</td>
</tr>
</tbody>
</table>

#### Table 8.3b

<table>
<thead>
<tr>
<th>Level</th>
<th>minx</th>
<th>miny</th>
<th>maxx</th>
<th>maxy</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>2463.180</td>
<td>5730.163</td>
<td>2497.752</td>
<td>5748.938</td>
</tr>
<tr>
<td>4.3</td>
<td>2478.945</td>
<td>5736.593</td>
<td>2481.917</td>
<td>5738.252</td>
</tr>
<tr>
<td>3.3</td>
<td>2479.194</td>
<td>5736.780</td>
<td>2479.422</td>
<td>5736.995</td>
</tr>
<tr>
<td>2.4</td>
<td>2479.266</td>
<td>5736.905</td>
<td>2479.323</td>
<td>5736.949</td>
</tr>
</tbody>
</table>

#### Table 8.3c

<table>
<thead>
<tr>
<th>Level</th>
<th>minx</th>
<th>miny</th>
<th>maxx</th>
<th>maxy</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>2463.180</td>
<td>5730.163</td>
<td>2497.752</td>
<td>5750.628</td>
</tr>
<tr>
<td>4.3</td>
<td>2473.897</td>
<td>5735.625</td>
<td>2475.419</td>
<td>5737.769</td>
</tr>
<tr>
<td>3.5</td>
<td>2474.216</td>
<td>5736.912</td>
<td>2474.596</td>
<td>5737.472</td>
</tr>
<tr>
<td>2.5</td>
<td>2474.380</td>
<td>5737.050</td>
<td>2474.417</td>
<td>5737.083</td>
</tr>
<tr>
<td>1.5</td>
<td>2474.403</td>
<td>5737.061</td>
<td>2474.407</td>
<td>5737.065</td>
</tr>
</tbody>
</table>

Tables 8.3a-c show the geographic regions used in each of the Benchmark A scripts, expressed in NZ Map Grid coordinates, and the associated viewing level at which each region was displayed. The first region of each script covers the entire domain of
the database; each subsequent region is a subset of the previous one, displayed at a scale factor approximately 10 times larger. (Departures from a factor of 10 are due to the map browser adjusting the window size to fit on the screen.)

Benchmark B was used to assess average behaviour over a large number of retrievals. This benchmark was applied only to Database 3.

100 points were chosen, uniformly distributed over those regions of the database containing non-trivial amounts of data. For each point, a nested sequence of retrieval windows centred on the point was created, each window being $1/10$ the size and displayed at 10 times the scale of the previous one.

These retrievals were designed to simulate a user zooming in on some feature of interest. Four retrieval windows were generated for each point, resulting in a total of 400 retrieval operations with viewing levels ranging from 4.3 down to 1.3.
Chapter 9

Preliminary Experiments

This chapter describes the development of a basic retrieval algorithm for answering window queries using the database schema of Chapter 8, and the results of some preliminary experiments to assess the optimum choice of Ingres storage structures for the relations holding spatial data. Benchmark timings obtained using a very simple form of spatial indexing are presented, for use as a control case to compare with later results.

9.1 Basic retrieval algorithm

As discussed in Chapter 2, the basic window query performed by the map browser can be described as

\[
\text{retrieve(level, layers, domain)}
\]

where \textit{level} is the viewing level, \textit{layers} is the set of layers that the user has selected, and \textit{domain} is the geographic region for which data is wanted.

The retrieval algorithm can be divided into three main parts:

1. Find a representation of each layer suitable for display at the given viewing level.
2. Find the IDs of all objects in those representations whose bounding rectangles intersect the domain.
3. Retrieve the spatial data associated with those objects.

Finding the relevant representations is straightforward and is not a particularly time-consuming part of the retrieval process, and so will not be considered in detail here. In the following discussion we assume that the IDs of the relevant representations have been found and placed in a temporary relation \textit{laytemp}:

\[
\text{laytemp(layid = i4, repid = i4)}
\]

A simple C/Quel implementation of the retrieval algorithm is presented in Figure 9.0. The procedures \textit{draw_point}, \textit{draw_line} and so on are assumed to transform the coordinates to the viewport's coordinate system and display the objects using symbol styles appropriate to the layer.
9.2 Ingres storage structures

Ingres provides a variety of storage structures for relations. The storage structures available and their properties are:

- **Heap:** An unordered, unindexed collection of tuples.
- **Sorted heap:** The tuples are sorted on specified fields, but no indexing is provided.
- **Hash table:** Tuples are indexed by specified key fields and stored in a hash table.
- **ISAM:** Indexed-Sequential Access Method. Tuples are sorted on specified key fields, and an index is provided to facilitate access by key value or range of key values.
- **B-Tree:** (RTI Ingres only) Tuples are stored in a B-tree, indexed by specified key fields.

Also, any of the above structures may be compressed, meaning that trailing blanks on strings are not stored to save space.

In addition to an index forming part of its primary storage structure, a relation may have any number of secondary indexes, which can be structured in any of the above ways.

The speed of the retrieval algorithm can be expected to depend heavily on the storage structures chosen for the object, point, circle, curve, textblock and textline relations. The range of reasonable choices of storage structure for these relations can be narrowed down by the following considerations.

9.3 The Object relation

9.3.1 Theoretical considerations

The algorithm presented above selects tuples from the object relation according to two criteria: (1) a set of repid values, and (2) range comparisons between the window boundaries and the minx, miny, maxx and maxy fields. Clearly, some combination of these fields should be included in the key in some order, and a storage structure used that orders tuples by the bounding rectangle coordinates in some way. However, it is not immediately apparent how best to arrange the key fields.
Chapter 9 – Preliminary Experiments

/* Algorithm 9.1 - Simple retrieval algorithm. Assumes that the layer and representation IDs for this retrieval have been found and placed in laytemp */
Retrieve(domain) Rect domain; {
    RetrieveObjects();
    RetrievePoints();
    RetrieveCircles();
    RetrieveCurves();
    RetrieveText();
}

/* Algorithm 9.1a - Retrieve object IDs and place them in the temporary relation objtemp */
RetrieveObjects() {
    retrieve objtemp(objid = object.objid, layid = laytemp.layid)
    where object.repid = laytemp.repid and
        object.minx <= domain.maxx and object.miny <= domain.maxy and
        object.maxx >= domain.minx and object.maxy >= domain.miny
}

/* Algorithm 9.1b - Retrieve point object data */
RetrievePoints() {
    int layid; float x, y;
    retrieve (layid = objtemp.layid, x = point.x, y = point.y)
    where point.objid = objtemp.objid {
        draw_point(layid, x, y);
    }
}

/* Algorithm 9.1c - Retrieve circle objects */
RetrieveCircles() {
    int layid; float x, y, r;
    retrieve (layid = objtemp.layid, x = circle.x, y = circle.y, r = circle.r)
    where circle.objid = objtemp.objid {
        draw_circle(layid, x, y, r);
    }
}

/* Algorithm 9.1d - Retrieve curve objects */
RetrieveCurves() {
    int layid, seq; float x, y, b, last_x, last_y;
    retrieve (layid = objtemp.layid, seq = curve.seq,
         x = curve.x, y = curve.y, b = curve.bulge)
    where curve.objid = objtemp.objid {
        if (seq != 0)
            if (bulge == 0) draw_line(layid, last_x, last_y, x, y);
            else draw_arc(layid, last_x, last_y, x, y, b);
        last_x = x; last_y = y;
    }
}

/* Algorithm 9.1e - Retrieve text objects */
RetrieveText() {
    int layid, seq; float x, y, angle; char chars[MAX_TEXT_LINE];
    retrieve (layid = objtemp.layid, x = textblock.x, y = textblock.y,
         angle = textblock.theta, seq = textline.seq, chars =
        textline.chars)
    where textblock.objid = objtemp.objid and textline.objid = objtemp.objid {
        draw_text_line(layid, x, y, angle, seq, chars);
    }
}

Figure 9.0
Basic retrieval algorithm
The problem of indexing by the bounding rectangle coordinates to facilitate intersection search can be more readily understood by considering the analogous problem in one dimension, which is easier to visualise.

As discussed by Samet (1989), a rectangle can be considered as a 2-dimensional interval. The one-dimensional equivalent of the window problem is: given a collection of one-dimensional intervals \((xmin, xmax)\) (analogous to the bounding rectangles of objects in the database) and a “window” interval \(W(wmin, wmax)\), find all the intervals that overlap the window interval, i.e. such that \(xmin \leq wmax\) and \(xmax \geq wmin\).

Each one-dimensional interval can be regarded as a point in a two-dimensional space, as illustrated in Fig. 9.1. Answering the window query in the one-dimensional case then corresponds to finding all the points which lie in the shaded subregion of Fig. 9.1b.

Suppose we were to order the collection of intervals by their \(xmin\) coordinates. We could then quickly find all the intervals satisfying \(xmin \leq wmax\), which would then need to be searched for those also satisfying \(xmax \geq wmin\). The shape of the resulting search region is shown in Fig. 9.2a. If we assume that the interval endpoints have coordinates that are uniformly distributed over the possible range of coordinate values, then on average only half the intervals retrieved will satisfy the window query.
In an attempt to make the search more selective, we might consider using \( \textit{xmax} \) as a secondary key, so that intervals with the same \( \textit{xmin} \) are ordered by their \( \textit{xmax} \). However, the improvement obtained is negligible. As shown in Fig. 9.2b, at most a small "kink" has been added to the edge of the search region, and even then the kink is only present if there happen to be points whose \( \textit{xmin} \) exactly equal \( \textit{wmax} \).
The same considerations apply to the use of $xmax$ or $(xmax, xmin)$ as keys, the search regions of Figs. 9.2c,d being obtained.

Generalising from intervals to rectangles, we regard a rectangle as a point in a 4-dimensional space. Finding rectangles which intersect the window corresponds to finding points lying within a 4-dimensional subregion analogous to Figure 9.1b. With uniformly distributed rectangle and window coordinates, the window subregion can be expected to contain on average about $(1/2)^4 = 1/16$ of the total number of points.

By using one of the rectangle coordinates as a key, we can reduce the number of rectangles to be examined to, on average, half the total number. Out of these, $1/8$ can be expected to lie within the window region, so that $7/8$ of the rectangles retrieved will be unwanted. As in the 2-dimensional case, adding more of the rectangle coordinates to the key is not likely to result in any useful improvement.

In conclusion, the best we can do with the $object$ relation in its present form is to use an ISAM or B-tree structure, indexed by $repid$ and one of the bounding rectangle coordinates, such as $minx$. The window query can then be answered in the following manner:

For each $repid$ value $r$:

For each tuple $o$ in $object$ such that $(o.repid, o.minx) \geq (r, 0)$ and $(o.repid, o.minx) \leq (r, domain.maxx)$:

If $(o.minx, o.miny, o maxx, o.maxy)$ intersects the domain,

return $o.objid$.

9.3.2 Experimental results

Tables 9.1-9.3 present Benchmark A results for a variety of $object$ relation storage structures and indexes. The figures listed are the average over three runs of the time taken (in seconds) to execute Algorithm 9.1a above.

In the UIngres case for Benchmark A3, retrieval 1 was spectacularly slow in comparison to the others. It was found that the UIngres query processor was not dealing with the query very effectively, preferring to search the whole $object$ relation for tuples satisfying the coordinate constraints and then test whether they had one of the required $repid$s, rather than use the $repid$ index field to narrow down the search first. The RTI Ingres query processor does not have this problem.
Table 9.1
Object ID retrieval times - Benchmark A1

<table>
<thead>
<tr>
<th>DBMS: Object relation</th>
<th>UIngres ISAM</th>
<th>RTI Ingres ISAM</th>
<th>B-tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>storage structure:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Retrieval no.:</td>
<td>1 2 3</td>
<td>1 2 3</td>
<td>1 2 3</td>
</tr>
<tr>
<td>Number of objects</td>
<td>31 7 2</td>
<td>31 7 2</td>
<td>31 7 2</td>
</tr>
<tr>
<td>retrieved:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index fields</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>repid, minx</td>
<td>0.4 0.4 0.4</td>
<td>0.2 0.2 0.2</td>
<td>0.2 0.2 0.2</td>
</tr>
<tr>
<td>repid, miny</td>
<td>0.6 0.5 0.5</td>
<td>0.2 0.2 0.2</td>
<td>0.2 0.2 0.2</td>
</tr>
<tr>
<td>repid, maxx</td>
<td>0.5 0.4 0.4</td>
<td>0.2 0.2 0.2</td>
<td>0.2 0.2 0.2</td>
</tr>
<tr>
<td>repid, maxy</td>
<td>0.5 0.8 0.4</td>
<td>0.2 0.2 0.2</td>
<td>0.2 0.2 0.2</td>
</tr>
<tr>
<td>repid, minx, miny</td>
<td>0.6 0.5 0.4</td>
<td>0.2 0.2 0.2</td>
<td>0.2 0.2 0.2</td>
</tr>
<tr>
<td>repid, minx, maxx</td>
<td>0.6 0.5 0.4</td>
<td>0.2 0.2 0.2</td>
<td>0.2 0.2 0.2</td>
</tr>
<tr>
<td>repid, minx, maxy</td>
<td>0.5 0.4 0.5</td>
<td>0.2 0.2 0.2</td>
<td>0.2 0.2 0.2</td>
</tr>
</tbody>
</table>

Table 9.2
Object ID retrieval times - Benchmark A2

<table>
<thead>
<tr>
<th>DBMS: Object relation</th>
<th>UIngres ISAM</th>
<th>RTI Ingres ISAM</th>
<th>B-tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>storage structure:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Retrieval no.:</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>Number of objects</td>
<td>54 9 75 25</td>
<td>54 9 75 25</td>
<td>54 9 75 25</td>
</tr>
<tr>
<td>retrieved:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index fields</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>repid, minx</td>
<td>2.9 2.2 1.4</td>
<td>0.6 0.3 0.7</td>
<td>0.3 0.2 0.3</td>
</tr>
<tr>
<td>repid, miny</td>
<td>2.9 2.3 1.4</td>
<td>0.6 0.2 0.3</td>
<td>0.3 0.2 0.3</td>
</tr>
<tr>
<td>repid, maxx</td>
<td>2.9 2.1 1.3</td>
<td>0.6 0.2 0.3</td>
<td>0.3 0.2 0.3</td>
</tr>
<tr>
<td>repid, maxy</td>
<td>2.9 2.3 1.3</td>
<td>0.6 0.3 1.4</td>
<td>1.4 0.2 0.3</td>
</tr>
<tr>
<td>repid, minx, miny</td>
<td>2.9 2.4 1.4</td>
<td>0.6 0.2 0.3</td>
<td>0.3 0.2 0.3</td>
</tr>
<tr>
<td>repid, minx, maxx</td>
<td>2.9 2.1 1.4</td>
<td>0.6 0.2 0.3</td>
<td>0.3 0.2 0.3</td>
</tr>
<tr>
<td>repid, minx, maxy</td>
<td>2.9 2.2 1.4</td>
<td>0.6 0.2 0.3</td>
<td>0.3 0.3 0.3</td>
</tr>
</tbody>
</table>

For Benchmark A3, the retrieval times for the lower viewing levels depended markedly on the choice of the first index coordinate, being considerably longer when \textit{maxx} or \textit{maxy} was used as an index field than \textit{minx} or \textit{miny}. Probably this was due to the fact that the regions being retrieved were located towards the southwest corner of the database domain. As can be seen from Figure 9.2, the size of the search region in data space varies with the value of the window coordinate that is being compared with the most significant index coordinate. The result is that small windows located near one edge of the domain in geographic space cause a large proportion of the \textit{object relation} to be searched, while those located near the opposite edge require only a small proportion to be searched.
As expected, nothing noticeable was gained by indexing on more than one coordinate of the bounding rectangle.

### Table 9.3
Object ID retrieval times - Benchmark A3

<table>
<thead>
<tr>
<th>DBMS: DBMS:</th>
<th>UlIngres</th>
<th>RTI Ingres</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieval no.:</td>
<td>ISAM</td>
<td>ISAM</td>
</tr>
<tr>
<td>Retrieval no.:</td>
<td>1 2 3 4 5</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>Number of objects retrieved:</td>
<td>89 3 281 39 9</td>
<td>89 3 281 39 9</td>
</tr>
<tr>
<td>Index fields</td>
<td></td>
<td></td>
</tr>
<tr>
<td>repid, minx</td>
<td>39.8 5.5 5.2 3.9 3.9</td>
<td>0.3 0.3 0.8 0.7 0.6</td>
</tr>
<tr>
<td>repid, miny</td>
<td>40.1 5.6 5.3 4.0 3.9</td>
<td>0.3 0.2 0.7 0.5 0.5</td>
</tr>
<tr>
<td>repid, maxx</td>
<td>39.3 5.5 5.6 3.9 3.9</td>
<td>0.3 0.2 1.8 2.3 2.4</td>
</tr>
<tr>
<td>repid, maxy</td>
<td>39.0 5.4 5.2 3.9 3.8</td>
<td>0.3 0.2 1.9 2.3 2.5</td>
</tr>
<tr>
<td>repid, minx, miny</td>
<td>39.2 5.5 5.2 3.9 4.0</td>
<td>0.3 0.2 0.8 0.6 0.5</td>
</tr>
<tr>
<td>repid, minx, maxy</td>
<td>39.4 5.5 5.3 3.9 3.8</td>
<td>0.3 0.2 0.7 0.6 0.5</td>
</tr>
<tr>
<td>repid, minx, maxy</td>
<td>39.2 5.5 5.2 3.9 4.0</td>
<td>0.3 0.2 0.7 0.6 0.5</td>
</tr>
</tbody>
</table>

### 9.4 The Point and Circle relations

Since the retrievals performed on the point and circle relations are very similar, the storage structure and indexing considerations are essentially the same for both. The only reasonable choices are:

1. Hash on objid
2. ISAM on objid
3. B-tree on objid

Table 9.4 lists the results of benchmark tests on the circle relation using the above three structures. The figures are times in seconds to execute Algorithm 1c above.

The retrieval times depended mostly on the number of tuples in the objtemp relation rather than the number of circle objects retrieved. This is understandable, since every objtemp tuple is being joined with the circle relation regardless of whether it actually represents a circle object.

For Benchmark A3, the ISAM structure was considerably faster in some cases than the hash table; the reason for this effect will be explained below.
### Table 9.4
Circle relation retrieval times (sec)

<table>
<thead>
<tr>
<th>DBMS:</th>
<th>UIngres</th>
<th>RTI Ingres</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle relation storage structure:</td>
<td>Hash</td>
<td>ISAM</td>
</tr>
<tr>
<td>Database Retrieval no.</td>
<td>No. of circles retrieved</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

### Table 9.5
Curve relation retrieval times (sec)

<table>
<thead>
<tr>
<th>DBMS:</th>
<th>UIngres</th>
<th>RTI Ingres</th>
</tr>
</thead>
<tbody>
<tr>
<td>curve relation storage structure:</td>
<td>Hash</td>
<td>ISAM</td>
</tr>
<tr>
<td>Database Retrieval no.</td>
<td>No. of curve points retrieved</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>69</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1275</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>400</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>150</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>33</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1450</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>872</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>123</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>27</td>
</tr>
</tbody>
</table>

### 9.5 The Curve relation

The curve relation has the additional requirement that the ordering of tuples belonging to the same curve object must be preserved. This can be achieved either by choosing a storage structure that imposes an ordering on the `seq` field, or by using a form of retrieve
statement that sorts tuples by the \textit{seq} field before returning them. Choices for the storage structure of the \textit{curve} relation are then as follows:

(1) Hash on \textit{objid} (with tuples sorted by \textit{objid}, \textit{seq} on retrieval)

(2) ISAM on \textit{objid}, \textit{seq}

(3) B-tree on \textit{objid}, \textit{seq}

Benchmark A results for these structures are summarised in Table 9.5. The figures are times to execute Algorithm 9.1d above.

At first sight it seems that there should be little difference between the hash table structure and the others, since no use is being made of any ordering on the \textit{objid} field. However, in the ULingres case, the hash table turned out to be extremely inefficient. It was also considerably less efficient in the RTI Ingres case, although the difference was less pronounced.

The reason for the large difference lay in the way that the data was originally organised, and the way that object IDs were assigned when the data was transferred to the relational database. The original data was divided geographically into tiles about 1km square, with a separate set of files for each tile. When the data was loaded into the relational database, each tile was processed in turn, with object IDs assigned sequentially. As a result, objects that were near each other geographically tended to be given object IDs that were nearby in numerical order.

Because of the way relations are stored in Ingres, use of a relation structure that imposes ordering, such as ISAM or B-tree, has the effect of clustering tuples with adjacent key values so that they are likely to occupy the same disk block. When applied to the \textit{objid} field, the result was a rough form of geographic clustering, so that geographically nearby objects were likely to be in the same disk block.

Using a hash table structure, on the other hand, randomly distributes the tuples, so that hashing on the \textit{objid} field had the effect of scattering objects over the disk without any regard for geographic proximity, so that many more disk accesses were required to retrieve them.

These results clearly demonstrate the need to geographically cluster objects on the disk to minimise disk accesses, and they also show one way in which such a clustering can be achieved. Some database management systems provide an explicit means to request clustering by a specified set of fields; where such a means is not provided, knowledge of
the way in which the system stores data in a relation can often be used to select a storage structure that has the required clustering characteristics.

9.6 The Text relations

Based on the above experience with the circle and curve relations, we can deduce a good choice of structure for the textblock and textline relations. The textblock relation is similar to the point and circle relations in that it contains one tuple for each text object indexed by the objid field. The textline relation is similar to the curve relation, containing a set of tuples for each object ordered by the seq field.

We can therefore expect to obtain the best results using an ISAM or B-tree structure, with textblock indexed on objid, and textline indexed on objid and seq. Benchmark results for these structures are listed in Table 9.6.

<table>
<thead>
<tr>
<th>Database</th>
<th>Retrieval number</th>
<th>No. of text lines retrieved</th>
<th>DBMS:</th>
<th>ULIngres</th>
<th>RTI Ingres</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Storage structure:</td>
<td>ISAM</td>
<td>ISAM</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ULIngres</td>
<td>0.2</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>ULIngres</td>
<td>0.2</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>ULIngres</td>
<td>0.2</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>24</td>
<td>ULIngres</td>
<td>0.9</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>ULIngres</td>
<td>0.5</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>ULIngres</td>
<td>0.9</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0</td>
<td>ULIngres</td>
<td>0.5</td>
<td>0.1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>24</td>
<td>RTI Ingres</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>RTI Ingres</td>
<td>0.7</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>46</td>
<td>RTI Ingres</td>
<td>4.8</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>RTI Ingres</td>
<td>1.4</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>2</td>
<td>RTI Ingres</td>
<td>0.8</td>
<td>0.2</td>
</tr>
</tbody>
</table>

9.6.1 Improving text retrieval performance

There are several ways in which the performance of Algorithm 1e might be improved:

(1) Instead of specifying the join of textblock and textline at retrieval time, we could use Quel to define a view as follows:
define view textview(textblock.all, textline.seq, textline.chars)
where textblock.objid = textline.objid

and then retrieve from textview as if it were a single relation. This would allow the query processor to perform some pre-processing of the query and possibly save time during execution.

(2) Combining the textblock and textline relations. The schema originally devised, while conceptually elegant, is not necessarily the most practical. Most of the text objects in a geographic database are likely to be names of roads, rivers and other landmarks, which are relatively short and not often composed of more than one line.

Analysis of the test databases revealed that they contained very few text strings of more than a dozen characters. That being the case, the whole text string can easily be stored in one tuple field, multiple lines separated by newline characters if need be. The textblock and textline relations can then be combined into a single text relation containing one tuple for each text object, eliminating a join at run time.

(3) Compressing the textline or text relation. Since most of the text strings in the test databases were so short, much of the space in the textline relation as originally specified was unused. Using one of the Ingres compressed storage structures ought not only to save space but also to speed retrieval, since (if objects are geographically clustered) the strings referenced by a given query will be spread over fewer disk blocks.

9.6.2 Experimental results

The results of benchmark tests using the above improvement methods are listed in Tables 9.7-9. Table 9.7 shows the results of using a textview relation as suggested above, Table 9.8 those of combining the textblock and textline relations into a single text relation, and Table 9.9 those of using an Ingres compressed storage structure for the text relation.

The results of all the text retrieval experiments are summarised together in Table 9.10 for comparison. The best performance was obtained using the single text relation, as might be expected, since it requires the least amount of work at retrieval time.

A surprising result was that the compressed ISAM and B-tree structures offered no improvement over their uncompressed counterparts, and in the ISAM case the compressed structure was considerably slower. It may be that factors other than the number of disk accesses were dominating the retrieval time, such as the time taken to translate tuples from compressed to uncompressed format and transmit them to the map browser.
9.7 Improving the algorithm

There are several improvements that can be made to the algorithm and database schema to improve performance.

9.7.1 Keeping track of object types

The algorithm as presented so far makes no explicit distinction between object types. It simply retrieves the object IDs required into \textit{objtemp} and then joins all of them with each of the four data relations, relying on the fact that, for a given object, only one of the four relations \textit{point}, \textit{curve}, \textit{circle} and \textit{text} will contain tuples with a matching object ID.

The disadvantage of this approach is that each tuple in \textit{objtemp} is subjected to three joining operations that can never succeed. One way of avoiding these unnecessary joins is to include the type of the object in \textit{objtemp} along with its object ID:

/* Algorithm 9.2a - Retrieve object IDs and types and place them in the temporary relation objtemp */

\begin{verbatim}
RetrieveObjects() {
    retrieve objtemp
    (objid = object.objid, type = object.type,
     layid = laytemp.layid)
    where object.bounds intersects the window
}
\end{verbatim}

Subsequent joining operations can then select object IDs of the appropriate type, for example,

/* Algorithm 9.2b - Retrieve point object data */

\begin{verbatim}
RetrievePoints() {
    retrieve (x = point.x, y = point.y)
    where objtemp.type = POINT and
    objtemp.objid = point.objid
    {
        draw the point;
    }
}
\end{verbatim}

Table 9.11 presents the results of benchmark tests using Algorithm 9.2, alongside the corresponding results using Algorithm 9.1 with the same relation storage structures.

Significant improvements were obtained in many cases, some examples of which are shown in bold in Table 9.11. In particular, the time taken to retrieve zero objects of a given type was now essentially constant, rather than depending on the total number of tuples in \textit{objtemp} as before, indicating that the new algorithm was working as intended.
Table 9.7
Text object retrieval times using a view to join the textblock and textline relations (sec)

<table>
<thead>
<tr>
<th>Database</th>
<th>Retrieval number</th>
<th>No. of text lines retrieved</th>
<th>ULIngres</th>
<th>RTI Ingres</th>
<th>RTI Ingres</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>ISAM</td>
<td>ISAM</td>
<td>ISAM</td>
<td>B-tree</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.2</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0.2</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0.2</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>24</td>
<td>4.4</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>0.4</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>4.9</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0</td>
<td>0.4</td>
<td>0.2</td>
<td>0.1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>24</td>
<td>0.7</td>
<td>0.3</td>
<td>0.2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0.6</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>3</td>
<td>46</td>
<td>3.3</td>
<td>0.6</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>0.9</td>
<td>0.2</td>
<td>0.2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0.6</td>
<td>0.2</td>
<td>0.2</td>
<td></td>
</tr>
</tbody>
</table>

Table 9.8
Text object retrieval times using a single text relation (sec)

<table>
<thead>
<tr>
<th>Database</th>
<th>Retrieval number</th>
<th>No. of text lines retrieved</th>
<th>ULIngres</th>
<th>RTI Ingres</th>
<th>RTI Ingres</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>ISAM</td>
<td>ISAM</td>
<td>ISAM</td>
<td>B-tree</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.2</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0.2</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0.2</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>24</td>
<td>0.7</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>0.2</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>0.8</td>
<td>0.2</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0</td>
<td>0.2</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>24</td>
<td>0.5</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0.3</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>3</td>
<td>46</td>
<td>0.7</td>
<td>0.4</td>
<td>0.4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>0.4</td>
<td>0.1</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0.3</td>
<td>0.1</td>
<td>0.1</td>
<td></td>
</tr>
</tbody>
</table>
Table 9.10
Comparison of text retrieval times using different text storage methods

<table>
<thead>
<tr>
<th>DBMS:</th>
<th>UIngres</th>
<th>RTI Ingres</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure: ISAM</td>
<td>ISAM</td>
<td>B-tree</td>
</tr>
<tr>
<td>DB</td>
<td>R#</td>
<td>#TL</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0.2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0.2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0.9</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0.7</td>
</tr>
<tr>
<td>3</td>
<td>46</td>
<td>4.8</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>1.4</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0.8</td>
</tr>
</tbody>
</table>
Experiments in Interactive Map Retrieval

Table 9.11
Comparison of retrieval times (sec)
A: Algorithm 9.1
B: Algorithm 9.2

<table>
<thead>
<tr>
<th>DBMS</th>
<th>DB Retr. no.</th>
<th>Point A</th>
<th>Point B</th>
<th>Curve A</th>
<th>Curve B</th>
<th>Circle A</th>
<th>Circle B</th>
<th>Text A</th>
<th>Text B</th>
</tr>
</thead>
<tbody>
<tr>
<td>UI</td>
<td>1</td>
<td>0.22</td>
<td>0.20</td>
<td>0.33</td>
<td>0.33</td>
<td>0.23</td>
<td>0.18</td>
<td>0.20</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.21</td>
<td>0.21</td>
<td>15</td>
<td>0.23</td>
<td>0.20</td>
<td>0.20</td>
<td>0.20</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.21</td>
<td>0.21</td>
<td>11</td>
<td>0.22</td>
<td>0.20</td>
<td>0.20</td>
<td>0.20</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.20</td>
<td>0.18</td>
<td>1275</td>
<td>0.32</td>
<td>1.22</td>
<td>0.00</td>
<td>0.59</td>
<td>0.17</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.17</td>
<td>0.16</td>
<td>400</td>
<td>0.49</td>
<td>0.49</td>
<td>0.00</td>
<td>0.21</td>
<td>0.16</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.19</td>
<td>0.18</td>
<td>150</td>
<td>0.57</td>
<td>0.55</td>
<td>0.00</td>
<td>0.74</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.17</td>
<td>0.17</td>
<td>33</td>
<td>0.33</td>
<td>0.27</td>
<td>10</td>
<td>0.34</td>
<td>0.21</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.23</td>
<td>0.21</td>
<td>1450</td>
<td>0.65</td>
<td>1.53</td>
<td>0.00</td>
<td>0.51</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.20</td>
<td>0.20</td>
<td>132</td>
<td>0.48</td>
<td>0.48</td>
<td>0.00</td>
<td>0.40</td>
<td>0.20</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.27</td>
<td>0.23</td>
<td>872</td>
<td>1.93</td>
<td>1.72</td>
<td>0.00</td>
<td>0.30</td>
<td>0.22</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.22</td>
<td>0.20</td>
<td>127</td>
<td>0.45</td>
<td>0.40</td>
<td>2</td>
<td>0.35</td>
<td>0.22</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.22</td>
<td>0.20</td>
<td>23</td>
<td>0.26</td>
<td>0.25</td>
<td>1</td>
<td>0.25</td>
<td>0.20</td>
</tr>
<tr>
<td>RTI</td>
<td>1</td>
<td>0.10</td>
<td>0.05</td>
<td>69</td>
<td>0.15</td>
<td>0.12</td>
<td>0</td>
<td>0.08</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.05</td>
<td>0.05</td>
<td>15</td>
<td>0.06</td>
<td>0.06</td>
<td>3</td>
<td>0.05</td>
<td>0.06</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.05</td>
<td>0.05</td>
<td>11</td>
<td>0.04</td>
<td>0.05</td>
<td>1</td>
<td>0.04</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.10</td>
<td>0.05</td>
<td>1275</td>
<td>0.53</td>
<td>0.49</td>
<td>0</td>
<td>0.13</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.05</td>
<td>0.05</td>
<td>400</td>
<td>0.19</td>
<td>0.19</td>
<td>0</td>
<td>0.06</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.11</td>
<td>0.05</td>
<td>150</td>
<td>0.26</td>
<td>0.25</td>
<td>0</td>
<td>0.16</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.07</td>
<td>0.05</td>
<td>33</td>
<td>0.12</td>
<td>0.09</td>
<td>10</td>
<td>0.09</td>
<td>0.07</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.12</td>
<td>0.05</td>
<td>1450</td>
<td>0.81</td>
<td>0.61</td>
<td>0</td>
<td>0.19</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.08</td>
<td>0.05</td>
<td>132</td>
<td>0.42</td>
<td>0.22</td>
<td>0</td>
<td>0.14</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.41</td>
<td>0.07</td>
<td>872</td>
<td>0.93</td>
<td>0.82</td>
<td>0</td>
<td>0.71</td>
<td>0.07</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.08</td>
<td>0.06</td>
<td>127</td>
<td>0.19</td>
<td>0.33</td>
<td>2</td>
<td>0.13</td>
<td>0.06</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.06</td>
<td>0.05</td>
<td>23</td>
<td>0.09</td>
<td>0.08</td>
<td>1</td>
<td>0.08</td>
<td>0.05</td>
</tr>
</tbody>
</table>

9.7.2 Optimising the schema

From the results presented it is clear that processing a Quel retrieve statement entails substantial overhead. For instance, the database 1 times for all operations were approximately the same, suggesting that the database was so small that the time to fetch the data was negligible compared to the overhead of processing the Quel statement. From the figures we can conclude that UIngres requires at least 0.2 seconds to process a Quel statement, and RTI Ingres about 0.05 seconds.

These overheads are sufficiently large compared to the retrieval times observed that they need to be taken into account in designing the retrieval algorithm and database structure.
Other things being equal, the smaller the number of Quel statements required to answer a window query, the better will be the performance.

The algorithm presented so far is hampered by the need to retrieve object data from four relations, *point*, *curve*, *circle* and *text*. Of these, the *point*, *curve* and *circle* relations contain very similar kinds of data – *x* and *y* coordinates in the *point* and *curve* relations, and radius information in the *curve* and *circle* relations.

Because of this similarity, it is easy to design a single relation that can be used to encode objects of all three types. There are many ways in which such a relation could be organised. For instance, observing that the definition we have been using for a *curve* includes circular arcs, we could regard points and circles as being special cases of curves and store them all in the existing *curve* relation. More storage space would be used, since two tuples would be required for each circle, and the *bulge* field of a one-point curve would be unused.

An alternative encoding could be devised to minimise storage space if required. For instance, we could adopt the convention that a one-point curve with a nonzero *bulge* field represents a circle centered at *x*, *y*. Or, if arcs were relatively rare compared with straight line segments, the *bulge* field could be omitted, an extra tuple being inserted for curved segments containing the bulge and suitably flagged.

Table 9.12
Total retrieval time for points, curves and circles (sec)
A: Separate *point*, *curve*, *circle* relations
B: Combined *curve* relation

<table>
<thead>
<tr>
<th>DB</th>
<th>Retr. no.</th>
<th>Number of Pts</th>
<th>Cur</th>
<th>Cir</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>69</td>
<td>0</td>
<td>0.71</td>
<td>0.38</td>
<td>0.22</td>
<td>0.12</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>13</td>
<td>4</td>
<td>0.63</td>
<td>0.26</td>
<td>0.16</td>
<td>0.08</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>9</td>
<td>1</td>
<td>0.61</td>
<td>0.26</td>
<td>0.15</td>
<td>0.06</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1275</td>
<td>0</td>
<td>1.56</td>
<td>0.50</td>
<td>0.59</td>
<td>0.47</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>400</td>
<td>0</td>
<td>0.86</td>
<td>0.62</td>
<td>0.28</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>150</td>
<td>0</td>
<td>0.89</td>
<td>0.73</td>
<td>0.35</td>
<td>0.26</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0</td>
<td>33</td>
<td>9</td>
<td>0.66</td>
<td>0.34</td>
<td>0.20</td>
<td>0.12</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1450</td>
<td>0</td>
<td>1.90</td>
<td>1.86</td>
<td>0.71</td>
<td>0.60</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>132</td>
<td>0</td>
<td>0.88</td>
<td>0.53</td>
<td>0.32</td>
<td>0.21</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>872</td>
<td>0</td>
<td>2.17</td>
<td>2.05</td>
<td>0.96</td>
<td>0.96</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0</td>
<td>123</td>
<td>2</td>
<td>0.82</td>
<td>0.43</td>
<td>0.45</td>
<td>0.21</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0</td>
<td>27</td>
<td>1</td>
<td>0.65</td>
<td>0.28</td>
<td>0.18</td>
<td>0.09</td>
</tr>
</tbody>
</table>
In any case, regardless of how the data is encoded in the relation, the reduction in the number of relations from four to two (curve and text) should result in appreciable reduction of overhead.

9.7.3 Experimental results

To assess the extent of the improvement achievable by combining the point, curve and circle relations, a single curve relation was constructed having the same fields as the original curve relation, but also containing point and circle objects. Point objects were represented by a single tuple with a zero bulge, and circle objects by a tuple with a nonzero bulge.
Benchmark results are listed in Table 9.12. Alongside are the sums of retrieval times from the original three data relations for comparison. A worthwhile improvement was observed in most cases.

9.8 Distribution of retrieval times

It was found in Section 9.3.1 that the time taken to search the Object relation appeared to depend on the position of the window along the coordinate being used as the most significant index key.

Benchmark B was used to obtain a better idea of the distribution of Object relation lookup times. Times to retrieve spatial objects from the other relations were also recorded. The resulting distributions of retrieval times are shown in Figures 9.3-9.5.
With RTI Ingres, Object ID retrieval times were obtained ranging from 0.1 to 3.5 seconds, most of them being distributed fairly uniformly from about 1 to 3 seconds. Most retrievals from the Curve and Text relations took less than 0.5 seconds each. These results suggest that spatial indexing is the main problem to be solved in improving interactive retrieval time.

9.9 Conclusions

It has been demonstrated that geographic data stored in a relational database can, once the identifiers of required objects are known, be retrieved in reasonable time provided that due attention is paid to choice of DBMS parameters such as storage formats and indexing.
Although the best choice of these parameters will depend on the particular DBMS being used, the following guidelines should be generally applicable to any relational database system:

(1) The query languages of many relational database systems are interpreted at runtime, resulting in significant overhead associated with each query language statement processed. In these cases it is important to minimise the number of statements issued, and keep those that are issued as simple as possible.

Minimising the number of relations that need to be referenced to answer a window query is one way of reducing the number of query statements. If there are several types of spatial object having similar kinds of data representing them, such as coordinates, it is worthwhile to consider encoding them all in a single relation.

The relations that need to be joined to answer a window query should also be examined; it may be possible to combine them into a single "prejoined" relation, as was done with the text relation here.

(2) If the DBMS provides views or some similar concept, prepackaging a complex query in this form may be helpful in reducing overhead.

(3) Clustering geographically related data on the disk is important, so that geographically nearby objects can be retrieved with a minimum of disk accesses. Even the rough form of clustering used here resulted in a considerable advantage. If the DBMS does not provide an explicit means to request clustering, it may be possible to achieve the desired effect by suitable choice of storage format and indexing.

The greatest difficulty in achieving sufficiently fast retrieval for interactive use is one of spatial indexing. Using the facilities provided by a typical relational DBMS such as Ingres, the most straightforward method of spatial indexing results in performance that is heavily dependent on the position of the search window and the overall size of the database. Even with the test databases used here, which are relatively small by GIS standards, average-case spatial index lookup times were on the long side as far as interactive response is concerned. It is clear that better spatial indexing techniques are required if adequate interactive performance is to be obtained.
Chapter 10

Quadtrees

The term *quadtree* refers to a large class of related data structures, for which there are many applications in the field of spatial data handling. This chapter discusses quadtrees in general, some specific quadtree-based data structures in particular, and describes an experiment carried out using one of these data structures.

Section 10.1 introduces the main principles of the quadtree and describes some of the ways in which it can be used to spatially organise or index geographic data.

One such method, due to Abel and Smith (1983, 1984, 1985), is particularly well suited to implementation in a relational database environment, and is described in Section 10.2.

Section 10.3 presents the results of an experiment to assess the effectiveness of Abel and Smith's method in the context of the map browsing system and database described in Chapters 8 and 9.

10.1 Quadtree data structures

Most methods of organising spatial objects are based on a systematic subdivision of space. In the quadtree, a rectangular region is recursively divided into four subregions, or quadrants, until each subregion meets some criterion. The subdivision process can be expressed in the form of a tree, with a node corresponding to each region, and each node having up to four descendants.

A great many data structures and algorithms exist based on quadtree subdivision; an extensive survey can be found in Samet (1984, 1988, 1990). Described below are a few representatives that are particularly relevant to the storage and indexing of vector geographic data.
10.1.1 The Point Quadtree

Finkel and Bentley (1974) describe a quadtree-based data structure for organising a collection of points in two dimensions; Samet (1984) refers to this structure as a point quadtree.

Each node of the tree stores one point, and the x and y coordinates of the point are used to divide the region corresponding to the node into four quadrants. Points lying in each quadrant are stored in the corresponding subtree. Figure 10.1 shows a collection of points and a corresponding point quadtree.

Finkel and Bentley present algorithms for inserting and deleting nodes, keeping the tree balanced, and searching for either a single point or all points lying within a search window.

10.1.2 The PR Quadtree

An alternative to dividing up the data, as the point quadtree does, is to divide up the coordinate space in which the data is embedded. In the PR quadtree, points are stored only in the leaf nodes, and each node may hold no more than one point. A quadrant containing more than one point is recursively divided into four equal-sized subquadrants until each quadrant is either empty or contains one point.

Figure 10.2 shows the PR quadtree corresponding to Figure 10.1.

The PR quadtree has the advantage over the point quadtree that the shape of the tree is independent of the order of insertion of points, and balancing is not an issue.
However, closely-spaced points can result in very deep subdivision, whereas the corresponding point quadtree would be much shallower.

If the maximum capacity of a leaf node is increased from 1, the PR quadtree can be used to partition points into “buckets” of a given maximum capacity, which could correspond to disk blocks or other suitable units of secondary storage.

10.1.3 Representing lines and areas using quadtrees

Quadtree structures for storing line and area objects can be classified into raster and vector techniques. One way of dealing with line and area objects using quadtrees is to approximate them using rasters. Quadtrees can be used to compactly represent binary rasters (rasters made up of 2-valued cells) and efficiently carry out certain operations on them (Samet 1984, 1985b, 1990; Ang and Samet 1991; Oliver and Wiseman 1983a,b; Samet, Shaffer, Nelson, Huang, Fujimura and Rosenfeld 1986).

Quadtrees can also be used to store vector representations of lines and polygons. The edge quadtree of Schneier (1981) subdivides a region containing a vector object until each quadrant contains a curve that can be approximated with a single line segment. A serious disadvantage is that vertices where two or more edges meet cannot be represented exactly.

The PM quadtree (Samet 1985a) is a development of the PR quadtree and edge quadtree that allows vertices of any degree to be handled. In the PM quadtree, quadrants are subdivided until each quadrant either contains at most one vertex, where any number of edges may meet.
Samet discusses three variations of the PM quadtree, and analyses them with respect to insertion, deletion, finding the region containing a point, and overlaying two sets of polygons.

The *PMR quadtree* (Nelson and Samet 1986) is a variant of the PM quadtree for secondary storage in which each leaf node is a bucket with a capacity for a certain number of line segments. So that vertices with an arbitrary number of incident edges may be handled, a probabilistic splitting rule is used.

The bucket capacity is treated not as a fixed limit, but as a *splitting threshold*. When insertion of a line segment causes the splitting threshold to be exceeded, the node is split (once) into four quadrants. Samet (1990) claims empirical evidence that, except in unusual cases, the bucket occupancy tends not to exceed the splitting threshold by much.

10.1.4 Quadtree-based spatial indexes

A quadtree can be used as a spatial index to a collection of objects by approximating each object with a bounding rectangle, and using a quadtree to organise the resulting set of rectangles together with pointers to their associated objects.

In the *MX-CIF quadtree* (Samet 1988) each rectangle is associated with the node corresponding to its smallest enclosing quadtree block (Figure 10.3). Within each node, the rectangles may be kept as an unstructured collection, or may be further organised in some way. Samet describes a variation in which the rectangles are partitioned into two sets according to whether they cross the east-west or north-
south bisector of the quadrant, and each set is organised as a binary tree in a one-dimensional analogue of the MX-CIF quadtree.

A problem with the MX-CIF quadtree and similar approaches is that, depending on the placement of a rectangle in the coordinate space, its minimum enclosing quadrant may be much larger than the rectangle itself. For a given search window, a quadrant intersecting the window may contain many rectangles that do not intersect the window, leading to inefficient searches.

This problem can be addressed by associating each rectangle with a collection of nodes whose quadrants more closely approximate the size and shape of the rectangle. Smith and Abel (1985) suggest the use of up to four quadrants per rectangle, although other numbers could be used. Samet uses the term expanded MX-CIF quadtree to refer to an MX-CIF quadtree in which rectangles are associated with multiple nodes.

A spatial index suitable for secondary storage can be constructed by regarding each leaf node as a bucket containing references to objects which intersect the associated quadrant; such a scheme is used in the GeoVision system (Palimaka, Halustchak and Walker 1986). If objects can overlap, a probabilistic splitting scheme similar to that of the PMR quadtree is necessary.

10.1.5 Other methods related to the quadtree

The bintree (Knowlton 1980, Tamminen 1984, Sammet 1990) is a binary tree which subdivides regions into two equal parts, with the direction of subdivision alternating between north-south and east-west at each level of the tree. Most of the quadtree-based data structures which use regular subdivision have a bintree equivalent derived in the obvious way.

Structures such as the point quadtree, which use the data elements stored in the tree to determine subdivision points, also have binary tree equivalents. In this case the resulting structure is known as a k-d tree, and will be discussed further in Chapter 11.

10.2 Locational Keys

The data structures described thus far have been designed on the assumption that they will be explicitly represented in the form of nodes and pointers, and accessed using conventional tree traversal algorithms expressed in a procedural language.
Relational database systems, however, are not well suited to the representation of hierarchical structures such as trees, and are usually accessed by means of a more declarative style of language which does not lend itself to expressing tree traversal algorithms.

Abel and Smith (1983, 1984; Smith and Abel 1985) describe a method of implementing a spatial index using a conventional one-dimensional index structure, such as a B-tree, for which most relational database systems provide direct and efficient support. As a consequence, their method can be used to implement a spatial index that can be searched relatively efficiently using a relational query language. A version of their method is used in the GEOVIEW system (Waugh and Healey 1987).

Abel and Smith use a quadtree subdivision of space to find the smallest quadtree cell that encloses each object. A numerical key is then assigned to each object according to its minimum quadtree cell.

The cells are numbered according to the Morton ordering (Morton 1966, Samet 1990). The key can be represented in a variety of ways. One way is to use a base-5 number of \( m \) digits, where \( m \) is an arbitrary maximum depth of subdivision. The four subcells of a cell are represented by the digits 1 to 4. From left to right, successive digits trace a path from the root of the tree down to the cell concerned. Remaining unused digit positions are filled with zeroes. Figure 10.4a shows an example of this form of cell numbering.

A more compact representation uses a pair of integers \((c, l)\), where \( c \) is a base-4 cell number, and \( l \) is a level number to indicate the depth of subdivision of the cell, as shown in Figure 10.4b.

The Morton ordering has the property that, when the keys are sorted into numerical order, the subcells of a cell immediately follow it in the sequence. As a result, cells that are nearby in space are often also nearby in the key sequence (although not always).

To answer a window query using the locational key index, the key of the smallest quadtree cell enclosing the window is found, and then the index is searched for all cells that are either contain or are contained in the window cell.

If \((k_w, l_w)\) is the key of the smallest cell enclosing the window, then due to the ordering property of the keys, the cell number \( c \) of any enclosed cell satisfies
Figure 10.4
Morton numbering of quadtree cells
(a) Base-5 encoding (b) Base-4 encoding plus level number

\[ c_w \leq c < h(c_w, l_w) \]

where, for a maximum decomposition level of \( m \),

\[ h(c, l) = c + 4^{m-l}w \]
The keys of cells that enclose the window cell can be found by successively truncating digits from the right of $c_w$.

Depending on the placement of an object, the minimum cell enclosing the object may be much larger than the object's bounding rectangle. To overcome this problem, Abel and Smith suggest using up to four quadtree cells to cover the object, improving the selectivity of the search at the expense of extra storage space for the spatial index.

Abel and Smith present an algorithm for efficiently carrying out the window search when a B-tree is used to hold the keys. They also present a modified version of the algorithm for use when the spatial index is held in a database system that does not provide access to the internal structure of the B-tree, such as in a relational database system.

10.3 An experiment using locational keys

An experiment was performed using the version of Abel and Smith's method described by Smith and Abel in their 1985 paper. In that version, the locational key of a quadcell is a pair of integers $(c, l)$ where $c$ is a base-4 number defining the cell's location and $l$ is the level of decomposition of the cell.

10.3.1 Implementation

The spatial index was implemented as a relation

$$\text{morton}(\text{repid, objid, c, l, minx, miny, maxx, maxy})$$

relating an object ID, the key of its smallest enclosing quadcell, and its minimum bounding rectangle. The latter was included to facilitate testing the bounds of candidate objects against the window without having to reference another relation.

Using this relation, the window query can be carried out as follows. The key $(k_w, l_w)$ of the smallest cell enclosing the window is calculated. Then, the object IDs of all objects intersecting the window are found in two phases.

The first phase finds all objects whose cells are completely enclosed by the window cell; the bounding rectangles of these objects are then further tested against the search window. The first phase is achieved by the following query:
range of $m$ is morton
append objtemp($m$.objid) where
  laytemp.repid = $m$.repid and
  $m$.c >= $cw$ and $m$.c < $h(cw, lw)$ and
  $m$.minx <= $max_x$ and $m$.miny <= $max_y$ and
  $m$.maxx >= $min_x$ and $m$.maxy >= $min_y$

The second phase finds objects whose cells enclose the window cell, in two subphases. First, the keys of the enclosing cells are calculated by successively truncating digits from $c_w$ and stored in a temporary relation $keytemp$($repid$, $c$, $l$). Conceptually this can be expressed as:

```plaintext
c_t = c_w; l_t = l_w;
while (l > 0) {
  truncate_key($c_t$, $l_t$);
  for (each representation $r$ of interest) {
    append keytemp(repid=$r$, $c$=$c_t$, $l$=$l_t$);
  }
}
```

In Ingres, this operation can be implemented much more efficiently by writing the tuples to a temporary file and then transferring them to the $keytemp$ relation with the Quel copy statement:

```plaintext
open temporary file $f$;
while (l > 0) {
  truncate_key($c$, $l$);
  for (each representation $r$ of interest) {
    write $r$, $c$, $l$ to $f$;
  }
}
close file $f$;
copy keytemp() from $f$;
```

Finally, the $keytemp$ relation is joined with the $morton$ relation:

```plaintext
append objtemp($m$.objid) where
  keytemp.repid = $m$.repid and
  $m$.c = keytemp.c and $m$.l = keytemp.l and
  $m$.minx <= $max_x$ and $m$.miny <= $max_y$ and
  $m$.maxx >= $min_x$ and $m$.maxy >= $min_y$
```

Experiments were performed using both the single and multiple key variations of the Abel and Smith method. Separate times were recorded for the three phases of the spatial index lookup listed in Table 10.1.

### 10.3.2 Single key per object

For the single-key case, four choices of storage structure were evaluated, listed in Table 10.2a. The times obtained are shown in Table 10.2b.
Table 10.1
Abel and Smith spatial index lookup phases

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Searching the cells covered by the window cell</td>
</tr>
<tr>
<td>IIa</td>
<td>Calculating the keys of the cells covering the window cell and placing them in keytemp</td>
</tr>
<tr>
<td>IIb</td>
<td>Searching the cells covering the window cell</td>
</tr>
</tbody>
</table>

Table 10.2
Abel and Smith, one key per object

(a) Storage structures and indexes

<table>
<thead>
<tr>
<th>Structure of motion relation</th>
<th>Secondary index</th>
</tr>
</thead>
<tbody>
<tr>
<td>A ISAM on repid, key</td>
<td>none</td>
</tr>
<tr>
<td>B ISAM on repid, key, level</td>
<td>none</td>
</tr>
<tr>
<td>C ISAM on repid, key</td>
<td>ISAM on repid, key, level</td>
</tr>
<tr>
<td>D ISAM on repid, key, level</td>
<td>ISAM on repid, key</td>
</tr>
</tbody>
</table>

(b) Benchmark A3 spatial index lookup times (RTI, sec)

<table>
<thead>
<tr>
<th>Storage structure</th>
<th>Retrieval no.</th>
<th>No. of objects</th>
<th>Phase</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>89</td>
<td>0.3</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>51</td>
<td>0.3</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>281</td>
<td>0.8</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>39</td>
<td>0.4</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>9</td>
<td>0.3</td>
<td>1.2</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>89</td>
<td>0.3</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>51</td>
<td>0.3</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>281</td>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>39</td>
<td>0.4</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>9</td>
<td>0.3</td>
<td>1.1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>89</td>
<td>0.3</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>51</td>
<td>0.3</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>281</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>39</td>
<td>0.4</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>9</td>
<td>0.3</td>
<td>1.1</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>89</td>
<td>0.3</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>51</td>
<td>0.3</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>281</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>39</td>
<td>0.4</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>9</td>
<td>0.3</td>
<td>1.1</td>
</tr>
</tbody>
</table>
10.3.3 Multiple keys per object

Approximating each object by a single quadcell can lead to some objects being represented by cells much larger than the object itself, with a resulting degradation in the performance of the windowing algorithm due to false hits. To overcome this problem, Abel and Smith suggest approximating each object with up to four quadcells, so that each object ID is associated with up to four locational keys.

Abel and Smith present a relational version of their multiple-key algorithm containing the following steps:

1. Calculate the keys of up to 4 cells covering the window rectangle.
2. Calculate the \((c, l)\) pairs of the cells enclosing each of the window cells found in step 1, and place them in a temporary relation.
3. Retrieve the IDs of objects having keys equal to the keys calculated in step 2, and place them in the result relation.
4. Calculate the cell number ranges \(c_{l1}..c_{h1}, ..., c_{l4}..c_{h4}\) of the cells covered by each of the window cells from step 1, and place them in a temporary relation.
5. Retrieve the IDs of objects having keys within one of the ranges calculated in step 4, and place them in the result relation.
6. Delete from the result relation any objects whose minimum bounding rectangle does not intersect the actual window rectangle.

Duplicate object IDs retrieved during steps 3 and 5 are eliminated by arranging for the result relation to be indexed on the object ID with duplicates disallowed.

The above algorithm was implemented with the following modification: instead of deleting objects which fail the rectangle intersection test in the last step, the rectangle comparison was incorporated into the retrievals in steps 3 and 5, so that step 6 was eliminated.

Also, a \textit{repid} field was incorporated into the temporary relations created in steps 2 and 4, and these steps were applied to each representation of interest.

The resulting algorithm can be expressed as:
range of m is morton
range of ktl is keytempl
range of kt2 is keytemp2

/* Phase Ia */
for each representation r of interest
for each key (ct, lt) covering one of the window cells
append keytempl(repid=r, c=ct, l=lt)

/* Phase Ib */
append objtemp(m.objid) where
m.repid = ktl.repid and
m.c = ktl.c and m.l = ktl.l and
m.bounds intersects the window rectangle

/* Phase IIa */
for each representation r of interest
for each window cell key (c, l)
append keytemp2(repid=r, clo=ct, chi=h(c, l))

/* Phase IIb */
append objtemp(m.objid) where
m.repid = kt2.repid and
m.c >= kt2.clo and m.c <= kt2.chi and
m.bounds intersects the window rectangle

An ISAM structure was used for the morton relation, indexed on repid, key and level. The results are listed in Table 10.3.

<table>
<thead>
<tr>
<th>Retrieval no.</th>
<th>No. of objects</th>
<th>Phase</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Ia</td>
<td>Ib</td>
</tr>
<tr>
<td>1</td>
<td>89</td>
<td>0.3</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>51</td>
<td>0.5</td>
<td>0.1</td>
</tr>
<tr>
<td>3</td>
<td>281</td>
<td>0.5</td>
<td>0.1</td>
</tr>
<tr>
<td>4</td>
<td>39</td>
<td>1.5</td>
<td>0.7</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>1.7</td>
<td>0.8</td>
</tr>
</tbody>
</table>

Surprisingly, extremely long times were obtained for some of the retrievals. It was found that executing a query of the form

\[ \text{retrieve (...) where } a.x \geq b.y1 \text{ and } a.x \leq b.y2 \]

in RTI Ingres took an unreasonably long time, even with a very small number of tuples in b and suitable indexing of a. By contrast, an equivalent query of the form
retrieve (...) where (a.x >= y1 and a.x <= y2)
or (a.x >= y3 and a.x <= y4)
or (a.x >= y5 and a.x <= y6) ...

was handled much more efficiently. Considering this, it might be possible to improve performance under RTI Ingres by dynamically constructing query statements with explicit comparisons between keys in the morton relation and the required key values and ranges. This would have the added benefit of removing the query statements for establishing the keytemp relations and their associated overhead.

10.4 Conclusions

Under the conditions investigated, the performance of Abel and Smith's method, in the single-key case, seemed to lie somewhere between the best and worst cases obtained using the one-dimensional indexing of Chapter 9.

However, retrieval times using the one-dimensional index grow linearly with the size of the database. Abel and Smith's method, being tree-based, can be expected to show a logarithmic increase. So, for a sufficiently large database, an Abel and Smith index should be more efficient than a one-dimensional index.

An experiment to evaluate the multiple-key variant of Abel and Smith's method yielded inconclusive results because of the idiosyncratic behaviour of RTI Ingres. This suggests that the success of the method may be quite sensitive to characteristics of the particular database system being used and the tuning of the implementation.
Chapter 11

K-D Trees

This chapter introduces the \textit{k-d tree} and describes one of the ways in which it can be used to provide a spatial index. It also describes an experiment in which a k-d tree was explicitly represented in a relational database and searched by issuing a sequence of query statements to the DBMS. The search was carried out using a breadth-first technique which uses a number of query statements proportional to the depth of the search.

11.1 The k-d tree

The \textit{k-d tree} was first proposed by Bentley (1975) as a means of organising objects indexed by a \(k\)-dimensional key. Each object stored in the tree is associated with \(k\) key attributes called \textit{discriminators}.

The tree is a binary tree in which each node holds one data object. At each node, one of the discriminator values \(d\) is used to make a branching decision; the left subtree holds objects with discriminators less than \(d\), and the right subtree those with discriminators greater than \(d\). As the tree is descended, each discriminator is used in turn, so that at level \(i\) of the tree the branching decision is made according to discriminator \(i \mod k\).

Where the objects are points in 2-dimensional space, the resulting structure is termed a \textit{2-d tree}. Figure 11.1 shows an example of a 2-d tree.

The 2-d tree can be regarded as the data-organising equivalent of the bintree; that is, the 2-d tree bears the same relationship to the bintree as the point quadtree does to the region quadtree. As with the bintree and quadtree, range searches can be carried out very efficiently on the 2-d tree.

In its simplest form, the k-d tree is a structure for organising points, and is not directly suitable for handling objects such as lines and areas. However, there are ways in which point methods can be adapted to handle these objects. One way is to
choose a "representative point" for each object, such as its centroid, and use a point method to index these points. Matsuyama et al (1984) use this approach.

Another way is to map objects in a 2-dimensional space into points in a higher dimensional space. For instance, four parameters defining an object's minimum bounding rectangle can be regarded as a point in a 4-dimensional space, and a point method used to index this space (Rosenberg 1985, Samet 1990, Ooi 1990).

The four parameters used to represent a rectangle can be chosen in a variety of ways. A common choice is the coordinates of the sides; an alternative is the centroid plus the horizontal and vertical extents (Samet 1990).

Rosenberg (1985) describes in detail the use of a 4-dimensional k-d tree to organise a collection of rectangles. He presents algorithms for building a k-d tree from a set of rectangles, and for finding all the rectangles which intersect a rectangular window.

11.2 The experiment

A k-d tree spatial index along the lines of Rosenberg's was implemented using two relations

kdroot(repid, objid)

kdnode(objid, minx, miny, maxx, maxy, loson, hison)
In this implementation, a separate tree is associated with each representation in the database, with $kdroot$ connecting a representation to the root node of its associated tree.

Each node has a tuple in $kdnode$, identified by a unique node ID. Since there is one node per object, the object ID may be used as the node ID. The tuple also contains the minimum bounding rectangle of the object, and the node IDs of its two sons.

The tree could be searched by executing the usual search algorithm, using a query statement to retrieve each node as required, but the resulting overhead would be prohibitive.

Instead, the tree may be searched in $O(\log n)$ query statements, where $n$ is the number of nodes in the tree, using the following algorithm. Let there be a temporary relation

$$kdtemp(depth, objid, minx, miny, maxx, maxy, loson, hison)$$

(1) Retrieve into $kdtemp$ the root nodes of the trees of all representations to be searched:

retrieved $kdtemp(depth = 0, kdnode.all)$ where $kdnode.objid = kdroot.objid$ and $kdroot.repid = a$ representation of interest

(2) For $i = 0, 1, 2, ...,$, construct a query statement that, given the nodes visited at level $i$ of the search, retrieves the nodes to be visited at level $i + 1$. Repeat until there are no more nodes to be visited.

```
i = 0;
while (any(kdtemp where kdtemp.depth = i)) {
    construct the predicates loson_condition and hison_condition;
    append kdtemp(depth = i+1, kdnode.all) where
        kdtemp.depth = i and
        (kdnode.objid = kdtemp.loson and loson_condition)
    or
        (kdnode.objid = kdtemp.hison and hison_condition)
}
```

The predicates $loson\_condition$ and $hison\_condition$ are constructed dynamically at each tree level $i$ according to the discriminator for that level. Each predicate either is empty or compares the node’s discriminator value with one of the rectangle coordinates.
(3) Some of the nodes visited may lie outside the window, although it is necessary to visit them so that their subtrees may be searched. Make a final pass over \textit{kdtemp} is to find the objects which intersect the window rectangle:

\begin{verbatim}
retrieve objtemp(objid = kdtemp.objid) where
kdtemp.bounds intersects the window
\end{verbatim}

\subsection{11.2.1 Pruning the search}

Rosenberg describes a method of reducing the number of unwanted nodes visited during a search by storing additional information in each node about the minimum and maximum values of keys in each subtree.

The \textit{kdnode} relation was extended to support this technique by adding three new fields \textit{lominbound}, \textit{himaxbound} and \textit{otherbound}. In the retrieval algorithm, additional tests were added to the \textit{loson_condition} and \textit{hison_condition} to test the window rectangle against these bounds. For further details see Rosenberg (1985).

\subsection{11.2.2 Tuning the algorithm}

During experimentation it was found that greater efficiency could be obtained from both UIngres and RTI Ingres by breaking the complex query in the main loop into three simpler queries, and using an additional temporary relation

\begin{verbatim}
kdtemp2(objid)
\end{verbatim}

The modified loop body consisted of:

\begin{verbatim}
/* Phase I */
retrieve kdtemp2(objid = kdtemp.loson) where
kdtemp.depth = i and loson_condition

/* Phase II */
retrieve kdtemp2(objid = kdtemp.hison) where
kdtemp.depth = i and hison_condition

/* Phase III */
append kdtemp(depth = i+1, kdnodes.all) where
kdtemp2.objid = kdnodes.objid
\end{verbatim}

\subsection{11.2.3 Experimental results}

Tables 11.1 and 11.2 list the results obtained from Benchmark A, using the three-phase retrieval algorithm with pruning described above.
### Table 11.1
Breadth-first K-D tree search
Spatial index lookup time (sec)

<table>
<thead>
<tr>
<th>Database</th>
<th>Retrieval no.</th>
<th>Search depth</th>
<th>No. of nodes visited</th>
<th>No. of objects found</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>31</td>
<td>31</td>
<td>4.0</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>23</td>
<td>7</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>20</td>
<td>2</td>
<td>4.6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Database</th>
<th>Retrieval no.</th>
<th>Search depth</th>
<th>No. of nodes visited</th>
<th>No. of objects found</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
<td>54</td>
<td>54</td>
<td>4.1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>31</td>
<td>9</td>
<td>4.9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>81</td>
<td>75</td>
<td>6.1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>94</td>
<td>25</td>
<td>7.0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Database</th>
<th>Retrieval no.</th>
<th>Search depth</th>
<th>No. of nodes visited</th>
<th>No. of objects found</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>6</td>
<td>89</td>
<td>89</td>
<td>5.7</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>23</td>
<td>3</td>
<td>6.5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>315</td>
<td>281</td>
<td>13.2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>229</td>
<td>39</td>
<td>13.5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>149</td>
<td>9</td>
<td>12.9</td>
<td></td>
</tr>
</tbody>
</table>

### Table 11.2
Breadth-first K-D tree search
Spatial index lookup time (sec)

<table>
<thead>
<tr>
<th>Database</th>
<th>Retrieval no.</th>
<th>Search depth</th>
<th>No. of nodes visited</th>
<th>No. of objects found</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>31</td>
<td>31</td>
<td>6.4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>23</td>
<td>7</td>
<td>10.0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>20</td>
<td>2</td>
<td>9.1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Database</th>
<th>Retrieval no.</th>
<th>Search depth</th>
<th>No. of nodes visited</th>
<th>No. of objects found</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
<td>54</td>
<td>54</td>
<td>11.6</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>31</td>
<td>9</td>
<td>6.5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>81</td>
<td>75</td>
<td>13.1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>94</td>
<td>25</td>
<td>13.0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Database</th>
<th>Retrieval no.</th>
<th>Search depth</th>
<th>No. of nodes visited</th>
<th>No. of objects found</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>6</td>
<td>89</td>
<td>89</td>
<td>11.0</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>23</td>
<td>3</td>
<td>11.0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>315</td>
<td>281</td>
<td>36.8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>229</td>
<td>39</td>
<td>28.9</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>149</td>
<td>9</td>
<td>21.3</td>
<td></td>
</tr>
</tbody>
</table>
As expected, the times are approximately proportional to the depth of the search, and are largely independent of the number of objects retrieved. They are also mostly independent of the size of the database, except so far as this affects the search depths.

11.3 Conclusions

The technique investigated in this chapter was successful in terms of looking up a spatial index in $O(\log n)$ time, where $n$ is the number of objects in the index. However, for the purposes of interactive browsing, the technique was found to be unsuitable, because of the overhead associated with the large number of complex query statements required.
Chapter 12

Paged Binary Trees

The tree searching technique used in Chapter 11 was found to be inefficient because of the large number of complex query statements issued to the DBMS. This chapter explores a technique which uses simpler query statements, and performs more processing in the retrieval engine of the map browser, to reduce the overhead incurred by going through the relational DBMS.

Section 12.1 reviews tree structures designed to be kept in secondary storage. Section 12.2 discusses a particular class of these structures, paged binary trees. Section 12.3 applies the paged binary tree technique to the k-d tree to yield the paged k-d tree, analyses it theoretically, and presents the results of an experiment using a paged version of the k-d tree described in Chapter 11.

12.1 Tree structures for secondary storage

Most methods of managing tree structures in secondary storage involve grouping the data items into buckets or pages which can be made to correspond to the unit of transfer. A well-known example is the B-tree and its variants (Bayer and McCreight 1972, Comer 1979) in which each bucket is a node in the tree containing some number of key values or data items.

12.1.1 B-tree related methods

Many spatial data structures have variants that are related to the B-tree.

The k-d-B tree (Robinson 1981; Beckley, Evens and Raman 1985) combines features of the k-d tree and the B-tree. There are two types of nodes in the tree, region pages and data pages, with data pages occurring only as leaf nodes. Region pages contain discriminator values which divide the space covered by the node into subspaces, together with pointers to nodes covering those subspaces. Data pages contain data items, or pointers to data items, that lie within the relevant subspace. Discriminators are used cyclically as in the k-d tree. As with the B-tree, insertions and deletions can be performed so as to keep the tree balanced.
Scheuermann and Ouksel (1982) proposed a complicated structure called a *multidimensional B-tree* (MDBT) consisting of a tree of B-trees, with one level in the top-level tree for each discriminating attribute. They compare the structure with the ordinary k-d tree and conclude that it has advantages over the k-d tree with respect to dynamic maintenance. They do not discuss the k-d-B tree, and it is unclear whether the extra complexity of the MDBT provides any advantage over the k-d-B tree.

Orenstein (1982) discusses a *k-d trie*, which is very similar to the k-d tree except that regions are always split into two subregions of equal size, rather than using a key value of one of the data items to split the region. In this respect the k-d trie can be regarded as a generalisation of the bintree to \( k \) dimensions.

Orenstein describes two versions of the k-d trie. In the "pure" version, regions are split until each contains a single data item. In the "hybrid" version, splitting terminates when a region contains less than a certain number of items, typically that which will fit in a page of secondary storage.

The R-tree (Guttman 1984, Roussopoulos and Leifker 1985) is a generalisation of the B-tree to two dimensions. The pages of an R-tree contain records, each of which comprises a rectangle and a pointer. A leaf page record contains a pointer to a spatial object together with its bounding rectangle; a nonleaf page record contains a pointer to another page and a rectangle bounding the rectangles of all the records in that page. The R-tree can be efficiently searched for objects intersecting a query rectangle, and can be dynamically updated using algorithms similar to those for a B-tree.

### 12.2 Paged binary trees

Knuth (1973) describes how a binary tree can be adapted for secondary storage by grouping nodes of subtrees into pages, forming a *paged binary tree*. Figure 12.1 shows an example of a paged binary tree with a page size of 7 nodes. It is evident that a depth-first traversal of part of a tree organised in this way will be localised within a few pages.

Cesarini and Soda (1982) compare paged binary trees with B-trees and conclude that the two structures have very similar properties. This can be seen by regarding each page of the paged binary tree as a B-tree node which is organised internally as a binary tree (rather than an array as in a conventional B-tree).

Because of this relationship, all of the usual insertion, deletion and searching algorithms used with B-trees can also be applied to paged binary trees. An added
advantage is that the paged binary tree can be read into memory and manipulated
there as an ordinary binary tree, simplifying applications where the tree is sometimes
in memory and sometimes not.

For the present purposes, it is assumed that the database will not change
frequently, so the ability to update the index dynamically is of secondary importance.
Nevertheless, use of a paged binary tree would allow the index to be updated
dynamically if need be.

12.3 Paged k-d tree

This section describes an experiment in which a paged version of the k-d tree used
in Chapter 11 was implemented, and searched using a conventional depth-first tree-
traversal algorithm written in C, and issuing quel statements to the relational DBMS
to retrieve pages of the tree.

12.3.1 Data structure

The k-d tree of Chapter 11 was adapted as follows. Nodes were grouped into
pages using a bottom-up grouping algorithm which ensured that each full page
contained a balanced binary tree forming a subtree of the original tree, and that as
many pages as possible were full.

Each page was assigned a unique page number, and each node a node number
within its page. The \textit{kdnode} relation was extended with a \textit{nodeid} field containing a
combination of the page and node number, and the \textit{loson} and \textit{hison} fields made to refer
to child nodes by their node IDs (rather than object IDs as before).
An ISAM structure indexed on nodeid was used for the kdnode relation, serving two purposes: to enable all the nodes belonging to a given page to be retrieved efficiently with a single retrieve statement, and to ensure that the nodes of a page were clustered together on the disk.

12.3.2 Retrieval algorithm

A conventional depth-first search algorithm was implemented in C, based on Rosenberg's code (1985), together with a page cache in virtual memory. The first time that the search algorithm references a node, the page containing that node is retrieved from the database and stored in the cache. Subsequent references to any node in the same page can then be satisfied without further database access.

So that reproducible results could be obtained from the experiment, the page cache was cleared prior to the processing of each window query. In practice, the contents of the cache would be retained between queries to speed up further accesses to the same regions of the spatial index.

12.3.3 Choice of page size

The optimum page size is determined by the relationship between the overhead involved in retrieving a page and the time required to retrieve each node of the page.

Figure 12.2 shows page retrieval times obtained experimentally for page sizes ranging from 1 to 1024 nodes. It is evident that the page retrieval time is substantially linear in the page size and is given by

\[ t_p = t_0 + pt_n \]

where \( p \) is the number of nodes per page and \( t_0 \) and \( t_n \) are constants dependent on the environment.

The effect on search speed of varying the page size can be analysed as follows. Let \( N \) be the total number of nodes in the tree, and let \( k \) be the number of nodes in a page.

If each page constitutes a complete binary tree (that is, all leaf nodes are on the same level) and the tree as a whole is balanced, then the height of the whole tree in nodes is

\[ h_T = \log_2(N + 1) \]
and the height of each page is

\[ h_p = \log_2(k + 1) \]

so that the height of the tree in pages is

\[ H = \frac{h_T}{h_p} \]

Define the node fanout factor \( f \) of the search as the average number of descendants of a visited node that are also visited. If the search visits the root node of a page, it will fan out \( h_p \) times within the page by a factor of \( f \) and exit through \( F \) branches from leaf nodes of the page, where

\[ F = f^{h_p} = f^{\log_2(k + 1)} \]

Since each branch leaving a page leads to a different page, \( F \) can be regarded as the page fanout factor. The total number of pages visited is then
provided that $N$ and $k$ are not too small. The time to retrieve $P$ pages is

$$t = P t_p = P(t_0 + kt_n)$$

To minimise $t$ we require

$$\frac{dt}{dk} = \frac{dP}{dk} (t_0 + kt_n) + Pt_n = 0$$

Substituting

$$\frac{dP}{dk} = \frac{d}{dk} \left( \frac{N \log f}{k} \right)$$

and solving for $k$ gives

$$k = \frac{t_0}{t_n} \frac{\log f}{1 - \log f}$$  \hspace{1cm} (12.1)$$

So we see that, to a first approximation, the optimum page size for a given $t_0$ and $t_n$ is independent of the size of the tree, and depends only on the value of $f$. 

From Figure 12.2, the following values for $t_0$ and $t_n$ were obtained: 

Using these values in Eq. (12.1) gives the following optimum page sizes for varying $f$ values:
Since $f$ varies from search to search, it is clearly not possible to find a single $k$ value that is optimum for all searches. However, we would like if possible to find one that is reasonably close to optimal for the range of $f$ values likely to be encountered.

To make some assessment of the range of $f$ values found in practice, $f$ values were calculated using statistics gathered from the breadth-first k-d tree search experiment of Chapter 11. For each retrieval, the total number of nodes in all trees searched was determined, together with the number of nodes visited during the search. From these figures, an "equivalent" $f$ value was calculated which would give the same number of visited nodes in from single, balanced tree containing the same total number of nodes.

The equivalent $f$ values obtained are listed in Table 12.1 together with the total number of nodes and the number of visited nodes.

Two overall $f$ values were calculated, one by summing the node count columns and calculating an equivalent $f$ value using the same procedure described above, and the other by taking an average of the individual $f$ values weighted by the number of visited nodes. Both methods gave a result of 1.45.

<table>
<thead>
<tr>
<th>Retrieval no.</th>
<th>Total no. of nodes in trees searched</th>
<th>Number of nodes visited</th>
<th>Equivalent $f$ value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>89</td>
<td>89</td>
<td>2.00</td>
</tr>
<tr>
<td>2</td>
<td>161</td>
<td>24</td>
<td>1.37</td>
</tr>
<tr>
<td>3</td>
<td>12901</td>
<td>315</td>
<td>1.44</td>
</tr>
<tr>
<td>4</td>
<td>22825</td>
<td>229</td>
<td>1.36</td>
</tr>
<tr>
<td>5</td>
<td>22825</td>
<td>165</td>
<td>1.32</td>
</tr>
<tr>
<td>Total</td>
<td>58801</td>
<td>822</td>
<td>1.45</td>
</tr>
</tbody>
</table>
To assess the effect of varying $f$ with a fixed page size, the expected retrieval time per node was plotted for a range of $f$ and $k$ values. The number of nodes retrieved is approximately

$$n = \sum_{i=0}^{i=h-1} f^i = \frac{f^{h-1} - 1}{f - 1} = \frac{(N + 1)^{\log_f f}}{f - 1}$$

and the time to retrieve them is

$$t = \frac{P_{tp}}{(N + 1)^{\log_f f}} (t_0 + kt_n)$$

so the retrieval time per node is

$$\frac{t}{n} = \frac{f - 1}{(k + 1)^{\log_f f}} (t_0 + kt_n) \quad (12.2)$$

Figure 12.3 shows the retrieval times predicted by Eq. (12.2) for a range of $f$ and $k$ values. According to this figure, a page size of around 2$^6 = 64$ should give a reasonable compromise for a wide range of $f$ values.

### 12.3.4 Experimental results

Figure 12.4 shows the spatial index lookup times obtained from Benchmark A3 for a range of page sizes. The best overall performance was obtained with a page size of 64, in agreement with the theoretical analysis above.

Table 12.2 lists the spatial index lookup times obtained with a page size of 64, together with the maximum search depth, number of pages and nodes retrieved, and the number of objects found intersecting the window.
The performance obtained from the paged K-D tree compared favourably with that of the locational key index of Chapter 10, and in some cases was 2 or 3 times faster.

Compared with the k-d tree searching technique used in Chapter 11, under ULingres the performance of the paged k-d tree was similar or slightly better. Under RTI Ingres, the paged k-d tree was vastly more efficient, and gave times that are much more acceptable for interactive use.
12.4 Conclusions

In the previous chapter, a technique was investigated for making use of the processing abilities of a relational database system to search a k-d tree spatial index. It was found that the complex query statements involved could not be executed quickly enough by the DBMS to provide adequate interactive performance.

In this chapter, an alternative method has been investigated which uses much simpler query statements and performs more processing outside the DBMS. The method is based on standard techniques for managing tree structures in secondary storage, applied to the k-d tree. It has been shown experimentally that the method is capable of providing reasonably good interactive performance.

By making certain assumptions about the branching factor of the search within the k-d tree, a theoretical estimate can be found for the optimum page size. These assumptions have been empirically justified, and the resulting estimate has been verified experimentally.
Chapter 13

The SKD Tree

Methods which map 1- or 2-dimensional objects in 2-space to points in 4-space have the disadvantage that objects nearby in 2-space are not necessarily nearby in 4-space, so that a query which is spatially localised in 2-space may require searching widely separated parts of the 4-space index (Samet 1990).

Since the great majority of spatial queries will be localised in 2-space, this is a serious problem. To overcome it, variants of the 2-dimensional k-d tree have been devised to organise 1- and 2-dimensional objects in 2-space.

This chapter discusses two of these techniques. Section 13.1 describes a structure due to Matsuyama, Hao and Nagao (1984). Section 13.2 describes a structure proposed by Ooi (1990) called an skd tree. Section 13.3 presents the results of an experiment in which Ooi’s skd tree was implemented and its performance compared with the paged k-d tree of Chapter 12.

13.1 Matsuyama’s k-d tree

Matsuyama, Hao and Nagao (1984) proposed a spatial data structure in which the object space is alternately divided in the x and y directions until regions are obtained containing data that will fit in a page of secondary storage. A k-d tree is used to index the collection of data pages.

An object may cross a page boundary, but can be stored in only one page; the page chosen is the one whose region contains the object’s centroid. So that window queries can be answered without missing some objects, each data page also contains pointers to objects which overlap the region associated with the page but are stored in some other page.

Since the tree will be relatively small compared to the collection of data pages that it indexes, they propose that the tree be kept in main memory to speed searching.
A disadvantage of Matsuyama et al.'s structure is the extra storage space and complexity involved in maintaining multiple pointers to data objects. Also, in a very large database, it is conceivable that the index tree could become too large to keep in memory.

13.2 The SKD tree

Ooi (1990) proposed a structure which he calls a *spatial k-d tree*, or *skd* tree, which has similarities to both Matsuyama's structure and to the R-tree.

The skd tree is a paged binary tree containing two kinds of nodes, leaf and nonleaf. Each node is associated with a rectangular region to which objects are assigned.

Objects belonging to a nonleaf node are divided into two sets according to the position of their centroids along the splitting dimension, and assigned to respective subnodes. The region of each subnode is made large enough to enclose the objects which it contains.

When the number of objects in a region is sufficiently small, a leaf node is created and associated with a data page. The data page contains the bounding rectangles of the objects and pointers to the spatial data defining them (stored elsewhere).

Figure 13.1 shows a collection of rectangles and a corresponding skd tree with a data page capacity of 2 objects. The dotted rectangles show the regions associated with each node of the tree.

The main difference between Matsuyama's structure and the skd tree is the way in which the region corresponding to a nonleaf node is subdivided. In Matsuyama's tree the regions associated with a node's descendants are disjoint, and objects may lie partly outside their assigned regions.
In the skd tree, the two subregions of a node may overlap, and objects are always wholly contained in their respective regions. Two discriminator values are stored in each nonleaf node, giving the positions of the edges of the two subregions along the splitting dimension.

Unlike Matsuyama et al, Ooi does not propose that the whole index be kept in main memory. Instead, the nodes of the tree are grouped into pages as was discussed in Chapter 12, so that it should be possible to handle arbitrarily large indexes.

Ooi compares the skd tree with the R-tree and shows that intersection searches (finding objects that intersect a given rectangle) can be carried out as efficiently in the skd tree as in the R-tree, and that containment searches (finding objects that are contained within a given rectangle) can be carried out more efficiently in the skd tree.

13.3 An experiment using the skd tree

To assess the effectiveness of the skd tree at making the search more localised, an skd tree spatial index was implemented and tested. Two relations were used, skdnode to hold the index nodes and skddata to hold the data items.

The fields of skdnode are used in two ways, depending on whether the node is a leaf or nonleaf node, as follows:

<table>
<thead>
<tr>
<th>skdnode</th>
<th>skid</th>
<th>type</th>
<th>disc</th>
<th>bound1</th>
<th>bound2</th>
<th>ptr1</th>
<th>ptr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonleaf:</td>
<td>node ID</td>
<td>“N”</td>
<td>“X” or “Y”</td>
<td>maxlo</td>
<td>minhi</td>
<td>loson</td>
<td>hison</td>
</tr>
<tr>
<td>Leaf:</td>
<td>node ID</td>
<td>“L”</td>
<td>“X” or “Y”</td>
<td>minbound</td>
<td>maxbound</td>
<td>data page</td>
<td>unused</td>
</tr>
</tbody>
</table>

For both types of nodes, skid holds a unique node ID, formed from the concatenation of an index page ID and a node number within the page.

For nonleaf nodes, bound1 and bound2 give the maximum coordinate of the left subregion and the minimum coordinate of the right subregion, along the axis indicated by the disc field. Ptrl and ptr2 contain the node IDs of the left and right child nodes. The discriminating axis is chosen to minimise the amount of overlap between the two subregions.

For leaf nodes, bound1 and bound2 contain the minimum and maximum coordinates of the bounding rectangle of the associated data page, along the axis indicated by disc. The axis is chosen to minimise the bounding rectangle. The ptr1 field is used to hold the data page ID of the associated data page.
Each tuple of *skddata* holds the bounding rectangle of an object:

<table>
<thead>
<tr>
<th><strong>skddata</strong></th>
<th>dpid</th>
<th>minx</th>
<th>miny</th>
<th>maxx</th>
<th>maxy</th>
<th>objid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data page ID</td>
<td>Minimum bounding rectangle</td>
<td>Object ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Both relations were ISAM-structured on their respective ID fields so that tuples belonging to a given page could be retrieved together efficiently.

As with the k-d tree, a cache was used to hold nodes from index pages. No cache was used for the data pages, since the cache was cleared before carrying out each window query, and each data page is only referenced once per query (when its associated leaf node is encountered in the search). In practice, the cache would be retained and both index and data nodes would be cached.

Figure 13.2 shows the results obtained for page sizes ranging from 1 to 4096 tuples. The results were very similar to those from the K-D tree, with a page size of 64 once again providing the best overall performance.

Table 13.1 lists statistics obtained by using a page size of 64, alongside corresponding figures from the K-D tree.

<table>
<thead>
<tr>
<th>DBMS Retr. no.</th>
<th>No. of layers</th>
<th>No. of pages</th>
<th>Time</th>
<th>No. of index pages</th>
<th>No. of data pages</th>
<th>Index page time</th>
<th>Data page time</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-D Tree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U1</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>0.9</td>
<td>7</td>
<td>0.7</td>
<td>0.8</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>0.8</td>
<td>5</td>
<td>0.5</td>
<td>0.4</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>14</td>
<td></td>
<td>2.3</td>
<td>6</td>
<td>0.7</td>
<td>1.0</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>20</td>
<td>40</td>
<td>6.1</td>
<td>22</td>
<td>2.6</td>
<td>2.3</td>
<td>4.9</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>20</td>
<td>38</td>
<td>5.7</td>
<td>22</td>
<td>2.5</td>
<td>2.1</td>
<td>4.6</td>
</tr>
<tr>
<td>RT1</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>0.3</td>
<td>7</td>
<td>0.2</td>
<td>0.2</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>0.3</td>
<td>5</td>
<td>0.1</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>14</td>
<td></td>
<td>0.8</td>
<td>6</td>
<td>0.2</td>
<td>0.3</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>20</td>
<td>40</td>
<td>2.2</td>
<td>22</td>
<td>0.8</td>
<td>0.8</td>
<td>1.6</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>20</td>
<td>38</td>
<td>1.9</td>
<td>22</td>
<td>0.8</td>
<td>0.7</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Table 13.1
SKD Tree search statistics from Benchmark A3
with page size = 64
Chapter 13 – The SKD Tree

Figure 13.2
SKD tree spatial index lookup times
(Benchmark A3)
Due to the relatively small size of the test database, these results should be interpreted with caution. The use of a separate tree for each representation of each layers, and the division of the data into many layers, meant that many of the trees were quite small. At a page size of 64, most of the trees were no more than two pages deep, and the trees for some of the higher level representations fitted in a single page.

Looking at the number of layers referenced in each retrieval (third column of Table 13.1), it can be seen that the number of pages being retrieved is very close to the minimum possible. For example, in retrieval 1, a single page is being retrieved from each tree in the k-d case, and one index and one data page in the skd case. In retrieval 5, approximately two pages are being retrieved from each tree in both cases.

Under such conditions, it is difficult to determine how much improvement in locality is being obtained from the skd tree. It does appear, however, that there is some improvement at the lower viewing levels, as the total number of pages retrieved from the skd trees (index + data) is slightly less than from the k-d tree, with an accompanying increase in retrieval speed.

13.4 Conclusions

An skd tree spatial index has been implemented and compared with a 4-dimensional k-d tree. The results suggest that window searches in the skd tree are slightly more localised than in a corresponding 4-d tree, and thus can be expected to give faster retrievals.
Chapter 14

Using Custom Storage Structures

The tree-searching techniques used in Chapters 12 and 13 make very little use of the data processing abilities of the relational database system. The simple data retrieval operations needed could be performed just as easily if the data were kept in ordinary files, and doing so would avoid the considerable overhead of going through the relational database system.

This chapter presents the results of an experiment to compare the performance of retrievals from the relational database versus a custom data structure, using the skd tree spatial index and lookup algorithm described in Chapter 13.

Section 14.1 discusses the relative merits of using a relational database system as against a custom storage structure. Section 14.2 describes the custom storage structure used in the experiment, and Section 14.3 presents the results.

14.1 Introduction

If retrieval is faster without the relational database, then why use the relational database at all? Interactive browsing will probably not be the only use of a geographic database, and the efficiency of maintaining and updating the database must also be considered. The high degree of data independence offered by a DBMS is of proven benefit when maintaining and updating a database used by many applications, and the power and flexibility of relational database systems helps greatly in developing new applications and answering \textit{ad hoc} queries.

These benefits provide strong motivation for using a relational database to hold the data, and to minimise redundancy it is clearly desirable for all applications, including browsing, to use the same database if possible.

Nevertheless, if the performance penalty of using the relational database for browsing is too great, keeping a copy of all or some of the data in a separate data structure for browsing may be justified. Whether this is necessary will depend on how great the difference in performance is.
14.2 The custom data structure

The custom data structure consisted of a set of ordinary files, with each file corresponding to one of the relations $skdnode$, $skddata$, $curve$ and $text$. Each file was structured as an array of records, each record containing a tuple from the associated relation minus the ID field. A tuple was retrieved from the file by computing its offset from the ID value, seeking to the offset and reading a record. Tuples belonging to a page of the $skdnode$ and $skddata$ files were stored consecutively, so that a page could be retrieved in a single read.

Tuples of $curve$ belonging to a single object were also stored consecutively. Because the number of $curve$ tuples per object is variable, an auxiliary index file was constructed to facilitate finding objects in the $curve$ file. The index entry of an object was located using the object ID and contained the offset of the first tuple in the $curve$ file and the number of tuples.

A similar index file was used in conjunction with the $text$ file, since the records in this file were made variable-length to take up only as much space as required for each text string.

14.3 Experimental results

The skd tree search algorithm was evaluated under the following conditions:

1. Data retrieved from the relational database
   
   a. using UIngres
   
   b. using RTI Ingres

2. Data retrieved from custom files residing
   
   a. on a local disk
   
   b. on disks attached to a file server accessed via NFS

The results are presented in Table 14.1.
Table 14.1
Comparison of retrieval times from relational database and custom files
(Benchmark A3)

<table>
<thead>
<tr>
<th>Retrieval no.</th>
<th>Spatial index lookup time</th>
<th>Spatial data retrieval time</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RDBMS</td>
<td>Files</td>
<td>RDBMS</td>
</tr>
<tr>
<td></td>
<td>UI</td>
<td>Local</td>
<td>RTI</td>
</tr>
<tr>
<td>1</td>
<td>0.85</td>
<td>0.03</td>
<td>0.01</td>
</tr>
<tr>
<td>2</td>
<td>0.53</td>
<td>0.00</td>
<td>0.02</td>
</tr>
<tr>
<td>3</td>
<td>1.07</td>
<td>0.02</td>
<td>0.04</td>
</tr>
<tr>
<td>4</td>
<td>3.09</td>
<td>0.06</td>
<td>0.04</td>
</tr>
<tr>
<td>5</td>
<td>2.97</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>1</td>
<td>2.27</td>
<td>0.19</td>
<td>0.19</td>
</tr>
<tr>
<td>2</td>
<td>0.73</td>
<td>0.09</td>
<td>0.10</td>
</tr>
<tr>
<td>3</td>
<td>2.73</td>
<td>0.08</td>
<td>0.14</td>
</tr>
<tr>
<td>4</td>
<td>0.71</td>
<td>0.34</td>
<td>0.37</td>
</tr>
<tr>
<td>5</td>
<td>0.50</td>
<td>0.19</td>
<td>0.34</td>
</tr>
<tr>
<td>1</td>
<td>3.12</td>
<td>0.22</td>
<td>0.20</td>
</tr>
<tr>
<td>2</td>
<td>1.26</td>
<td>0.09</td>
<td>0.10</td>
</tr>
<tr>
<td>3</td>
<td>3.80</td>
<td>0.10</td>
<td>0.16</td>
</tr>
<tr>
<td>4</td>
<td>3.80</td>
<td>0.40</td>
<td>0.41</td>
</tr>
<tr>
<td>5</td>
<td>3.47</td>
<td>0.23</td>
<td>0.38</td>
</tr>
</tbody>
</table>

Searching the spatial index was much faster using the files than either of the relational database systems, a difference of 15 to 50 times being observed. There was less difference, however, between RTI Ingres and the local files in spatial data retrieval speed. In some cases there was a considerable improvement; in others, RTI Ingres appeared to be returning data nearly as fast as it could be read from disk.

14.4 Conclusions

When a tree-structured index such as the SKD tree is used, going through the relational DBMS to retrieve pages of the tree introduces a great deal of inefficiency in comparison with reading directly from disk files. In contrast, retrieving the spatial data itself from the relational database involves less of a performance penalty, and in some cases almost none.

A good compromise would be to leave the spatial data in the relational database, and keep the spatial index in a separate structure. This would allow most of the benefits of the relational database to be retained and provide reasonably optimal retrieval speed for browsing.
An apparent disadvantage of this approach is the need to ensure that the spatial index is rebuilt or updated whenever the main database is updated. However, this would still be necessary even if the spatial index were kept in the relational database.

A more substantial difference is that the powerful facilities of the relational database system would not be available for the maintenance of the spatial index, but since the structure of the spatial index is not likely to be changed frequently, and the set of operations required on the spatial index is small and fixed, this should not be a serious disadvantage.
Chapter 15

Exploiting Locality of Reference

Locality of reference is a phenomenon of which much use is made in computing. Virtual memory systems, disk caches and high-speed memory caches all rely on locality of reference to provide greatly increased performance at moderate cost over that which would otherwise be attainable.

In this chapter, Section 15.1 discusses locality of reference in general, and Section 15.2 discusses ways in which locality of reference arises in map browsing. Sections 15.3 and 15.4 discuss ways in which this locality can be exploited in a map browsing system, and present the results of some experiments to assess the effectiveness of these methods.

15.1 Locality of reference in general

In general, locality of reference is exploited by means of a pair of memories: a small, fast memory M1 and a large, slow memory M2. If references to the contents of M2 are localised within a small set of locations over a reasonable period of time, then by judiciously moving data between M1 and M2, most of the references can be satisfied from M1, the need to go to M2 arising only rarely. The result is the appearance of a memory with the size of M2 and almost the speed of M1.

Table 15.1 lists some examples of systems which exploit locality of reference, together with the memory pair involved in each case.

<table>
<thead>
<tr>
<th>System</th>
<th>M1</th>
<th>M2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual memory</td>
<td>Main memory</td>
<td>Paging area of disk</td>
</tr>
<tr>
<td>File system cache</td>
<td>Main memory</td>
<td>Disk files</td>
</tr>
<tr>
<td>Main memory cache</td>
<td>High-speed cache in processor</td>
<td>Main memory</td>
</tr>
</tbody>
</table>
15.2 Locality of reference in map browsing

Patterns of reference to geographic data during interactive map browsing often exhibit considerable locality of reference. Typically the user will select some area of interest from the reference map and concentrate further attention on and around it - zooming in to reveal more detail, and scrolling to examine adjacent areas.

In addition to spatial locality, references will usually be confined to a subset of the available layers. A large database may have a great many layers containing a wide variety of information, of which only a small proportion will be of interest to a user at one time. Furthermore, there will be sets of related layers that are often viewed together, such as street outlines and street names, or parcel boundaries, section numbers and building outlines, or rivers, lakes and coastlines.

This chapter explores ways in which the various kinds of locality of reference in map browsing can be exploited to make the map browser more efficient. The techniques investigated can be classified into:

- **Cacheing**: Keeping in main memory data previously retrieved from the database that is likely to be referenced again.

- **Prefetching**: Retrieving data that the user has not yet requested but is likely to request soon.

- **Layer grouping**: Identifying sets of layers (and representations of those layers) that are likely to be referenced together, and indexing them accordingly.

15.3 Cacheing and Prefetching

To investigate the effects of cacheing previously retrieved data, the page cache mechanism used with the skd tree in Chapter 13 was extended to include both index and data pages from the skd tree, and spatial data retrieved from the curve and text relations.

15.3.1 Cache architecture

Figure 15.1 shows the architecture of the cache system. The retrieval routines make requests to the cache manager for an item (index page, data page, or spatial object). If the item is already in the cache, it is returned; otherwise, the cache manager retrieves the item from the database, places it in the cache, and returns it to the caller.
15.3.2 Experimental results

Table 15.2 lists the numbers of items referenced, the number found in the cache, and the cache hit rate (the proportion of items required that were in the cache). Table 15.3 shows the retrieval times obtained with the cache enabled, compared with those when the cache was cleared between retrievals. A reasonably high hit rate was obtained for most of the retrievals.

The cache hit rates were observed to fluctuate considerably with viewing level, a result that can be understood by considering the reference pattern of the benchmark. With the window rectangle of each retrieval nested within the one before, the first reference to a given representation of a layer requires database access, but subsequent references to the same representation are satisfied entirely from the cache.

Due to the arrangement of representations in the database and their assigned ranges of viewing levels, the transition from retrieval 2 to 3 crossed the level boundaries of many representations, resulting in a low hit rate. By contrast, retrieval 5 referenced exactly the same representations as retrieval 4, giving a hit rate of 100%.

15.3.3 Other reference patterns

The benefits obtained from the use of a cache will clearly be highly dependent on the reference pattern. The above results show that in the case of one typical reference pattern, that of zooming, a significant benefit is possible, and in some cases a tremendous one.
Experiments in Interactive Map Retrieval

Table 15.2
Database 3 Cache Hit Rates

<table>
<thead>
<tr>
<th>Retrieval no.</th>
<th>No. referenced</th>
<th>No. found in cache</th>
<th>No. not in cache</th>
<th>Cache Hit Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SKD Tree Index Pages</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0%</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>60%</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>33%</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>4</td>
<td>18</td>
<td>18%</td>
</tr>
<tr>
<td>5</td>
<td>22</td>
<td>22</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>SKD Tree Data Pages</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0%</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>33%</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0%</td>
</tr>
<tr>
<td>4</td>
<td>17</td>
<td>3</td>
<td>14</td>
<td>18%</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>Spatial Objects</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>89</td>
<td>0</td>
<td>89</td>
<td>0%</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0%</td>
</tr>
<tr>
<td>3</td>
<td>281</td>
<td>0</td>
<td>281</td>
<td>0%</td>
</tr>
<tr>
<td>4</td>
<td>39</td>
<td>13</td>
<td>26</td>
<td>34%</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>Total Items</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>103</td>
<td>0</td>
<td>103</td>
<td>0%</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>4</td>
<td>7</td>
<td>36%</td>
</tr>
<tr>
<td>3</td>
<td>294</td>
<td>2</td>
<td>292</td>
<td>1%</td>
</tr>
<tr>
<td>4</td>
<td>78</td>
<td>20</td>
<td>58</td>
<td>26%</td>
</tr>
<tr>
<td>5</td>
<td>47</td>
<td>47</td>
<td>0</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 15.3
Database 3 - Retrieval times (sec)
A: caching disabled; B: caching enabled

<table>
<thead>
<tr>
<th>Retr. no.</th>
<th>SKD tree lookup</th>
<th>Object retrieval</th>
<th>Total</th>
<th>SKD tree lookup</th>
<th>Object retrieval</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>0.9</td>
<td>1.0</td>
<td>2.3</td>
<td>2.2</td>
<td>3.1</td>
<td>3.2</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>0.3</td>
<td>0.7</td>
<td>0.7</td>
<td>1.3</td>
<td>1.0</td>
</tr>
<tr>
<td>3</td>
<td>1.1</td>
<td>0.8</td>
<td>2.7</td>
<td>2.3</td>
<td>3.8</td>
<td>3.1</td>
</tr>
<tr>
<td>4</td>
<td>3.1</td>
<td>2.8</td>
<td>0.7</td>
<td>0.6</td>
<td>3.8</td>
<td>3.4</td>
</tr>
<tr>
<td>5</td>
<td>3.0</td>
<td>0.0</td>
<td>0.5</td>
<td>0.0</td>
<td>3.5</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Scrolling is another common reference pattern that should benefit from the cache. In the case of retrieval 5 above, if the user were to scroll the window a small amount, all of the newly-visible data would already be in memory from the previous retrieval, and the window could be refreshed almost instantaneously. Scrolling would be possible all the way to the edge of the retrieval 4 window before further reference to the database was necessary.

15.3.4 Prefetching

Further use could be made of the cache by prefetching data that the user has not yet requested, but is likely to request soon. For example, the window area requested by the user could be expanded and data retrieved for some larger surrounding area. Should the user then decide to scroll, some or all of the data would be immediately available.

The reason this would be effective is that there is considerable overhead involved in processing a window query. In the experiment, retrieval 4 did not take much longer than retrieval 5, despite both retrievals being from the same representations and one having a window covering 100 times the area of the other. In cases like this, quite a large area could be prefetched with no noticeable degradation of performance.

Prefetching could also be carried out during idle time. While the user is thinking, the system could be prefetching areas surrounding active windows as a background task. Priority would need to be given to explicit requests from the user so that the background activity did not impair interactive performance. In a multi-user environment, requests from other users of the database would also need to be given priority over prefetching, so that some means of prioritising requests to the DBMS from multiple users would be necessary.

Which form of prefetching to use would depend on the overhead of a window query in relation to the amount of idle time available. In the example mentioned earlier, there is clearly much overhead, and little time is lost in making a larger retrieval initially. If the overhead were small, it would be better to retrieve only the requested data initially and prefetch in the background.

Prefetching could also be employed in other dimensions, such as viewing levels and layers. Data could be prefetched for the same area as an active window but at a lower viewing level, in anticipation of the user zooming in on some portion. Prefetching of related layers that are not currently being viewed might also be considered.
15.3.5 Replacement strategy

With memory sizes currently found in typical workstations, it is reasonable to assume a cache size of several megabytes. Nevertheless, it is conceivable that a cache of this size could become filled during a long browsing session, whereupon a cache replacement strategy will be necessary.

A reasonable policy would be to give preference to data currently visible in a window, or closely related to an existing window (within its prefetch region, for instance). Data not in this preferred set could be replaced on a least-recently-used basis.

15.4 Layer grouping

If a set of layers are frequently viewed together, it may be advantageous to construct a single skd tree to index them all, rather than using a separate tree for each one.

15.4.1 Analysis

Consider an skd tree containing a total of $N_d$ data pages, from which we wish to retrieve $n_d$ data pages. We will assume that the required data pages are clustered together into a subtree with a height of $h$ pages. If the total height of the tree is $H$ pages, then the search will proceed down through $H - h$ levels, referencing one index page at each level, and then fan out over the subtree containing the required pages, as shown in Figure 15.2.
If there are $b$ branches from each index page, then the height of the whole tree is given by

$$H^{(i)} = \log_b N_d + 1$$

and the height of the subtree by

$$h^{(i)} = \log_b n_d + 1$$

The number of levels traversed in order to reach the subtree is

$$r_i = H^{(i)} - h^{(i)}$$

and the number of index pages in the subtree is

$$s_i = \sum_{i=0}^{h^{(i)-2}} b^i$$

If there are $m$ such trees, the total number of index pages referenced will be

$$n_i^{(i)} = m (r_i + s_i)$$

$$= m \left[ \log_b \frac{N_d}{n_d} + \frac{n_d - 1}{b - 1} \right]$$

Now consider a single tree containing the items from the data pages of the original $m$ trees; this new tree will have a total of $mN_d$ data pages. Provided the window rectangle is not too small compared to the area covered by a data page, carrying out the same window query will retrieve approximately $mn_d$ data pages.

The total height of the tree will be

$$H^{(3)} = \log_b mN_d + 1$$

and the height of the subtree to be searched will be

$$h^{(2)} = \log_b mn_d + 1$$

so that the total number of index pages referenced is now
\[ n_i^{(2)} = H^{(2)} - h^{(2)} + \sum_{i=0}^{b^2} b^i \]
\[ = \log_b \frac{N_d}{n_d} + \frac{m n_d - 1}{b - 1} \]
\[ = \log_b \frac{N_d}{n_d} + m \frac{n_d - 1}{b - 1} + \frac{m - 1}{b - 1} \]
\[ = n_i^{(2)} - (m - 1) \left[ \log_b \frac{N_d}{n_d} - \frac{1}{b - 1} \right] \]

In other words, the number of index pages that must be retrieved is less when a single tree is used than when individual trees are used, the difference being proportional to the number of trees eliminated.

15.4.2 Experimental results

An experiment was conducted to test the prediction made in Section 15.4.1 above. The layers listed in Table 15.4 were selected as a typical set that a user might want to view together, and the lowest-level representations of them were indexed in two ways:

(a) A separate SKD tree for each layer

(b) A single SKD tree for all five layers.

Table 15.4
Representations used in layer grouping experiment

<table>
<thead>
<tr>
<th>No.</th>
<th>Layer</th>
<th>Viewing levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Roads</td>
<td>1.0-5.0</td>
</tr>
<tr>
<td>2</td>
<td>Section boundaries</td>
<td>1.0-5.0</td>
</tr>
<tr>
<td>3</td>
<td>Building outlines</td>
<td>1.0-3.0</td>
</tr>
<tr>
<td>4</td>
<td>Private foulwater drains</td>
<td>1.0-3.0</td>
</tr>
<tr>
<td>5</td>
<td>Foulwater sewers</td>
<td>1.0-3.0</td>
</tr>
</tbody>
</table>

Results were recorded for retrievals 3 to 5 of Benchmark A3 (these retrievals being the only ones that referenced the representations concerned).

Table 15.5 shows the number of pages retrieved from the spatial index for varying numbers of layers retrieved, for the separate trees case. Table 15.6 shows the corresponding numbers for the single tree case (the numbers are independent of the number of layers in this case).
The results for retrievals 4 and 5 agreed with those expected in that, when all 5 layers were being retrieved, fewer index pages were referenced in the single tree case.

Figure 15.3 compares the spatial index lookup times for the two cases. For retrievals 4 and 5, it is clear that the single tree was more efficient when more than half the layers were being referenced.

For retrieval 3, however, the single tree was much less efficient. Since that retrieval was at a viewing level of 3.5, only two of the five representations in the tree were relevant, so that much time was wasted retrieving index entries that would never be used.
The problem stems from the fact that not all of the representations in the tree span the same range of viewing levels, so that inevitably there are some viewing levels for which the arrangement is not optimal.

There are a couple of ways that this problem could be remedied. One would be to split the representations into two groups, one covering levels 1.0-5.0 and the other 1.0-3.0, and index them separately (Figure 15.4a). This arrangement would improve the efficiency of retrievals at levels 3.0-5.0 at the expense of some degradation at levels 1.0-3.0, where two trees would need to be searched.

Alternatively, all five layers could be indexed by a tree for use at levels 1.0-3.0, with an extra tree indexing those representations which also cover levels 3.0-5.0 (Figure 15.4b).
Figure 15.4
Two ways of indexing representations that span different ranges of viewing levels
This arrangement would allow maximum efficiency at all levels, but would require some extra storage space, since some of the representations are indexed twice.

15.5 Conclusions

Typical interactive map browsing reference patterns, such as zooming and scrolling, exhibit considerable locality of reference, both spatially and in other dimensions such as layers.

Spatial locality can be exploited by keeping a cache of retrieved data, which can substantially improve the performance of zooming and scrolling. In the case where a zooming operation does not cross a representation boundary, the request can be satisfied entirely from the cache.

Locality could be further exploited by prefetching data that the user has not yet requested but is likely to request soon. Such data includes data for areas surrounding a viewed area, data for the same area from a lower-level representation, and perhaps data for the same area from related layers. Prefetching could be done either by expanding the scope of the initial retrieval performed at the user's request, or by performing further retrievals in the background during idle time.

If several layers are being retrieved from, it is faster to look up a single index tree covering them all than to look up an individual index for each layer. Empirical results suggest that a saving is achieved when more than half the layers in the tree are being referenced.

So, to achieve the best performance, if a group of layers has a high probability of being viewed together they should be indexed by a single tree, whereas unrelated layers should be indexed by separate trees.

The representations indexed by a given tree should all cover the same range of viewing levels. To achieve this it may be necessary to either split a group of layers into two or more separately-indexed groups, or provide multiple indexes for some of the representations covering different ranges of viewing levels.
Chapter 16

Conclusions

The main theme of this thesis has been to investigate the design and implementation of graphical, interactive browsers for large databases of geographic information.

It has been assumed that the user of a map browsing system will have access to a powerful, personal workstation or sophisticated graphics terminal connected by a network to a large database, possibly distributed, of geographic information.

The results of work done in two main areas has been presented: graphical user interfaces, both generally and as regards interactive map browsers; and design of geographic databases to support map browsing systems.

16.1 User interface

The design of a map browsing system from the user’s point of view has been discussed. To the user, the database should appear as a large, seamless map, made up of multiple layers, each containing geographic information on a particular theme. A set of basic functions that a map browser should provide has been proposed. These functions include zooming, panning, selection of layers to view, and creating and managing multiple windows viewing different parts of the map.

Additional functions that a map browser could usefully provide have also been suggested, such as alternative map projections, retrieval of non-spatial attribute data for a feature, visual overlaying of map layers, editing facilities for visually enhancing displayed maps, and saving or printing of maps thus created.

A division of the map browsing system into software components has been described. These components may be distributed across a network in a variety of ways, to make the best use of the available computing and database resources wherever they reside.

Implementing the user interface of a map browser is a special case of the general problem of implementing applications with graphical user interfaces. The complexity of
modern user interfaces makes this problem a major one, and much effort has and is being put into solving it. Two main classes of tools which address the problem have been discussed.

User interface metaphors can be classified into two classes: conversational world, characterised by step-by-step sequential dialogues between the user and the application, and model world, characterised by multithreaded dialogues and direct-manipulation techniques.

Conversational world interfaces are sufficiently simple that the dialogue part of an application can be separated from the computational part and described using a declarative formalism from which an implementation can be automatically generated. Formalisms that have been used for this purpose include context-free grammars, finite state machines and special-purpose programming languages.

In contrast, model world interfaces often incorporate a large part of the application's functionality and exhibit diverse and complex behaviour, making them difficult to describe formally except by using a general-purpose programming language. Most of the successful systems for constructing model world interfaces are based on libraries written in an object-oriented programming language, and are known as object-oriented frameworks or toolkits.

In a network environment, new problems related to graphical user interfaces arise. The concept of a display server has been developed to allow the user interface of an application to appear on a user's workstation while computation is performed on another machine. Two display server systems have been discussed, the widely used X Windows system, and Sun Microsystems' PostScript-based extensible NeWS system.

The author chose NeWS as the basis for implementing the user interface of an experimental map browsing system. NeWS was chosen because of the advantages of PostScript's rich graphics model for the intended application of displaying maps.

Working with an early, partially-developed version of NeWS, the author found it necessary to develop his own user interface toolkit. Creating user interface code for NeWS involves writing large PostScript programs by hand. The author found that, due to the nature of the PostScript programming language, this task is tedious and error-prone.

To make the task easier, the author has designed and implemented a new language, P, which makes the facilities of PostScript available to the programmer in a
much more convenient form. P is implemented by means of a translator that converts P source to PostScript source. Experience has shown P to be an effective tool for creating understandable and error-free programs, both for NeWS and in other situations requiring hand-written PostScript. There is also potential for P to support a source-level debugging environment for NeWS in a more comprehensive way than would be possible using PostScript alone.

The Mockintosh, an object-oriented user interface toolkit for NeWS developed by the author, has been described. Some conclusions reached as a result of this experience have been presented. The main conclusions concern communication between applications and the user interface code residing in the NeWS server. One important conclusion is that all of the communication between the application and the user interface code should be in application-specific terms. Some remarks have also been made about effective use of lightweight processes and graphics states in NeWS toolkits generally.

The following observations have been made about the general communication model between the NeWS server and its clients, and the tools provided for constructing client-server interfaces. The NeWS communication model is asymmetrical, being based on a remote procedure call model from client to server, and a message-passing model from server to client. This model does not match the structure of event-driven interactive applications, in which user interface objects, in response to user actions, invoke computational functions in the application.

The communication model could be made more symmetrical by extending the protocol-construction tools provided with NeWS to allow the specification of remote procedure calls in both directions. In the case where the client is written in an object-oriented language, the tools could be further extended to allow objects in server and client to communicate with each other naturally in an object-oriented way.

With window systems like X Windows becoming widely used and standardised, it is becoming common for users to use software from diverse sources side by side on the same workstation. In this environment, maintaining consistency of the style or "look and feel" of user interfaces between applications becomes a problem. The author has shown that the design of current graphical user interface toolkits results in the style of an application's interface being largely fixed when the application is created, both by the choice of toolkit and by decisions made when the application is written.

A new architecture for user interface systems has been proposed, based on the
requirement that the user interface style should be completely under the control of the user. The user should be able to select an interface style of choice, and all the applications he or she uses should automatically conform to that style.

The author has made the key observation that, for this scheme to work, the interface between applications and the user interface system must be completely *style independent*; that is, it must not contain anything that depends on any particular user interface style. It is the lack of style independence in the application program interfaces of current user interface toolkits that make it impossible to change their user interface styles in an arbitrary way.

The beginnings of a design for a style-independent API have been presented. Further development of this design will require several problems to be addressed. The nature of these problems has been discussed, and possible ways of overcoming them have been outlined.

### 16.2 Database design

An extension to general geographic data models has been proposed to support the requirements of interactive browsing. The main extension required is to allow data to be retrieved that is suitable for display at a variety of scales, or viewing levels. Two methods of accomplishing this have been suggested. One is to store multiple copies of the data, designed for display at different viewing levels; it has been shown that the amount of extra data that this would involve is minimal and would not unduly increase storage requirements. The other method is to store spatial data in a variable-resolution data structure, an example being the striptree.

To support rapid retrieval of data for a particular geographic region, a spatial index will be required, or alternatively the spatial data itself could be stored in a spatially-organised data structure. Also, it will be desirable for the database to incorporate a catalogue of layers, containing descriptive information about each layer for the user of the map browser to use in selecting layers to view.

Results have been presented from a set of experiments with different techniques of storing, indexing and retrieving spatial data in a relational database environment. The primary objective was to gain insights into the relative merits of these techniques for supporting interactive browsing. A secondary objective was to discover whether spatial data could be retrieved from a relational database with sufficient speed for interactive use.
The experiments can be classified into three groups, concerned with: (1) storage and retrieval of spatial objects; (2) spatial indexing; (3) locality of reference.

A preliminary set of experiments showed that spatial objects represented in a relational database can be retrieved with sufficient speed to support interactive browsing. These experiments verified that it is very important to cluster objects representing geographically nearby entities so that they can be retrieved with a minimum of disk accesses. A method has been demonstrated of achieving this in database systems which do not allow clustering to be requested directly.

The flexibility of the relational database makes it attractive for storing geographic data, but conventional relational database systems provide only one-dimensional indexing facilities which are not directly suitable for spatial indexing. Two classes of technique for overcoming this problem were investigated.

One class aims to make use of the database system's existing indexing methods and query processing abilities to provide a spatial index. Two techniques of this sort were evaluated: (1) Abel and Smith's method, which implicitly uses a quadtree to assign locational keys to objects, and organises the keys using a one-dimensional index; and (2) a method in which a k-d tree was explicitly represented and searched in a breadth-first manner using database query statements.

Of these two techniques, only Abel and Smith's method was fast enough to provide adequate interactive performance. The successfulness of Abel and Smith's method seemed to depend critically on characteristics of the particular relational database system used: an implementation of the multiple-key variant of their method failed to work because of an idiosyncrasy of RTI Ingres.

The other class of technique uses the relational database mainly as a repository of data and carries out most of the processing elsewhere. Two paged tree structures were evaluated, a paged k-d tree and an skd tree. Both of these performed better than the single-key version of Abel and Smith's method, and provided reasonably good interactive performance. The skd tree appeared to be slightly more efficient under some conditions due to searches being more localised within the tree.

An experiment was conducted to determine how much efficiency could be gained by storing the spatial index and/or spatial data outside the relational database. It was found that the spatial index could be searched up to 50 times faster by reading index pages directly from a disk file rather than retrieving them from a relation. Reading spatial object data from disk files was also faster, but the difference was less
pronounced, and under some conditions there was almost no advantage.

Locality of reference in map browsing has been discussed. Typical patterns of reference in map browsing that exhibit locality include zooming, scrolling and examination of thematically related layers. Techniques for exploiting this locality of reference, both in the organisation of the database and the implementation of the map browser, have been suggested. These techniques include cacheing previously retrieved data, prefetching data not yet requested but likely to be requested soon, and combining related layers for the purposes of spatial indexing. Results of experiments to assess the effectiveness of these techniques indicate that they can be very effective at improving the interactive responsiveness of the map browser.

The main conclusions from these experiments can be summarised as follows:

(1) Where retrieval speed is the primary concern, the most effective way of implementing a spatial index in a conventional relational database is to use a paged tree structure, such as an skd tree, and use the relational database simply as a repository for the index pages.

(2) If more speed is required, considerable efficiency can be gained by storing the spatial index outside the relational database system. A further, although smaller, gain can be made by storing the spatial data outside the relational database as well. A good compromise may be to keep the spatial data in the relational database, retaining much of the flexibility, while storing the spatial indexes outside to improve retrieval speed.

(3) A substantial contribution to the interactive performance of the map browser can be made by the use of techniques for exploiting locality of reference, such as cacheing and prefetching, and by organising the database to take advantage of common reference patterns. These techniques are not restricted to relational database systems but are applicable in general.
Appendix A: P Syntax and Semantics

A.1 Syntax

The syntax of P is described here in an extended BNF. The notation [...] means that the enclosed items are optional, and {...} means that the enclosed items may be repeated zero or more times. Symbols in bold type are reserved words and terminal symbols of the language.

```
program ::= expression-sequence
expression-sequence ::= expression (; expression)
expression ::= [boolean-expression [assignment-operator boolean-expression]]
assignment-operator ::= ::= !:=

boolean-expression ::= boolean-term [or boolean-term]
boolean-term ::= boolean-factor [and boolean-factor]
boolean-factor ::= relational-expression [not boolean-factor]
relational-expression ::= arithmetic-expression
                           [relational-operator arithmetic-expression]
relational-operator ::= = | <> | < | > | <= | >=

arithmetic-expression ::= term [adding-operator term]
adding-operator ::= + | -
term ::= factor [multiplying-operator factor]
multiplying-operator ::= * | / | %
factor ::= - factor | element | procedure-call | index-expression | method-call

procedure-call ::= factor [ ( expression-list ) ]
index-expression ::= factor [ expression [ @ expression ] ]
method-call ::= receiver . element [ ( expression-list ) ]
element ::= identifier | literal | block | control-structure | ( expression )
literal ::= 'identifier | quoted-string | number
expression-list ::= expression , expression

block ::= { [parameters] [local-declarations] [statement-sequence] }
parameters ::= | name-list |
name-list ::= identifier , identifier
local-declarations ::= local declaration-list ;
declaration-list ::= declaration , declaration
declaration ::= identifier [= expression]

statement-sequence ::= statement ; statement
statement ::= [case-labels] expression
case-labels ::= name-list :

control-structure ::= if-statement | for-statement | forall-statement |
pathforall-statement | loop-statement | repeat-statement |
with-statement | case-statement | class-declaration
```
if-statement ::= if expression then expression [else expression]
for-statement ::= for expression to expression [by expression] do expression
forall-statement ::= forall expression do expression
loop-statement ::= loop expression
repeat-statement ::= repeat expression times expression
with-statement ::= with expression do expression
case-statement ::= case expression of expression
pathforall-statement ::= pathforall moves expression
     lines expression
     curves expression
     closes expression

class-declaration ::= class element . identifier ;
     [cvars declaration-list ;]
     [cmeths expression-sequence]
     [ivaI's declaration-list i]
     [imeths expression-sequence]

A.2 Semantics

The semantics of each P construct is defined here in terms of its translation into PostScript\(^1\).

<table>
<thead>
<tr>
<th>P</th>
<th>PostScript</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr1 ; expr2</td>
<td>expr1 expr2</td>
</tr>
<tr>
<td>expr1 , expr2</td>
<td>expr1 expr2</td>
</tr>
</tbody>
</table>

Expression lists and sequences

Assignments

| ident ::= expr  | ident expr def                     |
| expr1 ::= expr2 | expr1 expr2 def                    |

| ident ::= expr  | ident expr store                   |
| expr1 ::= expr2 | expr1 expr2 store                  |
| expr1 [expr2 ] := expr3 | expr1 expr2 expr3 put |
| expr1 [expr2 @] := expr3 | expr1 expr2 expr3 putinterval |

\(^1\)Translation of the language constructs for defining classes is implementation-dependent and therefore is not listed here.
### Unary operators

<table>
<thead>
<tr>
<th>not expr</th>
<th>expr not</th>
</tr>
</thead>
<tbody>
<tr>
<td>- expr</td>
<td>expr neg</td>
</tr>
</tbody>
</table>

### Binary operators

<table>
<thead>
<tr>
<th>expr1 or expr2</th>
<th>expr1 expr2 or</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr1 and expr2</td>
<td>expr1 expr2 and</td>
</tr>
<tr>
<td>expr1 = expr2</td>
<td>expr1 expr2 eq</td>
</tr>
<tr>
<td>expr1 &lt;&gt; expr2</td>
<td>expr1 expr2 ne</td>
</tr>
<tr>
<td>expr1 &lt; expr2</td>
<td>expr1 expr2 lt</td>
</tr>
<tr>
<td>expr1 &gt; expr2</td>
<td>expr1 expr2 gt</td>
</tr>
<tr>
<td>expr1 &lt;= expr2</td>
<td>expr1 expr2 le</td>
</tr>
<tr>
<td>expr1 &gt;= expr2</td>
<td>expr1 expr2 ge</td>
</tr>
<tr>
<td>expr1 + expr2</td>
<td>expr1 expr2 add</td>
</tr>
<tr>
<td>expr1 - expr2</td>
<td>expr1 expr2 sub</td>
</tr>
<tr>
<td>expr1 * expr2</td>
<td>expr1 expr2 mul</td>
</tr>
<tr>
<td>expr1 / expr2</td>
<td>expr1 expr2 div</td>
</tr>
<tr>
<td>expr1 % expr2</td>
<td>expr1 expr2 mod</td>
</tr>
</tbody>
</table>

### Procedure and method calls

<table>
<thead>
<tr>
<th>ident(exprlist)</th>
<th>exprlist ident</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr(exprlist)</td>
<td>exprlist expr exec</td>
</tr>
<tr>
<td>expr.ident</td>
<td>expr ident send</td>
</tr>
<tr>
<td>expr.ident(exprlist)</td>
<td>exprlist expr ident send</td>
</tr>
<tr>
<td>expr1.expr2</td>
<td>expr1 expr2 send</td>
</tr>
<tr>
<td>expr1.expr2(exprlist)</td>
<td>exprlist expr1 expr2 send</td>
</tr>
</tbody>
</table>

### Elements

<table>
<thead>
<tr>
<th>ident</th>
<th>ident</th>
</tr>
</thead>
<tbody>
<tr>
<td>'ident</td>
<td>/ident</td>
</tr>
<tr>
<td>&quot;string&quot;</td>
<td>(string)</td>
</tr>
<tr>
<td>number</td>
<td>number</td>
</tr>
<tr>
<td>(exprseq)</td>
<td>exprseq</td>
</tr>
<tr>
<td>[exprlist]</td>
<td>[exprlist]</td>
</tr>
<tr>
<td>expr1 [expr2]</td>
<td>expr1 expr2 get</td>
</tr>
<tr>
<td>expr1 [expr2@expr3]</td>
<td>expr1 expr2 expr3 getinterval</td>
</tr>
</tbody>
</table>
**Blocks**

<table>
<thead>
<tr>
<th>{exprseq}</th>
<th>{exprseq}</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ident1, ident2...}</td>
<td>{n dict begin</td>
</tr>
<tr>
<td>local ident3 = expr3,</td>
<td>... /ident2 exch def</td>
</tr>
<tr>
<td>ident4 = expr4,</td>
<td>/ident1 exch def</td>
</tr>
<tr>
<td>...</td>
<td>/ident3 expr3 def</td>
</tr>
<tr>
<td>identn = exprn;</td>
<td>/ident4 expr4 def ...</td>
</tr>
<tr>
<td>exprseq}</td>
<td>/identn exprn def</td>
</tr>
<tr>
<td>{... label1, label2: ...}</td>
<td>exprseq end}</td>
</tr>
</tbody>
</table>

| \{... label1, label2: ...\} | \{... /label1 /label2 ...\} |

**Control structures**

| if expr1 then expr2 | expr1 expr2 if |
| if expr1 then expr2 else expr3 | expr1 expr2 expr3 ifelse |

| for expr1 to expr2 do expr3 | expr1 1expr2 expr3 for |
| for expr1 to expr2 by expr3 do expr4 | expr1 expr2 expr3 expr4 for |
| forall expr1 do expr2 | expr1 expr2 forall |

| pathforall moves expr1 lines expr2 curves expr3 closes expr4 | expr1 expr2 expr3 expr4 pathforall |

| loop expr | expr loop |
| repeat expr1 times expr2 | expr1 expr2 repeat |
| with expr1 do expr2 | expr1 begin expr2 exec end |

| case expr1 of expr2^2 | expr1 expr2 case |
| [... label1, label2: ...] | [... /label1 /label2 ...] |

---

^2 The case operator is a NeWS extension to PostScript; in other environments it is easily provided by means of a runtime support routine.
I wish to thank the following people for their contributions to this work: My supervisor, Professor John Penny, for his encouragement and advice; my colleagues, Paul Ashton and Richard Pascoe, for many interesting discussions during research meetings and other times; and my parents, Lois and Tom, for their unfailing support.
References


223


Bentley 1975: Multidimensional binary search trees used for associative searching. *Communications of the ACM*, vol. 18, no. 9, pp. 509-517

Bentley and Friedman 1979: Data structures for range searching. *ACM Computing Surveys*, vol. 4, pp. 397-409


Cesarini F. and Soda G. 1982: Binary trees paging. *Information Systems*, vol. 7, no. 4, pp. 337-344


Douglas D. H. and Peucker T. K. 1973: Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer*, vol. 10, no. 2, pp. 112-122


Greenberg S. and Witten I. H. 1985: Interactive end-user creation of workbench hierarchies within a window system. Man-Machine Systems Laboratory, Department of Computer Science, The University of Calgary, 2500 University Drive NW, Calgary, Canada T2N 1N4


Krasner G. E. and Pope S. T. 1988: A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. ParcPlace Systems, 2400 Geng Road, Palo Alto, CA 94303


Morton G. M. 1966: A computer oriented geodetic data base and a new technique in file sequencing. IBM Ltd., Ottawa, Canada
Experiments in Interactive Map Retrieval


Samet H. 1985b: Data structures for quadtree approximation and compression. *Communications of the ACM*, vol. 28, no. 9, pp. 973-993


Smith J. L. and Abel D. J. 1985: A general purpose windowing algorithm for spatial databases. Technical Report No. 1, CSIRO, Division of Information Technology, P.O. Box 1800, Canberra ACT 2601


Stonebraker M., Rowe L. A. and Hirohima M. 1990: The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, no. 1, pp. 125-142


Sun 1987: *NeWS Technical Overview*. Sun Microsystems, Inc., 2550 Garcia Ave., Mountain View, CA 94043


Tamminen M. 1984: Comment on quad- and octrees. *Communications of the ACM*, vol. 27, no. 3, pp. 248-249

Tesler L. 1981: The Smalltalk environment. *BYTE*, vol. 6, no. 8, pp. 90-147

Vlissides J. M. and Linton M. A.: Applying object-oriented design to structured graphics. Stanford University


Williams G. 1983: The Lisa computer system. *BYTE*, vol. 8, no. 2, pp. 33-50

Williams G. 1984: The Apple Macintosh computer. *BYTE*, vol. 9, no. 2, pp. 30-54