

THE EXPERT'S VOICE®



MapServer

Open Source GIS Development

Bil Kropla

apress®

Beginning MapServer

Open Source GIS Development



Bill Kropla

Beginning MapServer: Open Source GIS Development

Copyright © 2005 by Bill Kropla

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-490-8

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Jason Gilmore

Technical Reviewers: Howard Butler and Stephen Lime

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser

Associate Publisher: Grace Wong

Project Managers: Tracy Brown-Collins and Beckie Stones

Copy Edit Manager: Nicole LeClerc

Copy Editor: Damon Larson

Assistant Production Director: Kari Brooks-Copony

Production Editor: Kelly Winquist

Compositor: Susan Glinert

Proofreader: Linda Seifert

Indexer: Carol Burbo

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit <http://www.springeronline.com>.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Downloads section.

For my children.

Contents at a Glance

Foreword	xiii
About the Author	xv
About the Technical Reviewers	xvii
Acknowledgments	xix
Introduction	xxi
■ CHAPTER 1 Building MapServer	1
■ CHAPTER 2 Simple MapServer Examples	15
■ CHAPTER 3 Creating the Mapping Application	31
■ CHAPTER 4 Modifying a Map's Look and Feel	55
■ CHAPTER 5 Using Query Mode	103
■ CHAPTER 6 Using Perl MapScript	167
■ CHAPTER 7 Using Python MapScript	187
■ CHAPTER 8 Using PHP/MapScript	207
■ CHAPTER 9 Extending the Capabilities of MapScript with MySQL	231
■ CHAPTER 10 Utility Programs	291
■ CHAPTER 11 MapServer Reference	309
■ APPENDIX	369
■ INDEX	391

Contents

Foreword	xiii
About the Author	xv
About the Technical Reviewers	xvii
Acknowledgments	xix
Introduction	xxi

■ CHAPTER 1 **Building MapServer** 1

Planning the Installation	1
Selecting Supporting Libraries	1
Getting the Software	4
Building and Installing the Software	5
Building and Installing zlib	6
Building and Installing libpng	7
Building and Installing libJPEG	7
Building and Installing FreeType	8
Building and Installing GD	9
Building and Installing Proj.4	9
Building and Installing GDAL	9
Building and Installing shapelib	10
Building and Installing MapServer	11
Configuring MapServer and Apache	12
Online Resources	13
Summary	14

■ CHAPTER 2 **Simple MapServer Examples** 15

Basic Concepts	15
Building a “Hello World” Application	16
Creating the Mapfile	16
Creating the Initialization File and HTML Template	20

Building the First Map	23
Creating the first.map Mapfile.	24
Building the HTML Template for the First Map	28
Summary	30

■ CHAPTER 3 **Creating the Mapping Application** 31

Mapfile Concepts	31
The Structure of the Mapfile	32
The LAYER Object.....	33
The CLASS Object.....	34
Mapfile Syntax	34
The Mapfile	35
Layer 1: Urban Areas	37
Layer 2: Water Features.....	39
Layer 3: State Boundaries	41
Layer 4: Road Network.....	41
The HTML Template	43
The Initialization File.....	43
The Template File.....	44
Summary	50
Code Listings	51

■ CHAPTER 4 **Modifying a Map's Look and Feel** 55

The Graphic Design of Maps	61
Labeling for Clarity	66
Fonts	66
Color.....	67
Orientation.....	67
Using Scale to Reduce Clutter	71
Classifying Features	74
Using Expressions to Define Classes	74
Using Classes	76
Using Symbols	79
Using Annotation Layers	81
Creating Scale Bars	83
Creating Legends	85
Using Reference Maps	87
Summary	88
The Code	88

CHAPTER 5	Using Query Mode	103
	How MapServer Processes a Query	103
	Query Types	104
	Query Templates	105
	Maintaining State in Query Mode	107
	Querymaps	107
	Map-Only Query Modes	107
	Query Examples	108
	Query Modes	129
	QUERY Mode	130
	NQUERY Mode	130
	ITEMQUERY Mode	131
	ITEMNQUERY Mode	131
	FEATUREQUERY Mode	131
	FEATURENQUERY Mode	132
	ITEMFEATUREQUERY Mode	132
	ITEMFEATURENQUERY Mode	133
	INDEXQUERY Mode	133
	Query Templates	133
	Map-Level Query Templates	133
	Layer-Level Query Templates	134
	Class-Level Query Templates	135
	The QUERYMAP Object	135
	The JOIN Object	136
	Substitution Strings and CGI Variables	137
	Query Substitution Strings	137
	Query CGI Variables	138
	A Query Application	139
	The Mapfile	139
	The Initialization File	145
	The HTML Template	146
	The Query Templates	150
	Summary	155
	Code Listings	156

CHAPTER 6	Using Perl MapScript	167
	Building and Installing Perl MapScript	168
	Building Perl	168
	Building Perl MapScript	169
	The Perl MapScript “Hello World” Application	169
	A Practical Perl MapScript Application	172
	Summary	181
	Code Listings	181
CHAPTER 7	Using Python MapScript	187
	Building and Installing Python MapScript	187
	Building Python	187
	Building Python MapScript	188
	The Python MapScript “Hello World” Application	189
	A Practical Python MapScript Application	191
	Summary	200
	Code Listings	200
CHAPTER 8	Using PHP/MapScript	207
	Building and Installing PHP/MapScript	207
	Building PHP	208
	Building PHP/MapScript	209
	The PHP/MapScript “Hello World” Application	210
	A Practical PHP/MapScript Application	212
	Summary	222
	Code Listings	222
CHAPTER 9	Extending the Capabilities of MapScript with MySQL	231
	Describing Application Requirements	232
	Addressing Some Design Issues	233
	Mozilla vs. IE	233
	Creating the MySQL Database	234
	Creating the Application User Account	237
	Installing the JavaScript Tool Tip Code	238
	Patching PHP MapScript	238

Building the Application	239
The Application in Action	239
Creating the Mapfile	247
The PHP Script	251
Summary	265
Code Listings	265
 CHAPTER 10 Utility Programs	291
MapServer	291
shp2img	291
legend	292
scalebar	292
sortshp	292
sym2img	293
shptree	293
shptreevis	294
tile4ms	296
shapelib	297
dbfcreate	297
dbfadd	297
dbfdump	297
shpcreate	298
shpadd	298
shpdump	298
shprewind	299
dbfcats	299
dbfinfo	300
shpcat	300
shpinfo	300
shpcentrd	301
shpdx	301
shpproj	301
GDAL/OGR	301
ogrinfo	302
ogr2ogr	304
ogrtindex	307
Summary	307

CHAPTER 11 MapServer Reference	309
Mapfile Keywords	310
Map Object	310
CLASS Object	315
FEATURE Object	320
GRID Object	320
JOIN Object	321
LABEL Object	323
LAYER Object	327
LEGEND Object	334
OUTPUTFORMAT Object	336
PROJECTION Object	340
QUERYMAP Object	341
Reference Map Object	342
SCALEBAR Object	344
STYLE Object	347
WEB Object	348
CGI Variables	350
Substitution Strings	357
APPENDIX	369
The Shapefile Specification	369
File Structure	370
Shapefile Data Structures	371
Cartographic Projections	373
Projection Categories	373
Creating and Using Symbols	381
Symbol Definition Reference	381
Creating Vector Symbols	383
FONTSET Examples	385
HTML Legends	386
INDEX	391

Foreword

Maybe it's just me, but when someone decides it's worth the effort and expense to actually write a book about something like MapServer, it's a big deal—sort of a watershed moment. So needless to say, I was very excited when I first heard about this book. At long last, I have something to show to my wife—tangible evidence that I was really doing something during those long nights in the basement (strangely enough, C code just makes her eyes glaze over—go figure).

First, a bit of history: MapServer arose out of necessity, since in the mid-nineties there were few, if any, decent commercial alternatives. Initial work centered on web-based front ends to commercial GIS software. This worked well, but was painfully slow—and I endured constant complaints from graduate students about licenses for our expensive GIS being used by the public to make maps online. There had to be a better way.

Mercifully, I found *shapelib*—which, when paired with the *GD* graphics library, brought MapServer to life (and stopped the complaints). Add a few best-of-breed open source packages (e.g., *GDAL*, *Proj.4*, *PostGIS*, and *FreeType*) and a number of talented developers—and here we sit 10 years later with a pretty powerful piece of software.

At its essence, MapServer is conceptually very simple, but unless you share the thought processes of the core developers, the learning curve can be a bit steep. For many open source projects, documentation is a weak point, and MapServer is no exception. Actually, there's a huge amount of MapServer documentation, but it's scattered loosely across mailing lists, sample applications, and websites. That's why this book is so valuable—especially to someone new to MapServer.

Bill has brought together all of the information for someone getting started with MapServer in one place—from installation to MapScript, it's all here. I appreciate the fact that he read any existing documentation and actually put it to the test, often finding syntactical inaccuracies and undocumented features. Fortunately, we all benefit from his pain and suffering.

Bill's work is more than a MapServer reference manual (see Chapter 11 and the Appendix for that stuff). It's full of numerous detailed examples—mapfiles, templates, and scripts that make learning this stuff far easier. When writing about a technical subject, perspective is everything. As a developer, I often find myself glossing over important details, such as installing the software—in the case of MapServer, however, these details are crucial.

I continue to enjoy working with this software everyday—either building applications with it or doing core development. I hope that this book will help users get up to speed quickly so they can move on to the fun part!

Stephen Lime
MapServer/MapScript Creator

About the Author



■ **BILL KROPLA** has almost 20 years of experience working in information technology, and has spent the last several years deeply involved in the wireless industry developing wireless mapping solutions for tracking shipping vehicles. Bill holds a B.Sc. in Physics from the University of Manitoba and an M.Sc. in Electrical Engineering and Applied Mathematics, also from the University of Manitoba. He also holds an omnibus patent for methods, hardware, and software used in the wireless industry.

About the Technical Reviewers

■ **HOWARD BUTLER** is a GIS software developer at the Center for Survey Statistics and Methodology at Iowa State University in Ames, Iowa. He also participates in the GDAL (www.gdal.org), ZCO (<http://zmapserver.sourceforge.net>), and MapServer (<http://mapserver.gis.umn.edu>) projects, taking on roles of developer, documenter, and advocate, depending on the situation. His spare time is spent helping out on the farm he grew up on; spending time with Rhonda, his wife-to-be; attempting to train his mentally challenged cat Deuce; and keeping his old house standing. He maintains a weblog on the intersection of open source GIS and Python at <http://hobu.biz>.

■ **STEPHEN LIME** is the original author of MapServer and MapScript, and is still one of the leaders of the MapServer development team. He has a B.A. in Mathematics from the University of Minnesota Duluth and an M.S. in Forestry from the University of Minnesota. His real job is as the Applications Development Manager for the Minnesota Department of Natural Resources. The bulk of his spare time is spent with his wife and daughters (or trying to sneak off to play golf).

Acknowledgments

I am indebted to Jason Gilmore, my editor. Without his input, this would be a much thinner (and less useful) volume. His stylistic suggestions were always on the money, and his technical expertise saved me from some embarrassing oversights. Thanks, Jason.

Thanks also for the advice of technical reviewer Howard Butler, who kept me honest. Every piece of software can be understood in one of three ways: the way the documentation *says* it works, the way the user *thinks* it works, and the way the developer *knows* it works. Howard provided that last piece, and his insights and suggestions were invaluable. His kind words about one of the maps in the book were very encouraging.

Steve Lime showed me, very gently, that I knew a lot less than I thought I did about queries. The pointers he gave me helped to remedy that problem. I am grateful for the assistance. Of course, there's a larger debt, since without Steve there would be no MapServer and no need for this book.

Having been a project manager, I know it's a thankless task. Beckie Stones chided me (with good humor) throughout the process, making sure I didn't fall too far behind, pointing out that figures really ought to have captions, and finding people who evidently had learned how not to be seen. Thanks Beckie.

There are several others who contributed to this effort, but with whom I had less contact: Damon Larson (who enforces the serial comma laws), Tracy Brown-Collins, Tina Nielsen, Kelly Winquist, and Julie Miller. Thank you all, it's been a pleasure working with you.

I would also like to thank John Fairall and John Brock for their assistance in finding a suitable image of the Çatalhöyük map.

Although this book could not have been written without the participation of these and many others I have never met (people who have worked very hard to keep me from making mistakes, stupid and otherwise), any errors or omissions are my responsibility.

Finally and most importantly, I want to thank my mother, Pat, and my children Alex and Elizabeth for their encouragement. You three make it worthwhile.

Introduction

Maps have played a prominent role in human activity for thousands of years. One of the earliest known maps was uncovered during the course of an archaeological excavation in Turkey in 1961. The Çatalhöyük site is a neolithic settlement dating from approximately 6500 BC. The map consists of a partially preserved, painted plaster representation of the community in plan view, with a smoking volcano in the background. While not drawn to scale, the map does seem to preserve the relative orientation of the structures represented. Figure 1 is a modern rendering of a portion of the original wall fragment.

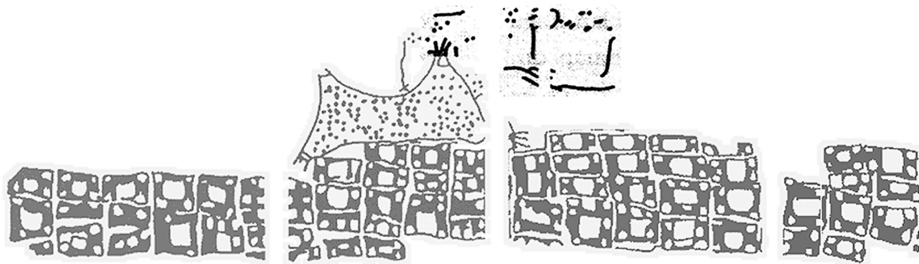


Figure 1. *Town plan of Çatalhöyük (circa 6500 BC), showing an active volcano in the background*
© John Brock, *Measure & Map* magazine, 2001

A date of 6500 BC means that people began drawing maps 3,000 years before they learned to write. The utility of maps, implied by this great age, is the result of a map's ability to present an enormous amount of information very clearly and compactly.

From the beginning, maps have been used to show where things are, but it's important to remember that the map isn't the territory. The world consists of things that have geographical (i.e., spatial) relationships with one another, and though a map can be a precise replica of the world, it's generally not. A map is a model that contains representations of the things in the world, but these representations aren't required to resemble the things they represent or even to possess the same spatial relationships. The real world is the domain of geographers and geography. Geographers live on the surface of a large, approximately oblate spheroid with topography—that is, in a three-dimensional space. The map is where the cartographer lives—it's small and flat. The process of representing real-world things on this flat surface is cartography.

The golden age of maps and cartography corresponds to the exploration age. Maps were the keys to great wealth, and you had to have a copy to get there—wherever there might be. The tools that fueled the last golden age were the compass, the sextant, and accurate clocks. Digital maps, geographic information systems (GISs), and location-based services represent the next golden age of maps and cartography. The tools fueling this age are computers, the Internet, and the Global Positioning System (GPS).

In the modern world, communicating information clearly and forcefully is critical in business, science, and politics. Whether the information to be communicated is demographic data, polling results, or environmental data, it possesses a geographic distribution. Just as graphs present numerical information in an easy-to-understand way, maps can show more clearly than any tabular format how the information relates to location, enabling the user to review this information in terms of its spatial orientation. While this distribution information is implicit in a table of numbers, it can be difficult or impossible to see.

Historically, accurate maps have been difficult to make, hard to maintain, and static. In the absence of any competing technology, these factors have limited the utility of paper maps. It's only very recently (when compared to the age of the Çatalhöyük map) that this has become clear. Road maps provide a good example of how limited maps have been. We all have them in our automobiles—and they're always out of date. They're difficult to read without squinting, and the street we're looking for is always just around the fold. And if we don't want to see streets, but rather the locations of Chinese restaurants in town, we have to buy a different kind of map. In other words, maps have tended to be dense, single-purpose documents that act as archives of past locations.

Digital maps ensure the convenient and efficient rendering of graphical images. Because of this, they can also be dynamic—showing current information in real time. But the hard cartographic work has always been (and still remains) the collection and maintenance of the underlying information. In fact, the dynamic nature of digital maps exacerbates the maintenance effort since performance and data requirements are so great.

The development of digital maps was driven by the needs of industry (e.g., mining), natural resource managers, and researchers, for a management tool. Yet, with the rise of the Internet and the commoditization of hardware, digital maps have become ubiquitous—weather maps displayed on the morning weather report, driving instructions obtained from GPS-enabled automobile navigation systems, and Internet sites that provide street maps on demand are just a few of today's commonplace digital mapping applications.

Nevertheless, most of these applications don't address the needs of the mobile user. For example, a GPS-enabled automobile navigation system can determine your present position and tell you how to get somewhere else, but since it has access to onboard data only (at best), it can't provide services that require real-time information. Such services would include optimum routing with congestion avoidance and real-time location-based services (e.g., the lowest fuel price within five miles).

However, applications are being developed that are network-aware and smart (i.e., they're wireless and GPS enabled). Some examples are management tools for GPS-enabled garbage pickup, systems to provide driving instructions to emergency vehicles, and systems that allow shippers to locate shipments in transit. Mobile technologies like WiFi, and 2.5 and 3G cellular, will bring new possibilities.

When wireless technology becomes ubiquitous and bandwidth is cheap, what will the killer app be? Prior to the existence of the Internet, no one would have predicted the popularity and profitability of a company like Google—a catalog of the contents of tens of millions of computers on the Internet, available free of charge for anyone to use. This particular application was the invention of two college students with a good idea and access to cheap technology. While I won't attempt to predict what the killer mobile application will be, the fact that it will be mobile suggests that mapping capabilities will be a necessary adjunct.

But the problem is, if some bright college student wants to put together the killer app and get rich, it could cost thousands of dollars to purchase the data and/or services required just

to get into the game. Proprietary technology, while powerful, is very expensive. Whether it's the outright purchase of proprietary software, subscription purchase of spatial information, or complete application outsourcing, the production of quality mapping applications with commercial software is costly. If you have a stable set of system requirements, some money in the bank, and a market opportunity that falls right on top of you, the proprietary options can be a good choice. Likewise, if you prefer to make development and maintenance headaches someone else's problem, or if your projected traffic volumes require heavy iron, then you'll probably want to go with the proprietary product.

But if your entrance into the market is more tentative, with a dynamic set of system requirements (or none at all); or if you're short on cash or just experimenting with the technology, you should investigate MapServer (<http://mapserver.gis.umn.edu>), which is the topic of this book. MapServer is a map-rendering engine that works in a web environment as a CGI script or as a stand-alone application via an API accessible from several programming languages. To quote from the MapServer home page, "MapServer is an OpenSource development environment for building spatially enabled Internet-web applications." Developed at the University of Minnesota with help from NASA and the Minnesota Department of Natural Resources, MapServer is now maintained by close to 20 developers from around the world.

There are a number of reasons you might consider using MapServer: maybe your boss balked at the price of a commercial product for putting maps into his pet project, and told you to find something on the Internet; maybe you have a data set that incorporates some spatial information and you want to share it in a graphical way on the Web; perhaps you'd like to expand your own pet project and you feel that providing online maps will have a lot of impact; or maybe you just have a fondness for maps and the thought of making beautiful maps from digital sources fills you with delight. But before looking at MapServer to see if it's what you need, you have to be aware of what it isn't. MapServer is a tool for rendering geographic data to the Web—it's not a full-featured GIS (although it could be used to build one). If you know what a GIS is, skip the next section and go directly to the discussion of MapServer's capabilities. If you don't know what a GIS is, read the next section so you can determine whether your project requires a GIS, or whether a map-rendering system such as MapServer will suffice.

Geographical Information Systems

While the concept of a GIS is an interesting topic in its own right, this book focuses on producing maps with MapServer. GIS concepts and vocabulary overlap a good deal with cartography, and throughout the book I explain new concepts as necessary. This section is intended to describe the basic features of a GIS so that you can determine whether you need that kind of power or not.

The first matter that comes to mind when considering the use of a GIS is the map that it produces. However, this kind of graphical output is actually the result of a chain of activities not often considered by those unfamiliar with the process. In fact, the map may not be the most important output of a GIS. The goal of any GIS is to allow the user to query and analyze spatial information, which can be useful for assisting in public policy development, in scientific research, and for making private sector business decisions (to name just a few areas of application). The information provided by a GIS can do this by producing maps that present the information of interest in a graphical way—but in many cases, the output may simply consist of tabular data that relates quantitative information to geographical location.

While there are many definitions of what constitutes a GIS, there are four capabilities that every GIS must have. These capabilities are discussed in the following section.

Data Manipulation

Data manipulation capabilities include the usual database management tools for maintaining and querying databases (extended to include spatial data), but also the ability to import or translate foreign data formats, integrate data from multiple sources into a spatial database, correct attribute or spatial data, and remove spatial data from a database (a process called thinning). Since your data is often collected from external sources and hasn't been specifically compiled to answer your questions, there are no guarantees that it will be in a format readable by your GIS. You'll need the ability to translate these files to a format your GIS can use. You may have several databases that contain the information you wish to use, but it may be inconvenient or impossible to work with all of them at the same time.

Note Real-world objects such as towns, roads, and lakes that are represented in a spatial database as geometrical objects—like points, lines, and polygons—are called features. The information (such as town names, road types, or areas of lakes) constitute the attributes of the features.

For example, you may have a spatially aware database that contains spatial information for the road network in some area. In another database, you store county land-use information. You'll probably want the ability to import the land-use information into the spatially aware database so that it becomes available for GIS use. Of course, no data source is perfect and no data set is completely clean—attribute information may contain factual errors and spelling mistakes, and spatial information can have gaps and bogus features. These mistakes must be corrected before the data is used in any analysis, so you'll need some means of seeking them out and fixing them. Spatial databases are large—sometimes so large that even optimized hardware and software may bog down when asked to render highly detailed maps or search large databases. If some features are of no interest to you and just take up space and time, an effective way of improving response is to thin the data by removing these features. A GIS should provide the functionality to do this.

Analytical Capabilities

Analysis is the real bread and butter of a GIS. The whole point of a GIS is to give the user the ability to ask questions about geographic relationships between data elements. The data elements consist of features (and their locations) and the attributes associated with those features. While a GIS can perform the usual sort of nonspatial queries (like how many lakes have areas greater than 10 square miles), the power of a GIS is really judged by how well (i.e., how easily) it provides answers to questions like:

- What feature is found at a specific location?
- Where are certain types of features located?
- What are the attributes of features in the neighborhood of a location?
- What is the spatial distribution of an attribute?
- Has the distribution changed over time?

- How does the distribution of one attribute relate to another?
- What happens if some attributes are changed?

Spatial Referencing of Attribute Information

Spatial referencing (or georeferencing) is the process of assigning real-world coordinates to features. Features derived from scanned maps or aerial photographs contain various kinds of distortion. Tools are required to scale, rotate, and translate images to remove this distortion and, once the images have been appropriately modified, assign coordinates to significant features. This is sometimes referred to as *geocoding*, but this term is also commonly used to signify the assigning of street addresses to spatial coordinates.

Note A spatial reference is the location information (essentially the latitude and longitude) associated with the geometrical objects (like points, lines, and polygons) that constitute features.

Graphical Input and Output Capability

It's often helpful to use a georeferenced image and build a map on top of it. For example, an aerial or satellite photo showing forested areas in some region may be overlaid with proposed road construction to assist in an environmental assessment. In order to do this, a GIS must be able to import and display these georeferenced images. At the other end of the process, there are two types of output that a GIS must provide: tabular and graphical. The tabular output consists of attribute information that has been selected and associated with other information based on some spatial relationship that exists between elements. It's quantitative. The graphical output from a GIS consists of the maps that it generates based on a selection of attributes. The graphical data is qualitative, but it helps to concretize the tabular data, to make the tabular data easier to understand, and to encourage sharper questions.

In summary, a GIS is a system of software components that provides the ability to maintain a spatially aware database; it provides analytical tools that enable spatial queries of the database; it allows the association of locations with imported graphical data; and it provides graphical and tabular output.

MapServer

MapServer creates map images from spatial information stored in digital format. It can handle both vector and raster data. MapServer can render over 20 different vector data formats, including shapefiles, PostGIS and ArcSDE geometries, OPeNDAP, Arc/Info coverages, and Census TIGER files.

Not all the information displayed on a map needs to be in vector format. For example, aerial or satellite photos of a region can be displayed behind rendered vector data to provide a clearer picture of how those vector elements relate to real-world features. MapServer reads two raster formats natively: GeoTIFF and EPPL7, but can read over 20 formats (including Windows bitmaps, GIFs, and JPEGs) via the GDAL package. However, although MapServer understands and can render these raster types, it has no way of tagging images with spatial information.

Note A significant distinction should be made between vector data and raster data since each is used and stored differently. A vector representation of a geometrical object essentially consists of a list of the coordinates of the points that define the object. A raster object, on the other hand, consists of a string of values that represent a digital image. A vector object contains explicit spatial references by definition; a raster object, since it's just an image, requires tags that allow it to be properly positioned, oriented, and scaled.

MapServer can operate in two different modes: CGI and MapScript. In CGI mode, MapServer functions in a web server environment as a CGI script. This is easy to set up and produces a fast, straightforward application. In MapScript mode, the MapServer API is accessible from Perl, Python, or PHP. The MapScript interface allows for a flexible, feature-rich application that can still take advantage of MapServer's templating facilities.

MapServer is template based. When first executed in response to a web request, it reads a configuration file (called the mapfile) that describes the layers and other components of the map. It then draws and saves the map. Next, it reads one or more HTML template files that are identified in the mapfile. Each template consists of conventional HTML markup tags and special MapServer substitution strings. These strings are used, for example, to specify the paths to the map image that MapServer has created, to identify which layers are to be rendered, and to specify zoom level and direction. MapServer substitutes current values for these strings and then sends the data stream to the web server, which then forwards it to the browser. When a requester changes any form elements on the page (by changing zoom direction or zoom value, for example) and clicks the submit button, MapServer receives a request from the web server with these new values. Then the cycle starts again.

MapServer automatically performs several tasks when generating a map. It labels features and prevents collisions between neighboring labels. It provides for the use of both bitmapped and TrueType fonts. Label sizes can be fixed or configured to scale with the scale of the map. The option to not print labels for specified map scale ranges is also provided.

MapServer creates legends and scale bars (configurable in the mapfile) and generates reference maps. A reference map shows the context of the currently displayed map. For example, if the region of interest is North Dakota, the reference map would show a small map of North Dakota, with the extent of the current map outlined within it. Zooming and panning are under user control.

MapServer builds maps by stacking layers on top of one another. As each is rendered, it's placed on the top of the stack. Every layer displays features selected from a single data set. Features to be displayed can be selected by using Unix regular expressions, string comparisons, and logical expressions. Because of the similarity of data and the similarity of the styling parameters (like scale, colors, and labels), you can think of a layer as a theme. The display of layers is under interactive control, allowing the user to select which layers are to be rendered. While layers can't be generated on the fly, empty layers can be populated with dynamic data and manipulated via URLs. MapServer has powerful and sophisticated query capabilities, but in CGI mode it lacks the tools that allow the kind of analysis provided by a true GIS.

This overview has described some of the features of MapServer and shown why it's not a full-featured GIS: it provides no integrated DBMS (database management system) tools, its analytical abilities are limited, and it has no tools for georeferencing.

Since MapServer's functions can be accessed via an API from various programming languages (such as PHP, Perl, and Python), it can serve as the foundation of a powerful spatially aware application that has many of the analytical and reporting functions of a true GIS. In addition, while there are no integrated tools for manipulating spatial data, there are third-party tool sets that perform many (although not all) of these functions.

When run as CGI in a web environment, MapServer can render maps, display attribute data, and perform rudimentary spatial queries. When accessed via the API, the application becomes significantly more powerful. In this environment, MapServer can perform the same tasks it would as CGI, but it also has access to external databases via program control, as well as more complex logic and a larger repertoire of possible behaviors.

Applying MapServer

What follows is a brief description of three kinds of applications that can be developed with the MapServer API. (They could also be done via CGI, but that process is slow, cumbersome, and ugly.) With the addition of a MySQL database and a programming language like PHP, these applications can provide considerable functionality without a huge development effort, because the difficult, spatially aware piece is done by MapServer. None of these are particularly innovative, but they do demonstrate the sort of tasks that can be accomplished.

Real Estate Sales Tool

By adding lat/long coordinates for each sale property to an MLS (multiple listing service) or similar service, you can create a spatially aware catalog providing the functionality that users have come to expect from graphical interfaces (such as click-and-drag spatial queries and informational boxes that pop up when mousing over hot spots).

Real-Time Track and Trace

By collecting GPS locations in real time and forwarding them back to a host via 2.5 and 3G cellular technology, MapServer can help you construct a customer-facing application that shows the actual location of a load in real time. A MySQL database would serve very well for storing this kind of data.

Real-Time Traffic Advisories and Congestion Avoidance

By collecting traffic levels electronically—or via manual entry of GPS coordinates, street addresses, or intersections—MapServer could display traffic levels in real time, make them available over the Web, and suggest alternate routes.

How This Book Is Organized

This book is an in-depth treatment of elementary MapServer. Using MapServer in CGI mode is the focus of the first five chapters.

Chapter 1 begins with a detailed description of the installation process; it identifies supporting software, shows you where to get it, and discusses options for both supporting software and MapServer. Configuration of MapServer and the Apache web server is also discussed.

Chapter 2 provides a line-by-line description of two simple examples that will help you become familiar with MapServer and its operation.

Chapter 3 introduces a more complex mapping application that you'll work through and complete by the end of Chapter 4. You'll also become familiar with the more common mapfile keywords, HTML templates, and ways to create multiple classes in a layer. The use of zoom, pan, and layer selection will also be discussed.

Chapter 4 addresses look-and-feel issues. You'll learn about labels and annotation layers, classes and regular expressions, and ways to add scale bars, legends, and reference maps.

Chapter 5 describes MapServer's native query ability. You'll learn the details of MapServer's numerous query modes by building an application that uses them all.

Chapter 6 introduces the MapServer API available in Perl, which you'll use to build a demo application using Perl.

Chapter 7 introduces the MapServer API available in Python, with which you'll build an equivalent demo application.

Chapter 8 introduces the MapServer API available in PHP. You'll use this to build a similar demo application.

Chapter 9 describes a complete mapping application using PHP/MapScript in conjunction with MySQL.

Chapter 10 describes the various utilities for manipulating shapefiles that are available both in the MapServer distribution and elsewhere.

Chapter 11 is a compact descriptive reference list of all MapServer keywords, HTML substitution strings, and CGI variables.

The Appendix provides more detailed coverage of several topics: the shapefile specification, cartographic projections, creating MapServer symbols, FONTSET examples, and HTML legends.

Prerequisites

MapServer can run in most Unix and Unix-like environments. The Unix version is a source distribution and must be compiled and installed before use. While detailed instructions will be provided to assist you, be aware that all systems are configured differently, so if something goes wrong with the compile or the install, it helps to know something about the process. If you haven't done this sort of thing before, talk to someone who has. Installing software for the first time can be tricky, but we've all had to do it.

MapServer is also available as an executable binary for Windows. The binaries were compiled on Windows 2000, but they're known to run on Windows NT and Windows 9x. If you have the tools, you might try to compile from source, but I wouldn't recommend it. Life is short.

Since MapServer is a web application, it will also be helpful to have a clear understanding of the Apache web server (<http://httpd.apache.org>). Although MapServer should be able to run in any compliant CGI environment, my own preference is to run it under Apache, and as

such, all the examples and code in the book assume you're using Apache. In a Microsoft environment, you're on your own, although Apache 2.0 should serve as at least an adequate testing environment if you're using Windows. That said, if you understand your web server and its configuration, then MapServer should operate as described in this book. Of course, you need the appropriate system access and permissions to configure your web server and create directories where needed.

Accessing MapServer functionality via its API requires the installation of one or more versions of MapScript. This book covers PHP/MapScript for PHP access (www.php.net), and SWIGMapScript for access from Perl (www.perl.com) and Python (www.python.org). You'll build an identical application in each language. It's assumed that you already possess some knowledge of these languages, since a primer is out of the scope of this book. I'd like to recommend the following books should you be seeking comprehensive guides for any of these languages:

- *Beginning PHP 5 and MySQL: From Novice to Professional*, by W. Jason Gilmore (Apress, 2004)
- *Practical Python*, by Magnus Lie Hetland (Apress, 2002)
- *Beginning Perl, Second Edition*, by James Lee (Apress, 2004)

The MapScript application will also access a MySQL database (www.mysql.com). Although instructions for setting up the database will be provided, you'll have to know how to install and configure MySQL, and you'll also need appropriate permissions for creating files and making them available to the application. Check out *The Definitive Guide to MySQL, Second Edition*, by Michael Kofler (Apress, 2003), for a comprehensive introduction to the MySQL database server.

Downloading the Code

The code for this book is available for download from the Apress website, located at www.apress.com.



Building MapServer

In this chapter, you'll learn how to build MapServer. Since much of its functionality is supported by external libraries, I'll briefly discuss the capabilities of each, and which libraries are needed to provide a basic MapServer environment. I'll present download sites for the software, and I'll also describe the build and install processes in some detail. The builds are highly automated and will usually produce a binary executable, even if you've left something out. You'll only notice that you've left something out when MapServer does something you don't expect. Since you'll probably ask the question "Is it me or is it MapServer?" many times before you gain a clear understanding of how MapServer works, getting the build right will greatly reduce your frustration level. After you've installed the libraries and built MapServer, I'll describe the configuration of the Apache web server, but only as it relates to the MapServer environment. Finally, I'll present some pointers to some online resources. It's assumed that you're familiar with the shell environment, that you understand directory structures, and that you have the authority to make library and web server changes.

You should have some familiarity with the build tools Autoconf and Make, as well as the general configuration of Apache (or know someone who does). But if this is your first time, don't despair—you have an interesting journey ahead.

Planning the Installation

Building MapServer for the first time can be a challenging experience for someone unfamiliar with Unix build environments. There are many interdependencies between the libraries, and the sequence in which the supporting libraries are created is very important. Follow the build order described in this chapter to avoid problems.

Selecting Supporting Libraries

Table 1-1 lists the external libraries used by MapServer. Minimum release level is noted if it's an issue; also noted is whether the library is mandatory or optional. Most of these libraries are optional and won't be required for an introduction to MapServer. They provide access to spatial information stored in proprietary databases, and they also provide support for WMS (web mapping service) and output formats other than PNG, JPEG, and GIF. Following the table is a brief description of each of the libraries. Later in this chapter, I'll show you how to install several of the most important libraries, followed by instructions regarding the installation of MapServer.

Table 1-1. *External Libraries*

Library	Minimum Release	Mandatory/Optional	To Be Installed?	Notes
GD	2.0.12	Mandatory	Y	
FreeType	2.x	Optional	Y	
libJPEG		Optional	Y	
libpng	1.2.7	Mandatory	Y	
zlib	1.0.4	Mandatory	Y	
GDAL	1.1.8	Optional	Y	
OGR		Optional	Y	
Proj.4	4.4.3	Optional	Y	Required for WMS
shapelib		Optional	Y	
libcurl	7.10	Optional	N	Required for WMS
SDE Client libraries		Optional	N	
PostgreSQL Client libraries		Optional	N	
Oracle Spatial Client libraries		Optional	N	
Ming	0.2a	Optional	N	
LibTIFF		Optional	N	
LibGeoTIFF		Optional	N	
PDFLib	4.0.3	Optional	N	License required

GD

GD is a library of graphics routines. Since MapServer uses GD to render images, it's mandatory to install it. Note that GD has its own list of dependencies that include zlib, libpng, FreeType 2.x, and libJPEG. These provide GD with the ability to do image compression (for those formats that support it), to render PNG (Portable Network Graphics) images, to use TrueType fonts, and to render JPEG (Joint Photographic Experts Group) images. Since the major patent underlying the GIF (Graphics Interchange Format) image format has lapsed, GIF support has been restored to GD (as of release 2.0.28)—it's available at www.boutell.com/gd.

FreeType

FreeType is a font-rendering engine. It's not referenced directly by MapServer, but rather used by GD for rendering fonts. Since TrueType fonts are more attractive than the bitmapped fonts that MapServer provides, it's worthwhile to include this library, available at www.freetype.org.

libJPEG

libJPEG is used by MapServer to render JPEG images. A new version hasn't been released since 2001. Any reasonably current release of your operating system likely has a usable version already installed. If that's not the case, it's available at www.ijg.org/files.

libpng

libpng provides a library of routines for rendering PNG images. It's not referenced directly by MapServer, but rather used by GD. libpng also requires zlib. It's available at www.libpng.org/pub/png.

zlib

zlib is a data-compression library used by GD. It's available at www.gzip.org/zlib.

GDAL

GDAL (Geospatial Data Abstraction Library) is a translator library for raster data. It provides the ability to import and project georeferenced raster images. You won't use that functionality within the context of this book, but the library is required for a basic MapServer install since it also contains the OGR library.

OGR

The OGR Simple Features Library provides access to reading and some writing of a variety of vector formats. While useful in a more sophisticated MapServer environment, in the book you'll use OGR for several utilities that it provides. GDAL (including OGR) is available at <http://gdal.maptools.org>.

Proj.4

Proj.4 is a library of cartographic projection routines. It can be accessed on the fly by MapServer or in stand-alone mode to perform projection on an entire data set. It's available at <http://proj.maptools.org>.

shapelib

shapelib is a library of C routines for creating and manipulating shapefiles. You won't be writing any C code, but you'll take advantage of several utilities that are provided in the distribution. These utilities provide the ability to create shapefiles (which include DBF files), dump the contents of shapefiles and DBF files, and change the projection of shapefiles. Some of the utilities depend on Proj.4. The shapelib library can be found at <http://shapelib.maptools.org>.

libcurl

libcurl is a client-side URL-transfer library that supports FTP, FTPS, HTTP, HTTPS, GOPHER, TELNET, DICT, FILE, and LDAP. It's required if you wish to provide WMS support. The WMS protocol is used to move map images and text data across networks. In order to keep the MapServer environment simple, you won't be installing it.

SDE Client Libraries

SDE client libraries are part of the ESRI's Spatial Data Warehouse. If you wanted to give MapServer access to this, you would compile against these libraries. But this is outside the scope of this book, so you won't need to install it.

PostgreSQL Client Libraries

PostgreSQL client libraries are required for giving MapServer access to PostGIS data. They provide functionality similar to the ESRI product, but they're open source. This is, however, outside the scope of this book, so you won't be installing it.

Oracle Spatial Client Libraries

Oracle Spatial client libraries are required for giving MapServer access to the Oracle Spatial Data Warehouse. They provide functionality similar to the ESRI product. This is also outside the scope of the book, so you won't install it.

Ming

Ming provides MapServer with the power to create SWF (Shockwave Flash) movies. While it suggests some interesting applications, it's also outside the scope of this book, so you won't be installing Ming either.

PDFLib

PDFLib provides MapServer with the ability to produce output as PDFs (Portable Document Format)—also useful, but also outside the scope of this book. In addition to that, it's not open source.

By limiting installation of these libraries, you'll reduce functionality slightly, but greatly simplify the install process. Although the MapServer environment you create will be a basic one, it will still be capable of supporting some powerful applications. After becoming more familiar with MapServer, you can download the other libraries and rebuild with the additional support. Pointers to extensive documentation for these libraries are available on the MapServer website (<http://mapserver.gis.umn.edu>).

Getting the Software

Table 1-2 lists all the software required to build MapServer. It's all freely available for download, and it's all open source. Although licensing requirements differ, only one package—FreeType—uses the GPL (GNU General Public License) explicitly. The websites (or FTP sites) specified should always have the latest versions and bug fixes available. Download the source distributions identified in Table 1-2 and put them somewhere convenient.

Table 1-2. *Where to Find the Software*

Package	Location
MapServer	http://mapserver.gis.umn.edu/dload.html
GD	www.boutell.com/gd
FreeType	www.freetype.org
libJPEG	www.ijg.org/files
libpng	www.libpng.org/pub/png
zlib	www.gzip.org/zlib
GDAL	http://gdal.maptools.org
Proj.4	http://proj.maptools.org
shapelib	http://shapelib.maptools.org

Building and Installing the Software

This section presents a detailed description of the installation process for MapServer and all the supporting libraries. The Autoconf utility will eliminate most of the tedious work by configuring each build automatically. But keep in mind that some installations may already have installed these libraries. In these cases, if the installed version has an appropriate release level, you might not have to build and install it yourself, but just give the configuration the appropriate path. Be aware that although the release level might be right, the configuration options of any pre-installed library might not comply with the requirements of MapServer. If that's the case, then you'll have to reinstall.

The development environment for the examples in this book is the Slackware 9.0 distribution of Linux with kernel release 2.4.20. It runs right out of the box with gcc (version 3.2.2), GNU Make (version 3.80), and GNU Autoconf (version 2.57). You'll be creating MapServer version 4.4.1, although later versions will build in the same fashion. The configuration options chosen for each build will be the defaults whenever possible. When there are no defaults or the defaults aren't appropriate, it will be noted and the correct values for the development environment used. This should make the build descriptions more or less portable to other Unix-like environments. It's important, however, to understand that the canonical installation documents are the README, INSTALL, and Makefile files supplied in the source distribution of each library. This book isn't a substitute for them, so be sure that you read them.

Each installation will be different, and the README and other documents discuss many environment-specific issues that are too lengthy to cover here. Most of the libraries are configure-based distributions that use the Autoconf utility, which makes the build process highly automated, but there still remain decisions concerning what options are available, which ones are appropriate, and what values they should have. In the event that configure doesn't work, you'll have to read the documents supplied with the distribution and hunt down the error.

Note that GNU Make is an absolute requirement for building FreeType—there are no substitutions allowed. If your environment doesn't contain GNU Make, install it before proceeding.

The description given here assumes you have root privileges. If you don't have root privileges, read the installation documents for alternatives.

The configuration process should be able to identify any incompatibilities and missing capabilities. The downloaded tarballs will be untarred into directories in `/usr/local/src/`. The procedure descriptions that follow assume that all the libraries will be installed from scratch to default install paths specified in their respective configuration files.

Assuming that your environment isn't unusual in some way, building each of the libraries will involve the same steps:

1. Untar the distribution tarball into `/usr/local/src/`.
2. Change directory to the directory just created in `/usr/local/src/`.
3. Read the README and INSTALL files (or their equivalents).
4. Run `configure`, with command-line options if required.
5. Run `make`.
6. Run `make check` if available.
7. Run `make install` if necessary.
8. Run `ldconfig` so your OS can locate the new libraries.

If there are no problems with missing libraries, inappropriate compilers, or permissions, this is almost all there is to do. Some libraries may require one or more command-line options, and I'll note them when necessary. But—always read the README and INSTALL files for details first. If they don't exist, go to the website or FTP site from which you downloaded the source tarball and look for documentation there. If you'd like to know what options are available to `configure`, run

```
./configure --help=short
```

In order to ensure that the dependencies of each library are met, the build sequence will be in the following order: `zlib`, `libpng`, `libJPEG`, `FreeType`, `GD`, `Proj.4`, `GDAL`, and finally `MapServer`.

Building and Installing `zlib`

Untar the `zlib` tarball as follows:

```
tar -xvzf zlib-1.2.1.tar.bz2 -C /usr/local/src/
```

Then change directory to `/usr/local/src/zlib-1.2.1/`. After reading the README file and the comments at the top of the Makefile, you'll find that the only configuration option you have to specify is whether you want to build `zlib` as a shared library. Since you do, run `configure` with the command-line option `-s`. The default install destinations, `/usr/local/lib/libz.*` and `/usr/local/include/zlib.h`, are acceptable, so you don't need to specify a path. Since your environment might differ, check first. Execute the following lines to configure, build, test, and install `zlib`:

```
./configure -s
make test
make install
```

This should execute cleanly and leave you with libraries installed in `/usr/local/lib/`. If not, you may have the wrong version of `Make` or the wrong compiler, or you don't have permissions set correctly.

Building and Installing libpng

Untar the libpng tarball as follows:

```
tar -xvjf libpng-1.2.7.tar.bz2 -C /usr/local/src/
```

Then change directory to `/usr/local/src/libpng-1.2.7/`. The `README` file indicates that `zlib 1.04` or later is required, but `zlib 0.95` may work. Since you've just installed `zlib 1.2.1`, you should have no version problems. The `INSTALL` file will tell you that you must have `zlib` at the same level as `libpng` in `/usr/local/src/` in order for `configure` to find the compiled libraries and headers (i.e., you need the `zlib` source directory to be in `/usr/local/src/`). Since that's where you just put it, no changes are required.

The `libpng` build isn't configuration based. The `scripts` subdirectory contains makefiles for a variety of environments. You have to select the appropriate one and copy it to the `libpng` root. If you're running Linux, choose `makefile.linux` and `cp scripts/makefile.linux ./Makefile`.

`INSTALL` tells you to read the `Makefile` and `pngconf.h` files for any changes. Do so, but there should be no changes required. Next, run

```
make
make test
```

This should give you the message `libpng passes test`. Check for 9782 zero samples in the output and confirm that files `pngtest.png` and `pngout.png` are the same size. This indicates that `libpng` was built correctly. If you like, you can run another test:

```
./pngtest pngnow.png
```

This should also tell you `libpng passes test`. Check for 289 zero samples in the output. If the build completes normally and the tests indicate no errors, you can now install, by running

```
make install
```

Test the install by running the following (some makefiles won't contain a target to do this):

```
make test-install
```

You should see `libpng passes test`. `libpng` is now installed. If all went smoothly, you can go on to the next step. If it didn't go well, you may have the wrong version of `Make` or the wrong compiler, or you don't have permissions set correctly.

Building and Installing libJPEG

Untar the libJPEG tarball as follows:

```
tar -xvzf jpegsrc.v6b.tar.gz -C /usr/local/src/
```

Then change directory to `/usr/local/src/jpeg-6b/`. After reading `install.doc` (there's no `README` file), you'll find that the only configuration option you have to specify is whether you

want to build libJPEG as a shared library. Since you do, invoke `configure` with the command-line option `--enable-shared`, as follows:

```
./configure --enable-shared
make
make test
```

If the test is clean, run

```
make -n install
```

Specifying the `-n` switch will cause `configure` to display the *location* where `make` will install the files, but it won't perform the actual install. If the default is correct for your environment, continue with the install. If not, add `--prefix=PATH` to the `configure` command line, where `PATH` is the appropriate path to libraries and includes; then rerun `configure` and `make`. When the install path is correct, run

```
make install
```

This should be all there is to it, but if not, you may have the wrong version of `Make`, the wrong compiler, or incorrectly set permissions.

Building and Installing FreeType

Untar the FreeType tarball as follows:

```
tar -xvjf freetype-2.1.9.tar.bz2 -C /usr/local/src/
```

Then change directory to `/usr/local/src/freetype-2.1.9/`. The `README` file should point you to `docs/INSTALL` and `docs/UPGRADE.UNX`. The `UPGRADE.UNX` file should tell you that if you're upgrading from FreeType version 2.05 or earlier, you may have some issues that must be resolved. These issues are as follows:

- Is the TrueType bytecode interpreter going to be used?
- Has the correct install path been determined?
- Has GNU `Make` been installed?

The default is to not use the bytecode interpreter. So go with the default (remember, you want to keep the install as generic as possible) and you therefore don't need to specify an option. Earlier versions of FreeType defaulted to `/usr/local/` as the install directory, but because later versions are being placed in `/usr/` more frequently, you need to determine the correct install path. You can do this with the utility `freetype-config`, by running

```
freetype-config --prefix
```

This returns the proper path (in my case, the directory `/usr/`). The `UPGRADE.UNX` file also notes that GNU `Make` is required. No other will do. If you haven't got it, get it and install it. Next, run

```
./configure --prefix=PATH
make
make install
```

If you have the wrong version of Make or the wrong compiler, or you don't have permissions set correctly, you'll have to fix things.

Building and Installing GD

Untar the GD tarball as follows:

```
tar -xvzf gd-2.0.33.tar.gz -C /usr/local/src/
```

Change directory to `/usr/local/src/gd-2.0.33/`. GD requires zlib, libpng, FreeType 2.x, and libJPEG. You should have already installed these libraries in `/usr/local/`. You can run `configure` without any options and it should find all the libraries. If your environment has preexisting copies of any of these libraries and you chose not to install new ones, `configure` should still find them. If it doesn't, read the document `README.TXT`. You must provide paths to the preexisting libraries. For example, if you're using a preexisting install of FreeType 2.x, use the `configure` option `--with-freetype=DIR`, where `DIR` is the path to the directory containing the FreeType library and header files. In my case, I would use `--with-freetype=/usr/local/`. This causes `configure` to look in `/usr/local/include/` for headers and `/usr/local/lib/` for libraries. Assuming that you're making a fresh install of everything and using the path `/usr/local/`, you don't need any options. Then run

```
./configure  
make  
make install
```

If this completes without errors, you're done. If not, go back to `README.TXT`.

Building and Installing Proj.4

Untar the Proj.4 tarball as follows:

```
tar -xvzf proj-4.4.9.tar.gz -C /usr/local/src/
```

and change directory to `/usr/local/src/proj-4.4.9/`. The install will default to subdirectories `bin`, `include`, `lib`, `man/man1`, and `man/man3`, under `/usr/local/`. If your environment is different, you'll have to tell `configure` where to install things with the option `--prefix=DIR`, where `DIR` is the install path. Since the default is fine for most environments, you can just run

```
./configure  
make  
make install
```

As always, if it fails, you've got the wrong version of Make or the wrong version of the compiler, or you don't have permissions set properly. Of course, the further you get in this process, the greater the likelihood that one of your previous library builds was bad.

Building and Installing GDAL

Untar the GDAL tarball with

```
tar -xvzf gdal-1.2.3.tar.gz -C /usr/local/src/
```

and change directory to `/usr/local/src/gdal-1.2.3/`. The install directory provides no README file, but there's a detailed install document at http://gdal.maptools.org/gdal_building.html. A default configuration should work—if not, you can run `./configure --help=short` to get some idea of the options available. If external libraries are causing problems, GDAL can use internal versions of zlib, LibTIFF, LibGeoTIFF, libpng, and libJPEG. These are specified as, for example, `--with-png=internal` or `--with-zlib=internal`. If that's no help, then go to the website. Building GDAL is just a matter of running

```
./configure
make
make install
```

and you're done.

Building and Installing shapelib

Untar the shapelib tarball with

```
tar -xvzf shapelib-1.2.10.tar.gz -C /usr/local/src/
```

and change directory to `/usr/local/src/shapelib-1.2.10/`. The shapelib build isn't configuration based. In addition, there are two builds required: building the libraries and utilities, and building the contents of the contrib directory. Editing the Makefile to specify compiler and compiler flags might be necessary, but the default should be fine for most environments. Assuming that the defaults are OK, type the following commands:

```
make
make test
```

to build and test that the utilities were created successfully. This will produce the following binaries: `dbfadd`, `dbfcreate`, `dbfdump`, `shpadd`, `shpcreate`, `shpdump`, `shpwind`, and `shptest`. Copy these to someplace useful, like `/usr/local/bin/`.

You don't need to actually install the library to use the utilities, but if you wish to, type

```
make lib
make lib_install
```

The defaults will put the libraries in `/usr/local/lib/`. The contrib directory contains several useful utilities. Change directory to `tests/` and type

```
make
```

The make target, `check`, will test the build, but there appears to be a syntax error in the script `tests/shpproj.sh`. Load this script into a text editor and look for the line

```
dbfcreate test -s 30 fd
```

This line should instead read

```
dbfcreate test -s fd 30
```

Make the change and save it, and then run

```
make check
```

You'll see the following error message, complaining about a nonexistent file:

```
rm: cannot remove 'Test*': No such file or directory
```

You can just ignore it. The last line should read

```
success...
```

This will produce the following binaries: `dbfcat`, `dbfinfo`, `shpcat`, `shpcentrd`, `shpdata`, `shpdx`, `shpfix`, `shpinfo`, `shpproj`, and `shpwkb`. Copy these to `/usr/local/bin/` to complete the installation of `shapelib`.

At this point, all required libraries should be built and installed, and you can now go on to build MapServer itself.

Building and Installing MapServer

MapServer has many configuration options. The default for many options is: if it's not specifically requested, don't do it. That means that the MapServer build will be more complicated than the libraries, in which the defaults were usually sufficient. Several libraries won't be detected by default, so you'll have to specify paths with command-line options. `configure` should find the locations of these libraries, but if it doesn't, you'll have to append the location to each option. The options are

```
--with-proj[=/usr/local/lib]
--with-ogr[=/usr/local/lib]
--with-gdal[=/usr/local/lib]
```

where the appended paths are appropriate for my environment (and of course yours may be different). In order to build MapServer with these three libraries linked, do the following:

```
./configure --with-proj --with-gdal --with-ogr
```

Then look at the file `config.log` to see if the libraries were found. If not, add the paths and run `configure` again. When you've confirmed that the libraries have been found, run

```
make
```

This will produce an executable named `mapserv` in the source directory. If all has gone well, executing `mapserv` from the command line `./mapserv` should produce the following output:

```
This script can only be used to decode form results and
should be initiated as a CGI process via a httpd server.
```

This indicates that your build configuration didn't have any gross problems that prevented the compilation of a valid executable. The MapServer executable must now be made accessible to Apache, so copy it to the script directory as follows:

```
cp mapserv /var/www/cgi-bin/
```

MapServer should now be installed. However, in order to confirm that the configuration has linked in the appropriate libraries, you'll have to produce a map. Before you can do that, though, the MapServer environment and the Apache web server must be configured.

Configuring MapServer and Apache

MapServer reads and writes files. It must know where these files are, and the permissions must be set appropriately. These files are as follows:

- **fontset.** This file tells MapServer where to find a font. Each line of the file consists of a font alias and a path to the font file. The alias is separated from the path by white space. The alias is the name by which MapServer identifies a font, but you can choose any alias you like. For example, if you wanted to use boldfaced Arial in map labels, you could refer to the alias `arialbd`. The `fontset` file would contain a line like

```
arialbd          /usr/X11R6/lib/fonts/ttf/arialbd.ttf
```

Of course, you'll probably want more than a single font to use in your maps. You'll have to construct your own font set, since different environments will have different requirements, different locations, and different available fonts. Create a file named `fontset.txt` in Apache's `DocumentRoot`, containing lines similar to the aforementioned, but with your own fonts and aliases.

- **symbolset.** MapServer has the ability to create symbols on the fly. Each symbol is defined as a sequence of coordinate pairs (using a syntax I'll describe later) in the main MapServer configuration file (called a `mapfile`). But symbols can also be collected into a symbol file. The syntax is the same, but the symbols are now available to maps based on different `mapfiles`. A symbol file `symbols/example.sym` is provided with the source distribution. Copy this file to `DocumentRoot` and rename it `symbols.sym`.
- **shapes.** The basic MapServer environment gets its spatial data from ESRI shapefiles (discussed in the Appendix). These files must be accessible to MapServer. Create a directory named `mapdata` in the directory `/home/`, which will serve to store shapefiles. Make the directory readable and executable by the webserver user (usually `nobody`), as follows:

```
mkdir /home/mapdata
chown nobody:nobody /home/mapdata
chmod u+rx /home/mapdata
```

You'll be putting some shapefiles in this directory later.

- **images.** Whenever MapServer creates a map, it saves the image (or images) to a file. This file must be accessible to MapServer and the Apache server. To enable this, create a directory named `tmp` in the Apache `DocumentRoot`. Make it readable and writable by the Apache server, as follows:

```
mkdir /var/www/htdocs/tmp
chown nobody:nobody /var/www/htdocs/tmp
chmod u+rx /var/www/htdocs/tmp
```

The configuration is as simple as that. Of course, a publicly accessible web server would require more than this. You might, for example, keep your files outside the Apache tree and just create symlinks to them. In a production environment, you'd also want to pay a lot more attention to permissions—but this is an experimental environment, so you can get away with the slack security.

Note In the example environment, the Apache DocumentRoot is `/var/www/htdocs/`.

You now have a functioning MapServer executable, you've created the appropriate directories to put shapes and images in, you know where the fonts are, and you have a set of symbols. Make sure that the program `mapserv` is copied to `/var/www/cgi-bin/`, and then start or restart your web server with

```
apachectl start
```

or

```
apachectl restart
```

and you're done.

Online Resources

The following is a list of online resources to help you with various aspects of MapServer builds, installation, and usage.

- The MapServer website at <http://mapserver.gis.umn.edu/> is a trove of useful documents describing MapServer and how to use it. The document page at <http://mapserver.gis.umn.edu/doc.html> contains the documentation describing the installation of MapServer.
- Other useful resources are the MapServer wiki at <http://mapserver.gis.umn.edu/cgi-bin/wiki.pl> and the mapserver-users mailing list. For instructions on how to subscribe to the latter, go to the MapServer support page at <http://mapserver.gis.umn.edu/support.html>.
- The Boutell site's GD page at www.boutell.com/gd tells you everything you'll want to know about GD: how to install it, how to use it, etc.
- The FreeType site at www.freetype.org (based in France), has a couple of mirrors: <http://freetype.sourceforge.net> in the United States and <http://freetype.fis.uniroma2.it> in Italy. They provide downloads and documentation.
- The libpng site at www.libpng.org/pub/png provides documentation, downloads, a FAQ, and an interesting history of PNG.
- The Independent JPEG Group site at www.ijg.org is very sparse—it has a nearly empty home page and a directory list: www.ijg.org/files.

- The Proj.4 site at <http://proj.maptools.org> provides downloads and documentation. Particularly interesting are the documents that present the details of the various projections: [proj.4.3.pdf](#) and [proj.4.3.I2.pdf](#).
- The zlib site at www.gzip.org/zlib provides documents, downloads, and definitive answers for all questions about zlib.
- The GDAL site at <http://gdal.maptools.org> provides downloads and documents. Instructions for building GDAL are found [here](#).
- The shapelib site at <http://shapelib.maptools.org> provides downloads and documents. Instructions for using shapelib are found [here](#).

Summary

A MapServer implementation depends on numerous supporting libraries to perform its functions. Selecting which libraries to install requires that some thought be given to the capabilities you want your MapServer installation to have. The goal here hasn't been to create the most powerful and complex version of MapServer, but to provide a detailed guide to a useful implementation. If you require MapServer to do more, use the process presented here as a model and select your own set of libraries. These libraries have many interdependencies, and building them can be a complicated task, so paying attention to the details is the key to a successful outcome. (If something doesn't seem to work the way it should, the first thing you should do is determine whether it was properly installed.) Once the libraries have been created, the process of building MapServer itself is pretty straightforward—a quick configure and a make and it's done.

In the next chapter, you'll produce two simple MapServer applications, starting with a very simple "Hello World" example. It requires no spatial data and takes just a few lines of code, but provides a simple test of MapServer's configuration. The other produces a real map, but it's neither interactive nor, as a map, very useful. It does, however, provide an elementary introduction to several important MapServer concepts.



Simple MapServer Examples

In the previous chapter, you built and installed MapServer, but you've yet to test it. In this chapter, you'll create two simple applications that will demonstrate that your MapServer installation is functioning correctly, as well as provide a clear understanding of the procedural flow of a MapServer application. These applications won't do very much—in fact, the first won't even produce a map—but by beginning with the simplest configurations possible, you'll avoid confusing details and focus on how the several pieces of a MapServer application work together.

Basic Concepts

Your initial goal is to produce maps in a CGI environment, in which a user accesses an Apache web server from a web browser. In this environment, Apache invokes MapServer, passing along any form variables from the browser. Using this information, MapServer generates images and a web page, which Apache forwards back to the browser. Of course, MapServer needs more than just values from the browser to create a map. In fact, a CGI MapServer web application has four components: the mapfile, the HTML initialization form, one or more HTML template files, and a spatial database.

Note A web application can serve static pages with fixed content or it can provide dynamic content by using scripts to respond to web forms, query databases, and provide other functionality. The standard that determines how such scripts interact with the web server is known as the Common Gateway Interface and is always identified by its acronym, CGI. Detailed information on CGI scripts can be found at www.w3.org/CGI.

Each of these four components will be described in this chapter. It will be assumed that the user's web browser already displays the initialization file.

MapServer, like all web applications, is based on a stateless protocol—that is, at each invocation, it only knows what the browser has just told it. Statelessness precludes the use of applications that need to do more than answer the last question asked. However, some clever coding can offer a stateful server environment and give web applications the ability to perform

more complex tasks. For example, state can be maintained between invocations by storing state information in hidden form variables, in the URL, or in cookies. But some method is needed to bootstrap the application so that it has the information it requires at its first invocation. In a MapServer context, this is accomplished by the initialization file. In a CGI MapServer application, the initialization file is a conventional HTML form with the initialization information hard-coded into form variables. Almost any value that MapServer uses can be set in the initialization file.

When MapServer is first invoked by Apache from the HTML initialization form, a form variable is used to specify the name of the mapfile (usually with a `.map` extension). It then reads the mapfile to locate fonts, symbols, templates, and spatial data. The mapfile also specifies the size of the resulting map, its geographical extent, and whether it's in GIF, JPEG, or PNG format. Having read the mapfile, MapServer then renders one or more images: the map itself, the legend and scale bar images, and perhaps a reference map. It saves these images to a location specified in the mapfile.

In order to present its results, MapServer needs to format the map and associated elements as a web page. The program itself does not create the HTML—rather, it scans an HTML template for *substitution strings*. Substitution strings can be file references, details of map geometry, layer specifications, or zoom factors, for example. They can also be current values of CGI variables such as image size, mapfile name, map extent, etc. MapServer replaces the substitution strings with the appropriate values and returns the modified HTML to the requesting browser.

In this chapter, you'll build a mapping application that demonstrates the ease of use of MapServer and shows you how the pieces fit together. Subsequent chapters will deal with some of the subtler features of MapServer (projections, tile indexes, symbols, queries, etc.) that will make its power evident.

Building a “Hello World” Application

As mentioned previously, the first mapping application won't actually produce a map—it will produce a rectangular image with a tiny dot at its center, bracketed by the words “Hello World.” The application will be very simple so that any configuration problems or errors will be easy to identify.

This application doesn't make use of many resources—it uses no spatial data, symbols, or fonts. It does, however, need a place to put its images. You'll see how to specify the images directory to MapServer and confirm that Apache can find and access this directory.

Creating the Mapfile

The mapfile defines a collection of mapping objects that together determine the appearance and behavior of the map as displayed in the web browser. It's similar in function to the Apache `httpd.conf` configuration file. Based on the same underlying geographic data, mapping applications that use different mapfiles can display maps with different features that respond differently to user actions. Although it might seem that a static configuration file would have limited functionality, the design of MapServer and the format of the mapfile allow the development of very powerful applications.

A mapfile is hierarchical. Each mapfile defines a number of other objects. These objects include scale bars, legends, map colors, map names, map layers, etc. There are many more objects defined, which will be described more fully in the next several chapters. In addition, Chapter 11 contains a comprehensive reference for MapServer keywords. The first application uses a simple mapfile, and each object will be explained as it's used.

Mapfile definitions consist of keyword-value pairs. Some values are lists of items separated by white space, and these lists must be enclosed in quotes. Single quotes and double quotes are both acceptable. Keyword values with embedded blanks *must* be quoted, but it's good practice to quote all strings. Also note that MapServer keywords are not case sensitive, but some database access methods are.

I'll present a line-by-line description of the mapfile for the "Hello World" application so you can step through the details with a minimum of confusion. Using any text editor, open a file named `hello.map`. In the development environment, this path will be `/home/mapdata/hello.map`. Then type the following lines:

```
01  # This is our "Hello World" mapfile
02  NAME "Hello_World"
```

Note Don't type the line numbers—they're included for reference purposes only.

Line 01 is a comment—you can use `#` to insert comments since MapServer ignores any text that follows a `#`. The keyword `NAME` defines the string that will be prefixed to the names of the images that MapServer creates. The longer the `NAME` string, the more unwieldy the directory listings will be—so, when naming your own mapfiles in the future, remember to keep them short. Also, keep in mind any operating system limitations on file names that might exist.

Add the following lines to `hello.map`:

```
03  SIZE 400 300
04  IMAGECOLOR 249 245 186
05  IMAGETYPE png
06  EXTENT -1.00 -1.00 1.00 1.00
```

The keyword `SIZE` in Line 03 specifies the dimensions (in pixels) of the final map image. `IMAGECOLOR` defines the background color for the map image, and the keyword `IMAGETYPE` specifies the format of the map image. In this case, it's a PNG image. The geographic extent (the rectangular area covered by the map) is defined by the keyword `EXTENT`. The rectangular area is specified by the coordinates of the opposite corners (the lower left and the upper right).

Add the following lines to `hello.map`:

```
07  WEB
08  TEMPLATE "/var/www/htdocs/hello.html"
09  IMAGEPATH "/var/www/htdocs/tmp/"
10  IMAGEURL "/tmp/"
11  END
```

Note While these paths are appropriate for the development environment used throughout this book, your paths may differ.

In order to display the map created by MapServer, you need to embed it in a web page. This is done most conveniently by creating a template that contains all the HTML tags required to display the image, provide map controls, and present other information generated by MapServer. A web object defines the name of this template file and its location. When MapServer is invoked, it reads the mapfile and renders the map. It then reads the template file and inserts its own information into places in the template specified by substitution strings delimited by square brackets. MapServer then sends the HTML to the browser. A web object is introduced by the keyword `WEB` and closed by the keyword `END`.

The `TEMPLATE` keyword specifies the name of the HTML template, using either the relative path from the mapfile or an absolute path. The `IMAGEPATH` keyword tells MapServer where to put the map images it creates. The `IMAGEURL` keyword specifies a URL that tells the browser where to look to retrieve the image. MapServer will embed the URL in the page before sending it back to the browser. Note that the `IMAGEPATH` string is an absolute path on the local host, while `IMAGEURL` specifies the location with respect to the web server's `DocumentRoot`.

MapServer now knows what kind of image to produce, what size and background color to give it, and how to display the map it creates in a web page. It doesn't yet know what to draw and how to draw it—these tasks are governed by `LAYER` objects.

A layer references a single data set and contains a set of elements that will be rendered together at a particular scale using a particular projection (projections will be covered later in the book). A layer is introduced by the keyword `LAYER` and closed by the keyword `END`.

Add the following lines to the file `hello.map`:

```
12     LAYER
13         STATUS default
14         TYPE point
```

The value of the keyword `STATUS` determines whether the layer will be rendered. Specifying the value `default` means that the layer will always be rendered.

Each layer has a geometrical type associated with it. In this example, the feature is a point (a pair of coordinates), which you're choosing in this case for simplicity. The value of the keyword `TYPE` has been set to `point`. (Layer types will be discussed in more detail later in the next chapter.)

In order to create a map, MapServer must have some spatial data. Rather than clutter the "Hello World" map with complicated real-world data, an artificial point will be constructed with coordinates (0.0,0.0).

Add the following lines to `hello.map`:

```
15         FEATURE
16             POINTS 0.0 0.0 END
17             TEXT "Hello World"
18         END
```

The **FEATURE** keyword specifies an inline geographical feature. Instead of reading records from a spatial database, **FEATURE** allows the creation of “features” on the fly. The **FEATURE** keyword can only be used inside a **LAYER** object, and must be terminated by the keyword **END**. A feature is specified by the list of its coordinates.

The **POINTS** keyword describes this list of coordinate pairs. The values are separated by white space. Obviously, there has to be an even number of individual values. This list can represent a single point (if it contains only a single coordinate pair), or it can represent a line (if it contains more than one). If the first coordinate pair is the same as the last pair, then the list can represent a polygon (since equating the first and last points closes the figure). The list is terminated by the keyword **END**.

The **TEXT** keyword specifies the text string that will be used to label this feature. Again, if it contains spaces, it must be put inside quotation marks.

Add the following lines to `hello.map`:

```
19          CLASS
20              STYLE
21              COLOR 255 0 0
22              END
```

Within each layer, one or more classes are defined. A default class with no specified selection criteria will select every element in the data set for rendering. If selection criteria are specified, then only items that meet the criteria will be rendered for that class. The labels, line styles, marker types, and color used to render a feature are all defined at the class level.

The **STYLE** object defines the characteristics of the symbol used to draw features in this class. For simplicity, only a color is defined in this case. The **STYLE** object is terminated by the keyword **END**.

The **COLOR** keyword determines the color in which the feature is drawn by specifying its RGB components. These are integer values in the range of 0 to 255. Here, the feature has been rendered as a red dot with the default size of 1 pixel.

A class can also contain a **LABEL** object. The **LABEL** object is rendered with the class, and specifies the font type, size, and color of the label. Labels can be more complex than this, and they’ll be discussed in more detail later in the book. A label is introduced by the keyword **LABEL** and closed by the keyword **END**.

Add the following lines to `hello.map`:

```
23          LABEL
24              TYPE bitmap
25          END
```

The keyword **TYPE** determines the type of font used to render the label. There are two possibilities: bitmapped and TrueType. Bitmapped fonts are generated internally and don’t need outside references. TrueType fonts must be installed and identified by an alias found in the file specified by the **FONTS** keyword. For simplicity, this example uses bitmapped fonts. Note that the default color of the label is black—it can of course be drawn in a different color, but for now the default is simpler.

Add the following lines to `hello.map` to terminate the class, the layer, and the mapfile itself:

```
26         END # end class
27     END # end layer
28 END # end mapfile
```

The complete mapfile `hello.map` should look like this:

```
01     # This is our "Hello World" mapfile
02     NAME "Hello_World"
03     SIZE 400 300
04     IMAGECOLOR 249 245 186
05     IMAGETYPE png
06     EXTENT -1.00 -1.00 1.00 1.00
07     WEB
08         TEMPLATE "/var/www/htdocs/hello.html"
09         IMAGEPATH "/var/www/htdocs/tmp/"
10         IMAGEURL "/tmp/"
11     END
12     LAYER
13         STATUS default
14         TYPE point
15         FEATURE
16             POINTS 0.0 0.0 END
17             TEXT "Hello World"
18         END # end feature
19     CLASS
20         STYLE
21             COLOR 255 0 0
22         END
23         LABEL
24             TYPE bitmap
25         END
26     END
27 END
28 END
```

The structure of the mapfile just shown is very simple, and the map that it creates isn't really a map at all. But it should render a image with a label and display it on a web page. Before that can be done, however, the initialization and template files need to be created.

Creating the Initialization File and HTML Template

In order to interact with a user via the web interface, MapServer requires some straightforward HTML. This HTML serves three purposes:

1. It initializes the MapServer application when first invoked.
2. It formats the map and associated information in an effective manner.
3. It maintains state by saving parameters in input fields.

While the initialization task can be performed by a stand-alone web page, it's simpler in many cases to embed initial values in the template file and update these values in subsequent invocations. In keeping with the goal of absolute simplicity in this chapter, the initial values are embedded in the template file.

Using a text editor, open the file named `hello.html` in `/var/www/htdocs/` (or whichever directory is specified as `DocumentRoot`). Then type the following lines:

```
01  <html>
02  <head><title>MapServer Hello World</title></head>
03  <body>
04      <form method=POST action="/cgi-bin/mapserv">
05          <input type="submit" value="Click Me">
06          <input type="hidden" name="map" value="/home/mapdata/hello.map">
07          <input type="hidden" name="map_web_imagepath"
           value="/var/www/htdocs/tmp/">
08      </form>
09      <IMG SRC="[img]" width=400 height=300 border=0>
10  </body>
11  </html>
```

Load the page by typing its URL into your browser:

`http://localhost/hello.html`

and press Enter. If you get the following error (or something similar):

An error occurred while loading `http://localhost/hello.html`:
Could not connect to host localhost

you've forgotten to start your Apache server. To start it, type

`apachectl start`

Apache retrieves the page (without invoking MapServer) and forwards it to the browser for rendering. A submit button displaying the words "Click Me" should appear, along with a broken image icon, as shown in Figure 2-1. Notice the `IMG` tag in Line 09. It identifies the image source as `"[img]"`. This is not a legitimate image URL—when the browser tries to render the image (pictured in Figure 2-1), it chokes on the broken tag.

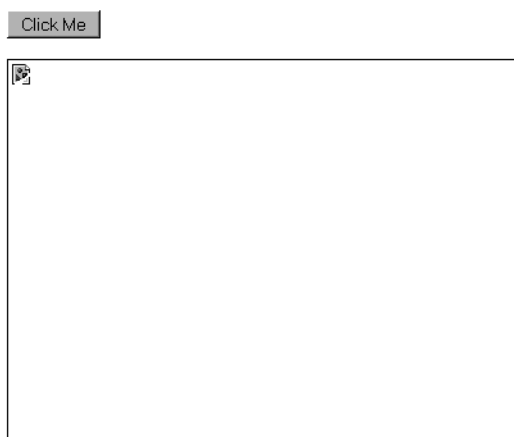


Figure 2-1. *A broken image*

Click the submit button. This time, Apache invokes the MapServer executable and passes it the form variables from Lines 06 and 07. The variable names `map` and `map_web_imagepath` are meaningful to MapServer. They identify the mapfile and image path to use on the server.

MapServer now reads the mapfile `hello.map` that was created previously. MapServer checks to see if the feature defined in the mapfile is within the defined extent. Since it is, it renders the point and produces an image. MapServer creates the name of the image by concatenating the name specified in the mapfile (i.e., `Hello_World`), a system-generated number, and the image-type extension. Since the path to the image is given by the value `map_web_imagepath`, the image file that MapServer saves looks something like

```
"/var/www/htdocs/tmp/Hello_World11008505275638.png"
```

Next, MapServer reads the template file. In this case, it's the same as the initialization file: `hello.html`. The string `[img]` is recognized as a substitution string that should expand to the URL of the image. MapServer knows the base URL is given by the value of `IMAGEURL` specified in the mapfile, so it substitutes `/tmp/Hello_World11008505275638.png` for the string `[img]`.

After substitution, Line 09 looks like

```
09      <IMG SRC="/tmp/Hello_World11008505275638.png"
        width=400 height=300 border=0>
```

After scanning and replacing any substitution strings (there's only one in this example), MapServer sends the contents of the template file (including a modified Line 09) to Apache for forwarding back to the browser. The browser receives the string and parses and renders it. The image is retrieved from the specified URL and displayed. The web page still shows a submit button, but instead of a broken image icon, the image is displayed. If everything is configured and typed correctly, you'll see a colored rectangle, 400 pixels wide by 300 pixels high, with a tiny red dot at its center. The dot is bracketed by the words "Hello" and "World." It should look like the image in Figure 2-2.



Figure 2-2. The “Hello World” image

Common errors at this stage could involve incorrectly set permissions on the directories created earlier, typos in the mapfile or template file, or forgotten libraries for rendering PNG images. Be aware that the geographical extent specified is strictly observed by MapServer. If the feature at location (0.0,0.0) isn’t within the extent, then the point won’t be rendered. In this case, the extent is bounded by the rectangular area (-1.00,-1.00), (1.00,1.00), so the point is within the extent and will be drawn.

Building the First Map

The mapping application you’ll be creating in this section is more complex than the previous and will produce an actual map. Most of the elements used in the “Hello World” application will be used, along with a few new ones that are required for using actual spatial data. The application will still be fairly simple since too much detail can be overwhelming at this stage. If you got the “Hello World” application to function correctly, you can assume that Apache knows where to find map images and has the permissions to read and write them. The next step is to put some spatial data into the data directory and see if MapServer can find it. This will also give you some more practice with the concepts of initialization files, mapfiles, and template files.

The spatial data used for the examples in this chapter comes from <http://nationalatlas.gov/atlasftp.html>. The data consists of five data sets, which are, in brief:

1. Cities and towns in the United States, Puerto Rico, and the US Virgin Islands (citiesx020.tar.gz).
2. Major roads in the United States, Puerto Rico, and the US Virgin Islands (roadtr1020.tar.gz).
3. State boundaries of the United States, Puerto Rico, and the US Virgin Islands (statesp020.tar.gz).

4. Polygon and line water features of the United States, Puerto Rico, and the US Virgin Islands (hydrogm020.tar.gz).
5. Urban areas of the United States, Puerto Rico, and the US Virgin Islands (urbanap020.tar.gz).

Download these tarballs from <http://nationalatlas.gov/atlasftp.html> to a convenient location on your computer. These data sets are all in ESRI shapefile format (shapefiles are discussed in more detail in the Appendix). For this example, you'll use only the third data set, state boundaries—however, go ahead and untar them all to the data directory. Run the following commands:

```
tar -xvzf citiesx020.tar.gz -C /home/mapdata
tar -xvzf roadtrl020.tar.gz -C /home/mapdata
tar -xvzf statesp020.tar.gz -C /home/mapdata
tar -xvzf hydrogm020.tar.gz -C /home/mapdata
tar -xvzf urbanap020.tar.gz -C /home/mapdata
```

There should be several files named statesp020.* These constitute the shapefile that contains the spatial information and attributes of the states.

Note The term *shapefile* is something of a misnomer since a shapefile actually consists of three files that share the same base name and are distinguished by file extension. One component contains spatial information. Each geographical feature in the data set is represented by a single record in this file that specifies the geographical coordinates of the feature. This file has an extension of .shp. An index file with an extension of .shx is used to access this file. For each feature in the .shp file, there's a record in the index file that contains the byte offset of the start of the feature. Finally, a file with extension .dbf stores the attribute information associated with each spatial feature, with one record per feature. This file is in dBase III format. Of course, there's more to shapefiles than stated here, but these details are addressed in the Appendix.

Creating the first.map Mapfile

Using a text editor, open the file named first.map. Then type the following lines:

```
01    # This is our first mapfile
02    NAME "First"
```

Again, the mapfile begins with a comment, which is ignored by MapServer. The NAME that will form the base for all the image files created by this mapfile is "First."

Add the following lines to the file first.map:

```
03    SIZE 400 300
04    IMAGECOLOR 255 255 255
05    IMAGETYPE JPEG
```

These lines specify the image size, background color, and image type. The JPEG image format has been selected, but other possible values are GIF, PNG, and WBMP (wireless bitmap).

Add the following lines to the file first.map:

```
06     SHAPEPATH "/home/mapdata/"
07     EXTENT -125.00 20.00 -65.00 50.00
```

The SHAPEPATH keyword in Line 06 tells MapServer where to find the directory containing the shapefiles required to render the map. This directory may also contain subdirectories. The geographic extent of the map is specified here with the keyword EXTENT. The geographic extent stretches from 125° west, 20° north to 65° west, 50° north.

Add the following lines to the file `first.map`:

```
08     WEB
09     TEMPLATE '/var/www/htdocs/first.html'
10     IMAGEPATH '/var/www/htdocs/tmp'
11     IMAGEURL '/tmp/'
12     END
```

As before, MapServer needs to know the template file, the image path, and the image URL.

You now have to tell MapServer what to render, which you do with the LAYER object. This layer is no more complex than the one from “Hello World,” since you’re just substituting a reference to a shapefile for a FEATURE object.

Add the following lines to the file `first.map`:

```
13     LAYER
14         NAME "US States"
15         STATUS default
16         TYPE line
17         DATA "statesp020"
18         LABELITEM "STATE"
```

The NAME keyword specifies the name of the layer. This provides the link between the layer and the web page. By embedding the layer name in an CGI form input field, it’s possible to interactively specify which layers to display. This is discussed in more detail in the next chapter. The NAME is limited to 20 characters.

The DATA keyword identifies the base name (i.e., without the extension `.shp`) of the shapefile to be rendered in this layer. The value associated with keyword DATA is actually the relative path from the SHAPEPATH noted previously.

The STATUS of a layer can assume one of three values: OFF, ON, or DEFAULT. If a layer’s STATUS is ON, the layer will be rendered; if it’s OFF, then the layer won’t be rendered. If STATUS is either OFF or ON, it can be changed to its opposite by the appropriate response from the web form. However, a STATUS of DEFAULT is permanently on—it can’t be turned off.

Each layer has an associated type that is specified by the keyword TYPE. The layer type determines how MapServer interprets the spatial data associated with the layer. The values associated with the keyword TYPE are explained in the following list:

- **point.** A point layer is used to render spatial data as isolated points—for example, the location of a city or other point of interest. Each feature in a point layer is rendered as a single symbol with a specific size and color.
- **line.** A line layer is used to render a series of points as a connected sequence—for example, a road or river. Successive points are joined with a line of a particular size and color.

- **polygon.** A polygon layer is used to render a series of points as an area-enclosing figure. It's distinguished from a line in that its first and last vertices must be the same. Since a POLYGON encloses an area, it has a fill color.
- **annotation.** An annotation layer is used to label features but not render them. That is, it processes the shapefile that determines where labels will go—it renders the label, but does not render the point, line, or polygon.
- **raster.** A raster layer renders a georeferenced image and gives the map maker the ability to embed vector data in a real-world context. This might be an aerial or satellite photograph, or an image that indicates color-coded elevation. In fact, any geographically distributed information can be used to create a tagged image that's rendered as a raster layer.
- **query.** A query layer is used to associate a mouse click on the map image with a specified data set. A query layer is not drawn, but its attributes can be queried. It can be used, for example, to reduce the amount of detail drawn on a map but still provide the user with the ability to query the map based on the underlying attributes of the spatial data set.

Note The layer type need not correspond with the type of the shapefile, but it must be compatible with it. Remember that a point is a single pair of coordinates, a line is a list of coordinate pairs, and a polygon is a list of coordinate pairs with the first pair the same as the last. As such, a polygon and a line can both be rendered as a series of disconnected points, and a polygon can also be rendered as a line that intersects itself. However, it's impossible to render a point as a line (or a polygon), since you need two points to define a line (and four to define a polygon).

The shapefile used in this example is a polygon. However, the layer will be defined as a line type in order to render only the state boundaries. This will cause the labels to be located at the edge of each state (one remedy for this involves using an annotation layer, which will be presented in Chapter 4).

Associated with a shapefile is a database file. Each feature in a shapefile has a corresponding record in the database. This record contains descriptive information associated with the feature. Each column of this database has a name. By specifying a column name as the value for LABELITEM, the contents of that column will be associated with the labels drawn for each feature of that layer.

It's unlikely that every feature in a shapefile will have the same significance. For example, graphically distinguishing major highways from residential streets will enhance the visual appearance and utility of your map. In order to accomplish this, classes are defined for the features in a shapefile. However, for simplicity's sake, there's only one class in `first.map`. Later, others will be added to show the utility of classes. A class is introduced by the keyword `CLASS` and closed by the keyword `END`.

Add the following lines to the file `first.map`:

```

19         CLASS
20             STYLE
21                 COLOR 0 0 0
22             END

```

This default class will cause MapServer to select every feature for rendering. The STYLE object contains the parameters that describe the way in which the symbol for this class will be drawn. It begins with the keyword STYLE and is terminated by the keyword END. In the present case, you'll use the default symbol (a 1-pixel-wide line) and color it black.

A class can also contain a LABEL object. The LABEL object is rendered with the class and specifies the font type, size, and color of the label. Labels can of course be more complex than this, which is a topic that will be discussed later. A label is introduced by the keyword LABEL and closed by the keyword END.

Add the following lines to the file `first.map`:

```

23             LABEL
24                 COLOR 0 0 0
25                 SIZE SMALL
26             END

```

The label will be rendered in the color specified by the keyword COLOR, and in the default font, at a size specified by the keyword SIZE (in this case, SMALL).

Add the following lines to the file `first.map`:

```

27         END
28     END
29 END

```

The complete mapfile `first.map` should look like this:

```

01  # This is our first mapfile
02  NAME "First"
03  SIZE 400 300
04  IMAGECOLOR 255 255 255
05  IMAGETYPE JPEG
06  SHAPEPATH "/home/mapdata/"
07  EXTENT -125.00 20.00 -65.00 50.00
08  WEB
09      TEMPLATE '/var/www/htdocs/first.html'
10      IMAGEPATH '/var/www/htdocs/tmp'
11      IMAGEURL '/data/tmp/'
12  END
13  LAYER
14      NAME "US States"
15      STATUS default
16      TYPE line

```

```

17      DATA "statesp020"
18      LABELITEM "STATE"
19      CLASS
20          STYLE
21              COLOR 0 0 0
22          END
23          LABEL
24              COLOR 0 0 0
25              SIZE SMALL
26          END
27      END
28  END
29  END

```

The structure of this mapfile isn't much different from the first example, and the map that it creates has some significant limitations (for example, it renders only state outlines and labels, and all the outlines are rendered the same way—as a 1-pixel-wide black line), but it demonstrates how little is needed to produce a map from real data.

Building the HTML Template for the First Map

The new template file is as simple as the “Hello World” template, and this second attempt won't push the boundaries of MapServer's capabilities.

Using a text editor, open the file named `first.html`. Then type the following lines:

```

01  <html>
02  <head><title>MapServer First Map</title></head>
03  <body>
04      <form method=POST action="/cgi-bin/mapserv">
05          <input type="submit" value="Click Me">
06          <input type="hidden" name="map" value="/home/mapdata/first.map">
07          <input type="hidden" name="map_web_imagepath"
08              value="/var/www/htdocs/tmp/">
09      </form>
10      <IMG SRC="[img]" width=400 height=300 border=0>
11  </body>
12  </html>

```

The only differences from `hello.html` are the title text and the name of the mapfile. The operation of this application is also very similar.

Load the page by typing its URL into your browser:

`http://localhost/first.html`

and press Enter.

Apache retrieves the page (again, without invoking MapServer) and forwards it to the browser for rendering. A submit button labeled “Click Me” will be displayed, along with a broken image icon (see Figure 2-3).



Figure 2-3. *The combined HTML initialization and template file for the first map, the broken image icon indicating that MapServer hasn’t created a map image yet*

The IMG tag in Line 09 is still broken because MapServer hasn’t made any substitutions in this file yet. Click the submit button and Apache will invoke MapServer and pass it the form variables from Lines 06 and 07. This time, the map values point to a different mapfile. Reading the mapfile `first.map`, MapServer retrieves the shapefile `statesp020` and renders all the features that lie within the extent specified (that is, between 125° west, 20° north and 65° west, 50° north). At the same time, it checks the column in the database with heading `STATE` and renders the contents as the label for each feature. It saves the map image it creates in a file with a name similar to the following: `/var/www/htdocs/tmp/First11078305275638.jpg`.

Next, MapServer reads the template file (in this case, it’s the same as the initialization file `first.html`) and substitutes the URL to the image for the `[img]` substitution string. The value for `IMAGEURL` in the mapfile is `/tmp/`, so it substitutes `/tmp/Hello_World11008505275638.jpg` for the string `[img]`.

After substitution, Line 09 of `first.html` now looks like

```
09      <IMG SRC="/tmp/First11078305275638.jpg" width=400 height=300 border=0>
```

MapServer then sends the contents of the template file to Apache for forwarding back to the browser. The browser receives the string, parses it, and renders it. The image is retrieved from the specified URL and displayed. The web page now shows a submit button and a map of the United States, with the states labeled with their names (shown in Figure 2-4). Since the layer was rendered as a line layer, the labels are located close to the line. If this had been made a polygon layer, the labels would have been located at the center of each polygon.

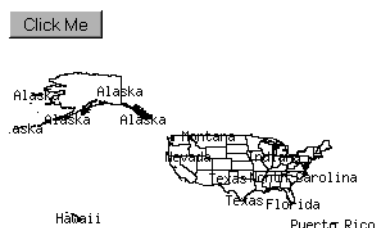


Figure 2-4. *The image that replaces the broken image icon*

If you don't see an image similar to that in Figure 2-4, check for the following errors: missing or incorrectly named shapefiles, typos in the mapfile or template file, or forgotten libraries for rendering JPEG images.

Summary

In this chapter, you've created two MapServer applications. While neither was very useful as maps, they allowed you to confirm that MapServer was properly installed. There are two issues here. The first concerns whether you've built a functioning MapServer binary. This is, of course, only the first hurdle. After building MapServer, you have to tell it where everything is located and make sure all the permissions are set correctly. Since a more useful MapServer application can easily be 20 times the size of those you've just created (in terms of lines of code), and can access a dozen HTML templates, keeping the application simple makes for an easier debug when something goes wrong.

You've also been introduced to the basic MapServer operations (in a CGI context) and seen how the pieces fit together. By limiting complexity, the entire application and all its details can be understood at once, which produces a much shallower learning curve.

In the next chapter, you'll construct a more interesting mapping application. It will be much more sophisticated and demonstrate some of the power of MapServer as a map-rendering engine. You'll see how to use layers and classes to display information more effectively. The application will be interactive and allow users to zoom and pan the map image. You'll see how to create scale bars and legends, and how to embed useful information (like the scale of the map and mouse-click coordinates) in the HTML output. In other words, you'll be creating your first "real" mapping application.



Creating the Mapping Application

In the previous chapter, you built a MapServer application that displayed a map of the United States, showing the outline of each state and using state names as labels. That map had limited utility—it showed no cities, highways, rivers, or lakes. There was no way to change the scale of the map or display different regions (no zoom or pan). Some states lacked labels, and when states *were* labeled, the labels were all located along the southern boundary of the state, which was confusing. Some of these deficiencies will be addressed in the mapping application that you'll build in this chapter, and the rest will be resolved in the next as new concepts are introduced.

In order to produce a more useful (and visually pleasing) map, you'll have to increase the complexity of both the mapfile and the HTML template. Fortunately, you'll achieve a great increase in usability from changes that will require the introduction of just a few new concepts. At the end of this chapter, you'll be able to create a mapping application that contains a significant amount of information—and the display of this information will be under interactive control. The map will show urban areas, major roads, rivers, and lakes; you'll create zoom and pan controls, and controls to select which layers to render. You'll learn to render labels more effectively and use color to differentiate between things like interstate highways and major roads.

In the previous chapter, you learned the fundamental concepts involved in rendering maps to the screen. The pace was pretty fast, but the intent was to help get you up and running quickly. Consequently, some topics were only glossed over, and the linear presentation didn't allow for a broad discussion of MapServer syntax and usage. Since the application described in this chapter is a bit more complex than the first map and requires a somewhat longer mapfile, the manner of presentation in this chapter will change slightly. Instead of plunging directly into a line-by-line analysis of the mapfile and template, I'll take a more conceptual approach. Once the general ideas have been presented, you'll begin a detailed investigation of the code. To aid you in this, a comprehensive description of mapfile keywords, HTML template substitution strings, and CGI variables can be found in Chapter 11.

Mapfile Concepts

The mapfile is the object that MapServer uses to define a CGI-based mapping application. It determines not only the look and feel of the map, but also how MapServer behaves when invoked

by the web server. In order to function properly, MapServer must understand how to handle dozens of mapfile keywords, CGI form variables, and substitution strings. The complexity can be overwhelming the first time you try to build a real application, but the logical structure of the mapfile and MapServer's straightforward processing will help you to overcome this impediment. This section introduces the mapfile. In subsequent sections, form variables and substitution strings will be discussed.

The Structure of the Mapfile

The mapfile consists of a hierarchy of objects. At the top of the hierarchy is the map object (i.e., the mapfile itself). The map object contains both *simple* and *structured* items. The simple items consist of keyword-value pairs, and the structured items contain other items, each of which can be either simple or structured. You've already seen examples of both types of mapfile constructs—for example, the file `first.map`, discussed in Chapter 2, contains the following lines:

```
NAME "First"
EXTENT -125.00 20.00 -65.00 50.00
```

Each of these lines contains simple mapfile objects. You'll also notice that each of these keywords specifies a value that only makes sense for the map as a whole. For example, the keyword `NAME` sets the name of the map to `First`. This keyword is used at the map level to specify the string that identifies all output files generated by MapServer in the course of producing a map from this mapfile. Note, however, that the same `NAME` keyword can be used at other levels as well—its function depends on where it's used. Similarly, the keyword `EXTENT` sets the extent for the whole map, so it must also be defined at the map level of the hierarchy. But like `NAME`, it too can be used at a lower level.

Note If you're new to MapServer, the use of the same keyword (like `NAME`, `TEMPLATE`, or `COLOR`) at different levels in the mapfile can be confusing. When you become familiar with mapfile concepts, this won't present a problem, and you'll in fact be grateful that the developers chose not to define a different keyword for a similar concept at each different level.

The file `first.map` also contains the following lines:

```
WEB
  TEMPLATE '/var/www/htdocs/first.html'
  IMAGEPATH '/var/www/htdocs/tmp'
  IMAGEURL '/tmp/'
END
```

This is an example of a structured object. The `WEB` object determines which HTML templates MapServer will use, and where the templates are located. The `WEB` object is generally used to determine how MapServer responds to web requests, and can contain more keywords than shown here. Since the `WEB` object defines things used to display the entire map, it makes sense that it's specified at the map level. Again, however, the keyword `TEMPLATE` can be used at a lower level, where its function is very different. (I'll show you how different in Chapter 5.) There are

several more MapServer objects defined at the map level, but for now I'll introduce just one—the LAYER object.

The LAYER Object

The notion of a *layer* is crucial to all digital mapping processes. In brief, a layer is a selection of features that comes from a single spatial data set, all drawn at the same scale. Let's break this definition down into its components to see what it means in practice.

Most (though not all) effective maps are scaled representations of the real world. The scale of the map might be 1:1,000,000, which means that 1 inch (or mile or meter) on the map is equal to 1,000,000 inches (or miles or meters) on the ground. With these sorts of maps, all features are rendered at the same scale in order to maintain relative sizes and distances.

Note An excellent example of a map that's *not* a scaled representation is the Tube Map of London, England (see www.tfl.gov.uk/tube/images/desktop_1024x768.jpg). This schematic representation of the London subway system has no associated scale and doesn't need one, since the *sequence* of subway stations is more important than the precise distance between them.

Every spatial feature has a type, which determines how the elements of the feature are stored, retrieved, and rendered. The types recognized by MapServer are POINT, LINE, and POLYGON. Fundamentally, the only property a point possesses is location. This location can be represented on the map by a symbol with a specific size and color. A line feature (consisting of an ordered sequence of points) can be represented by a line with a particular size (i.e., width) and color, but a line can also possess a style (dashed, for example). A polygon (an ordered sequence of points that encloses an area) can make use of all of these, but since it encloses an area, a polygon can have a fill color as well. Because of these differences in structure and rendering requirements, a layer can contain only one feature type.

Finally, consider a spatial data set that contains many features (all of the same type) that differ in significance depending on the scale of the map (bicycle paths versus interstate highways, for example) or the needs of the user. In some cases, it would be appropriate to render some features but not others (interstates but not bicycle paths), and in other cases, the selection might be reversed. The use of layers makes the task of generating maps more flexible by allowing the components of the map to be assembled in different ways to meet different needs.

As you've already seen, MapServer defines its layers by means of the structured LAYER object. This object specifies the data set to be rendered, the layer type, and the layer status (i.e., whether the layer should be rendered or not). Other optional layer characteristics (such as layer name, maximum or minimum scales at which the layer should be rendered, and labeling information) can also be supplied.

MapServer renders its layers in sequence, starting at the first layer specified in the mapfile. The map image isn't created by combining a separate image from each layer—rather, each layer is rendered on top of its predecessor until the last specified layer is drawn. This means that features rendered from layers defined higher (closer to the beginning) in the mapfile can be overwritten by features defined further down.

Every mapfile requires at least one layer. The LAYER object tells MapServer *what* to render, but another structured object, the CLASS object, contains the instructions that tell MapServer *how* to render features in a layer.

The CLASS Object

All the features in a spatial data set that are rendered in a layer must be of the same geometrical type. However, not every point, line, or polygon in a data set needs to have the same significance in a cartographic sense—a map ought to distinguish roads from bicycle paths to avoid (perhaps catastrophic) confusion, for example. In other words, features are classified according to specific criteria, and features that fall into one class should be drawn differently from features that fall into a different class. In order to accomplish this goal, MapServer uses the CLASS object to select the features to be rendered and to specify the way they're rendered.

Every layer requires at least one class, and selection criteria need not be specified. If they're not specified, all features in the data set will be included in the class by default. MapServer uses the CLASS-level keyword EXPRESSION to specify the selection criteria. This can be done in several ways, which are described below. A class's STYLE object contains the rendering information (such as symbol size and color) for the class. Multiple classes can be defined in a layer, and each feature of a class can be drawn in a different style. In addition to rendering features, the CLASS object is also responsible for labeling features. It performs this task by means of the LABEL object, which contains such information as font, color, and size.

Mapfile Syntax

As you've seen, the syntax of the mapfile is relatively straightforward. Generally speaking, MapServer keywords and values aren't case sensitive. In this book, however, keywords are always displayed in capitals, and values in lowercase. This practice is observed simply for clarity, and isn't required. However, you should note that case may be important when you're interacting with spatial data sets. For instance, the attribute names in the underlying database might be case sensitive, and therefore expressions containing references to attributes would also be case sensitive.

Note Version 4.4.1 of MapServer appears not to be case sensitive as far as attribute *names* in shapefiles are concerned, despite the statement to the contrary in the “MapFile Reference - MapServer 4.4” document (<http://mapserver.gis.umn.edu/doc44/mapfile-reference.html>). But it's still good practice to assume that they're case sensitive (and of course, attribute values are still case sensitive).

Strings that contain embedded blanks must be quoted. Single or double quotes are both acceptable, but they must be used in pairs—you can't quote a string using two different quote characters.

The Mapfile

At this point, the fundamental ideas of map, layer, and class should be clear, and you should understand how the pieces of a mapfile fit together. You've constructed two working mapfiles (`hello.map` and `first.map`), and you're now ready to proceed to an application that actually does something. The application you build will provide full pan and zoom capabilities, as well as the ability to turn layers on and off. Figure 3-1 shows what this map will look like.

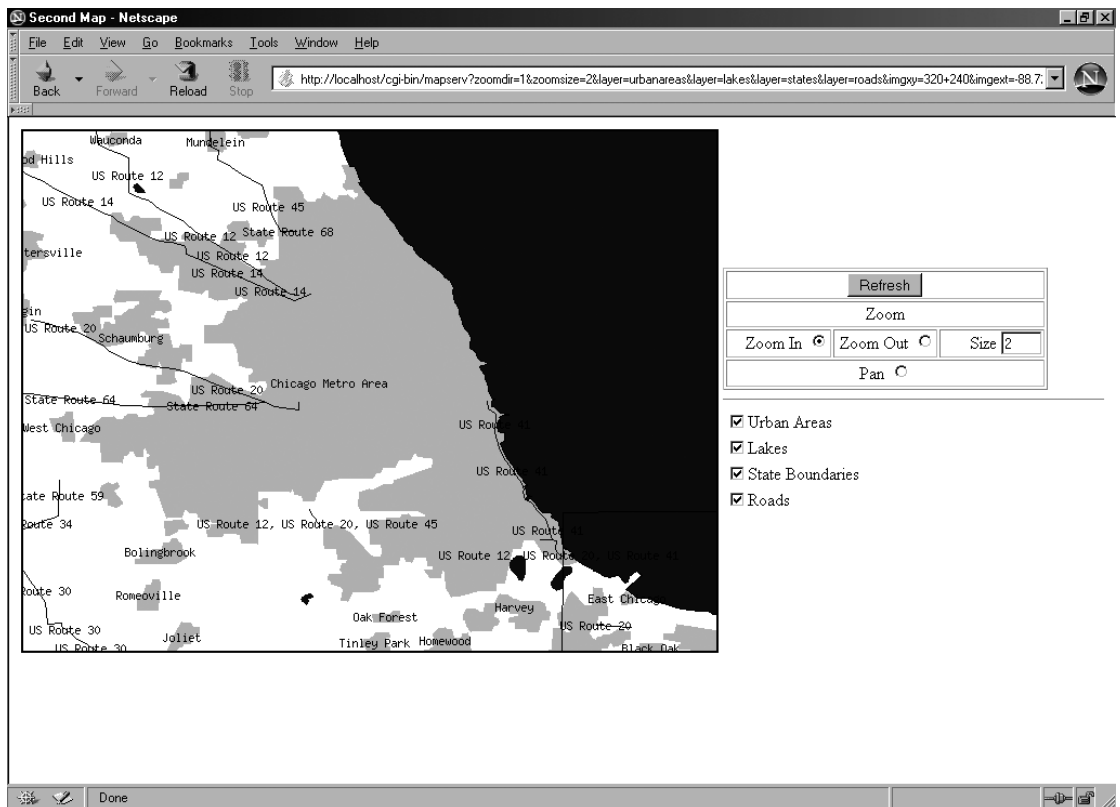


Figure 3-1. *The second map, with navigation controls and layer selection*

In the downloadable code for the book, the mapfile for this application is named `second.map`. The code in that file (with line numbers added) is shown in Listing 3-1, at the end of the chapter. If you haven't downloaded the code, open a file with any text editor, enter the code from the listing, and save it as `second.map`. The name of this file is important; MapServer manages its various mapfiles and template files by knowing their names.

In the code snippet that follows, Lines 001 through 009 set up the basic map image parameters. The keyword `NAME` defines the base name of any images created. Every time MapServer is invoked, it creates a unique identifier by concatenating the system time (i.e., the number of seconds

since 00:00:00 January 1, 1970) and the process ID. This unique identifier is appended to the base name to form the file name. A two- or three-character extension (which depends on the file type) is then appended. In some cases, MapServer will insert another string after the base name to differentiate a reference map image or legend image from the map image itself. The keyword `SIZE` specifies the pixel dimensions (width by height) of the map image. `IMAGECOLOR` sets the background color of the image to white (recall that colors are chosen in MapServer by specifying the three integer-valued RGB components between 0 and 255, with white being 255,255,255). `IMAGETYPE` is set to GIF. In the previous mapfiles, you used PNG and JPEG images. GIF is being chosen here just to give the software a workout. The base directory in which MapServer will look for the spatial data sets is specified as `/home/mapdata/` in Line 007. The initial extent of the map is determined by specifying the coordinates of the southwest and the northeast corners in Line 008. Finally, the file that contains the mapping between font alias and font location is specified by the keyword `FONTSET` in Line 009.

```
001 # This is our second map file
002 NAME "second"
003 UNITS dd
004 SIZE 640 480
005 IMAGECOLOR 255 255 255
006 IMAGETYPE gif
007 SHAPEPATH "/home/mapdata/"
008 EXTENT -180.00 0.00 -60.00 90.00
009 FONTSET "/var/www/htdocs/fontset.txt"
```

In the following code snippet, Lines 010 through 014 define the parameters of the `WEB` object. It begins with the keyword `WEB` and is terminated by the keyword `END`. The `WEB` object tells MapServer the name of the HTML template files (in this case there's only one, named `second.html`), the path to the images created, and the URL that points to those images. As before, `IMAGEPATH` specifies the path to the images created by MapServer. In this case, you're using an absolute path, but you could have used the relative path from the location of the mapfile. Note that you can't delete the initial or final `/` from the `IMAGEURL`. The string defined by `IMAGEURL` is appended to the base URL (i.e., `http://localhost`) to generate URLs for the images presented on the page.

Note If you're not sure why the `/` is important, or what the Apache directive `DocumentRoot` means, look it up at <http://httpd.apache.org/docs/mod/core.html#documentroot>.

```
010 WEB
011     TEMPLATE "/var/www/htdocs/second.html"
012     IMAGEPATH "/var/www/htdocs/tmp/"
013     IMAGEURL "/tmp/"
014 END
```

It's important to remember that MapServer renders layers in the order in which they're specified in the mapfile. The last layer in the mapfile is the top layer of the map—each is laid

over the previous layer to build the map image. This means that details presented earlier in the mapfile (that is, lower layers in the map image) may be obscured. This is most important when rendering polygon layers, because polygons can be filled with a specified color. If MapServer renders a point layer (representing, for example, the locations of cities) and then renders a polygon layer representing states, the cities won't be visible unless the color specified for the polygon layer is transparent. This problem is easily remedied by laying down the polygon layer first. It's also possible to render a polygon as a series of lines (i.e., as a `TYPE line` layer). Since a line is one dimensional, there's no area to be filled, so the detail from the layers below is still visible. Specifying `COLOR -1 -1 -1`, for which no fill color is used, also accomplishes transparency.

Layer 1: Urban Areas

Lines 015 through 032, which follow, define the first layer of the map. The data set contains selected urban areas in the United States, which can be downloaded from <http://nationalatlas.gov/atlasftp.html>. The feature attributes are AREA, PERIMETER, NAME, STATE, and STATE_FIPS. These attributes are described in the text file containing the metadata (`urbanap020.txt`), which includes the following definitions:

- AREA is the size of the shape in coverage units. The minimum area is 0 and the maximum is 0.298 coverage units.
- PERIMETER is the perimeter of the shape in coverage units. The minimum perimeter is 0 and the maximum is 6.774.
- NAME is the name of the urban area.
- STATE is the two-*character* FIPS code for the state (i.e., the usual two-letter postal abbreviation, such as ME for Maine).
- STATE_FIPS is the two-*digit* FIPS code for the state. The STATE_FIPS code is derived from the sequence number of an alphabetically sorted list of states names, Washington DC, and associated territories. When an urban area falls in more than one state, the state codes are listed, separated by dashes.

Note To quote from the US Census Bureau website (www.census.gov/geo/www/fips/fips.html), “Federal information processing standards codes (FIPS codes) are a standardized set of numeric or alphabetic codes issued by the National Institute of Standards and Technology (NIST) to ensure uniform identification of geographic entities through all federal government agencies. The entities include: states and statistically equivalent entities, counties and statistically equivalent entities, named populated and related location entities (such as, places and county subdivisions), and American Indian and Alaska Native areas.”

The layer begins with the keyword `LAYER` and is terminated with the keyword `END` at Line 032. This polygon layer renders urban areas across the United States from spatial data in the shapefile `urbanap020.shp`. The `NAME` keyword specifies a name for the layer. The name itself is optional, but if you use one, it must be no longer than 20 characters. The layer name is used as a CGI

reference to the layer from within the HTML template. Since you want to be able to turn layers on and off interactively from an HTML form, you must assign a name in this case.

The `STATUS` keyword determines whether a layer will be rendered or not, and whether its status can be changed. A layer with `STATUS default` is always rendered, while `STATUS on` or `STATUS off` can be toggled.

In order to label each urban area with the `NAME` attribute found in the shapefile, you need to identify `NAME` as the value of keyword `LABELITEM`. Whenever an urban area feature is rendered, the value of the `NAME` attribute for that feature will then be used to create the label.

Note There are several methods available if you want to look inside a shapefile. You can see feature attribute values by opening the associated DBF file in Excel or another spreadsheet program that can read DBF files. If you just want to find out the names of attributes, you can use the utility program `dbfinfo`, which is part of the `shapelib` library. Also, the utility program `ogrinfo` provides geographic information as well as feature values. See Chapter 10 for details.

Lines 022 through 031 specify the parameters of the only `CLASS` object in this layer. A `CLASS` object begins with the keyword `CLASS` and is terminated by the keyword `END` (on Line 031). Although this application will use classes more extensively than the previous map, for this layer you only need to specify a single default class that will include every feature in the shapefile. The `NAME` of a class is the label that will appear in the legend associated with the map. If a class has no name, it will still be rendered, but it won't appear in the legend. The keyword `COLOR` in the `STYLE` object specifies the color in which the feature will be drawn. Because this layer is a polygon, it will be filled with the color specified. If it were a line layer, the value of `COLOR` would specify the line color.

Each urban area will be labeled, and Lines 027 through 030 specify the label parameters. A `LABEL` object begins with the keyword `LABEL` and is terminated by the keyword `END` (on Line 030). In the `LAYER` object, the value of `LABELITEM` is set to `'NAME'`. This selects the attribute `NAME` as the source of the label text. Each label will be drawn in black, and its size will be `small`. Instead of `small`, you could have chosen `tiny`, `medium`, `large`, or `giant`.

```
015 LAYER
016     NAME "urbanareas"
017     DATA "urbanap020"
018     STATUS on
019     TYPE polygon
020     LABELCACHE on
021     LABELITEM 'NAME'
022     CLASS
023         NAME "Urban Areas"
024         STYLE
025             COLOR 212 192 100
026         END
027     LABEL
028         COLOR 0 0 0
029         SIZE small
```

```
030          END # label
031      END # class Urban Areas
032  END # layer urbanareas
```

Layer 2: Water Features

Lines 033 through 055 define the second layer of the map. The data set contains polygon water features in the United States, Puerto Rico, and the US Virgin Islands. The feature attributes are AREA, PERIMETER, FEATURE, NAME, STATE, and STATE_FIPS.

This layer is more complex than the previous and demonstrates the use of some keywords that you haven't encountered yet. It has been assigned the name "lakes", and the spatial data comes from the shapefile named hydrogp020—it's a polygon layer. You'll use the NAME attribute to label each feature. Layer STATUS is set to on, so you can turn it off and on from the browser.

In the "urbanareas" layer, a single class was used. Because no selection criteria were specified, every feature in the spatial data set was rendered. But since this new data set contains features other than lakes, you need some method of selecting only those features that you wish to render (i.e., lakes). You can do this by identifying to MapServer the attribute on which you'll base your selection. You can use the keyword CLASSITEM to accomplish this. In Line 040, CLASSITEM tells MapServer that FEATURE is the name of the attribute that will be used by this layer to define classes.

Since rendering a feature takes time, it might be too time-consuming to render every feature in a large data set. On the other hand, you don't want to ignore an entire class. MapServer provides a way to limit the number of features that are rendered. The keyword MAXFEATURES in Line 041 sets that limit to 100 for this layer. This means that no more than 100 lakes will be rendered, regardless of the extent of the map. If the map shows the entire continental United States, there will still be only 100 lakes shown. Limiting the number of rendered features this way can substantially reduce response time in cases in which a data set contains many features, but the technique is used most effectively if the spatial data set is sorted into some useful sequence—for example, lake area. In such a case, MapServer would render only the 100 largest lakes. Chapter 10 describes how to use the sortshp utility program to accomplish this.

The CLASS named "Lakes" uses the keyword EXPRESSION. The value associated with EXPRESSION can take one of three forms: a quoted string, a regular expression, or a logical expression. They work like this:

- If the value of EXPRESSION consists of a quoted string, then for every feature in the data set, MapServer compares the value of the attribute specified by CLASSITEM with the value of the quoted string. If the two are equal, that feature is included in the class.
- If the value of EXPRESSION consists of a regular expression, delimited by forward slashes, then for every feature in the data set, MapServer compares the value of the attribute specified by CLASSITEM with the regular expression. If a match is found, that feature is included in the class.
- If the value of EXPRESSION consists of a logical expression delimited by parentheses, then for every record in the data set, MapServer evaluates the logical expression. If it evaluates to true, that feature is included in the class. In addition to the various comparison operators, MapServer also supports a length function that returns the length (in characters) of its string valued argument. Logical expressions will be discussed in more detail in the next chapter.

In this case, you'll use a string comparison. In Line 044, the keyword `EXPRESSION` tells MapServer that for every feature in the data set, the attribute `FEATURE` is to be compared with the string `'Lake'`. If they're equal, that feature is a member of the class `"Lakes"` and will be rendered.

Caution Remember, while MapServer mapfile keywords aren't case sensitive, attribute names and values can be. For example, specifying `'lake'` will not match the value `'Lake'`. No match will occur, so no feature will ever be included in this class and no lakes will be rendered. There will be no error message either because it's not really an error. However, as noted previously, it appears that attribute names aren't case sensitive, since setting `CLASSITEM` to `"Feature"` will still work.

The features of this class will be rendered in blue (they're *water* features after all), and since the layer is a polygon, that will be the fill color. Labels will be small and black. The keyword `MINFEATURESIZE` allows you to set the size below which features will not be labeled. The size is specified in pixels. For lines, it represents the length of the line, and for polygons, it represents the smallest dimension of the bounding box. Setting `MINFEATURESIZE` to `auto` results in MapServer labeling only those features that are larger than their labels. Lines 053 through 055 terminate the `LABEL`, `CLASS`, and `LAYER` objects.

```

033 LAYER
034     NAME "lakes"
035     DATA "hydrogp020"
036     STATUS on
037     TYPE polygon
038     LABELCACHE on
039     LABELITEM "NAME"
040     CLASSITEM "FEATURE"
041     MAXFEATURES 100
042     CLASS
043         NAME "Lakes"
044         EXPRESSION 'Lake'
045         STYLE
046             SIZE 1
047             COLOR 0 0 255
048         END
049         LABEL
050             MINFEATURESIZE auto
051             COLOR 0 0 0
052             SIZE small
053         END # label
054     END # class Lakes
055 END # layer lakes

```

Note It's important to remember that an attribute that contains either a single quote (') or a double quote (") character can confuse MapServer. For example, `EXPRESSION ('[NAME]' eq 'O'Doyle')` won't select the feature with attribute `[NAME]` equal to `O'Doyle`, because the quoted value will appear to MapServer as an invalid string due to the three single quote marks. Thus, the expression will never evaluate to true. To fix this, you can replace the delimiter in the expression with double quotes, like this: `EXPRESSION ("[NAME]" eq "O'Doyle")`. The other option is to change the data set and replace every occurrence of a single quote with a double quote. You might have to do this anyway, since some shapefiles contain attributes that use both single and double quotes.

Layer 3: State Boundaries

Lines 056 through 067 define the third layer of your map. The data set contains polygon state boundaries in the United States, Puerto Rico, and the US Virgin Islands. The feature attributes are `AREA`, `PERIMETER`, `STATE`, and `STATE_FIPS`.

State labels have been omitted since the map is already very busy, and you want to keep things simple. The layer `TYPE` is `polygon`—therefore, if the color is set in the usual manner, it will fill the polygon and overlay the urban areas and lakes already rendered. However, by omitting the keyword `COLOR` and instead using the keyword `OUTLINECOLOR`, the polygon won't be filled, and the previous layers will remain visible. Again, `STATUS` is on, so you can control it.

```
056 LAYER
057     NAME "states"
058     DATA "statesp020"
059     STATUS on
060     TYPE polygon
061     LABELCACHE on
062     CLASS
063         STYLE
064             OUTLINECOLOR 0 0 0
065         END
066     END # class
067 END # layer states
```

Layer 4: Road Network

Lines 068 through 088 define the fourth layer of your map. The data set contains major roads in the United States, Puerto Rico, and the US Virgin Islands. The feature attributes include `LENGTH`, `FEATURE`, `NAME`, `STATE`, and `STATE_FIPS`. (Look in the file `roadtrl020.txt` to see a comprehensive list of attributes and attribute values.) As before, the value of the attribute `FEATURE` is the road type associated with each feature.

This layer of the map will show the principal highways of the United States. The spatial data is contained in the shapefile named `roadtrl020`. `STATUS` is on and the layer type is `line`. `LABELITEM` is set to `"NAME"` in order that each road is labeled with the contents of the `NAME` attribute. (Later, you'll see that this leads to aesthetic problems, but I'll address this in the next chapter.)

To select only principal highways, you'll select features based on the contents of the `FEATURE` attribute. This time, however, you'll use a regular expression rather than a string comparison.

Note If you're interested in seeing the range of features in this file, you can scan through the file `roadtrl020.txt` that comes with the shapefile. This file contains metadata (i.e., data about the spatial data). Its format is straightforward but difficult to read. If you feel comfortable with Unix command-line operations, you can execute the command `grep Enumerated_Domain_Value: roadtrl020.txt` to display the values for the attribute `FEATURE`.

As before, an expression is introduced by the keyword `EXPRESSION`. In the case of regular expressions, however, strings are delimited by forward slashes (/), and they're not quoted. The attribute `FEATURE` in *this* data set can contain any one of 29 different strings. The ones you're interested in (Principal Highway, Principal Highway Alternate Route, Principal Highway Business Route, etc.) all begin with the text `Principal Highway`. The regular expression that will match these strings is `/Principal Highway*/`. The asterisk tells MapServer that any feature with a `FEATURE` attribute that begins with the string `Principal Highway` will be accepted and rendered.

Finally, the map object is terminated in Line 089 by the keyword `END`.

```
068 LAYER
069     NAME "roads"
070     DATA "roadtrl020"
071     STATUS on
072     TYPE line
073     LABELCACHE on
074     LABELITEM "NAME"
075     CLASSITEM "FEATURE"
076     CLASS
077         NAME "Principal Highway"
078         EXPRESSION "/Principal Highway*/"
079         STYLE
080             SIZE 1
081             COLOR 0 0 0
082         END
083         LABEL
084             COLOR 0 0 0
085             SIZE small
086         END # label
087     END # class Principal Highway
088 END # layer roads
089 END # mapfile
```

Note Regular expressions are a common feature of most Unix and Unix-like operating systems. A regular expression (or regex) is a string of characters that represent string patterns. Similar to (but much more extensive than) the wildcard characters `?` and `*` of DOS and Windows, regular expressions are used to find matching patterns in strings. MapServer's regular expressions (running under Linux) are POSIX compliant, so the capabilities are portable to Windows. The syntax of regular expressions is beyond the scope of this book. Consult any standard Unix user's guide or one of the numerous online resources devoted to this topic.

The HTML Template

Since this mapping application is more complex than the previous applications, a separate HTML initialization file will be employed in addition to the HTML template. This file will specify the name of the CGI program to run, the name of the mapfile to use, the original extent of the map, the initial zoom factor, and the layers to be displayed on first invocation.

Note You might think that because MapServer is invoked by the initialization page, it would know the program to use in the HTML template file. But some thought should show you that this isn't the case. When MapServer is invoked by the initialization page, it doesn't know it has been invoked—it's not self-aware. It could assume that the program name to insert in the template file is its own name, but there will be times that you don't want that to happen. You need to tell MapServer what program the template file should invoke the next time the form is submitted. The CGI variable that MapServer interprets as the program name is `program`, and its associated substitution string is `[program]`.

The Initialization File

The code for the initialization file is in `second_i.html` and can be found in the book's code download. The contents of this file (with line numbers added) are shown in Listing 3-2. If you haven't downloaded the code, open a file with any text editor, enter the code from the listing, and then save it as `second_i.html`.

Lines 004 through 011 (shown in the following code block) produce a form that passes the CGI variables in Lines 006 through 010 to the MapServer executable, `/cgi-bin/mapserv`. Line 006 tells MapServer that the CGI form variable `program` contains the string that's to be substituted for the string `[program]` when it reads the HTML template file. In this case, it's the same as the action specified in Line 004. Line 007 indicates that MapServer should get its configuration information from the mapfile specified by the value of the form variable `map`—in this case `/home/mapdata/second.map`.

```
004 <form method=POST action="/cgi-bin/mapserv">
005   <input type="submit" value="Click to initialize">
006   <input type="hidden" name="program" value="/cgi-bin/mapserv">
007   <input type="hidden" name="map" value="/home/mapdata/second.map">
008   <input type="hidden" name="zoomsize" size=2 value=2>
009   <input type="hidden" name="layers"
010     value="urbanareas lakes states roads capitals">
011 </form>
```

The variable `zoomsize` is given the value 2 in Line 008. MapServer will place this value into a form variable when it scans the template file. It specifies the rate at which MapServer will zoom in and out when the map is refreshed. Lines 009 and 010 set the value of the form variable `layers` to a space-delimited list of layer names. MapServer will render every layer on this list, setting the status of each to on. These layer names must match the layer names specified in the mapfile by the keyword `NAME` in each of the layer objects. If they don't match, the layer name will be ignored and the layer will fail to render.

Note MapServer understands the name of every CGI variable used in this file. This is how it communicates between the HTML form and the mapfile—thus, spelling is important, but case isn't.

The Template File

We come now to the HTML template. It's more complex than the template for the "Hello World" application or the first map, since it must be capable of doing more. Some of the complexity arises because the layout of more elements on the screen will require more code. In addition to this, there are several substitution strings that allow you to manipulate the map from the browser.

In the downloadable code for the book, the code for this template file is named `second.html`. The contents of this file (with line numbers added) is shown in Listing 3-3. If you didn't download the code, open a file with any text editor, enter the code from the listing, and save it as `second.html`.

Lines 001 through 003 are the usual HTML preamble.

```
001 <html>
002 <head><title>Second Map</title></head>
003 <body>
```

Lines 004 through 052 produce the form that lets a user interact with MapServer by changing the values of CGI variables contained in the form. The first substitution string is `[program]` in Line 004. The initialization file sets this to `/cgi-bin/mapserv`. Recall that when MapServer is invoked, it reads the mapfile specified during initialization, creates the map image specified, scans the specified template file, and makes any substitutions in the substitution strings it recognizes (and for which it knows the values). So when MapServer scans `second.html`, it substitutes the string `/cgi-bin/mapserv` wherever it finds `[program]`.

```

004     <form name="the_form" method=GET action="[program]">
005         <table width="100%">
006             <tr>
007                 <td width="60%">

```

Line 008 contains the substitution string `[img]`. MapServer replaces this string with the name of the map image it has created, along with the URL that points to that image. Every invocation will produce a different name, but it should look something like the following: `second110254987124367.gif`. Notice that the tag containing `[img]` is an input field. This allows point-and-click navigation of the map. If the user clicks the image, the coordinates of that click are sent back to MapServer. MapServer then puts the clicked point at the center of the map (effectively, panning across the image), and (if the zoom size and direction are set) changes the scale and extent of the map (zooming in or out). Line 013 provides a submit button to refresh the map.

```

008             <input name="img" type="image" src="[img]"
009                 width=640 height=480 border=2></td>
010             <td width="40%" align="left">
011         <table border="1" width="300">
012             <tr><td align="center" colspan="3">
013                 <input type="submit" value="Refresh"></td></tr>

```

Lines 014 through 030 provide zoom controls for the map. The form variable `zoomdir` determines the zoom direction. Setting `zoomdir` to 1 means that when the map is refreshed (by a click of the Refresh button or the map image), the next view will be zoomed in by the current value of `zoomsize`. Setting `zoomdir` to -1 means that the next view will be zoomed out by the value of `zoomsize`. Setting `zoomdir` to 0 means that the next view will be at the same scale as the last. The value of `zoomdir` is specified by one of the three radio buttons in Lines 016, 019, or 025. Checking one radio button in a group unchecks the others so that the zoom direction is passed, unambiguously, to MapServer from one invocation to the next.

Also note the substitution string `[zoomdir_1_check]` in Line 017. This represents the selection state of the radio button associated with a `zoomdir` value of 1. If the radio button with value 1 had been checked prior to this invocation, MapServer would replace `[zoomdir_1_check]` with the string `checked`. Similarly, if the button with value -1 had been checked, MapServer would replace `[zoomdir_-1_check]` with `checked`. If a radio button is unchecked, then its substitution string will be replaced with a null `''`.

Line 022 contains the form variable `zoomsize`, with its value set to the substitution string `[zoomsize]`. When MapServer scans the template, it replaces this with the current value of `zoomsize`.

```

014             <tr><td align="center" colspan="3">Zoom</td></tr>
015             <tr><td align="right" width="100">Zoom In
016                 <input type="radio" name="zoomdir" value=1
017                     [zoomdir_1_check]></td>
018                 <td align="right" width="100">Zoom Out
019                 <input type="radio" name="zoomdir" value=-1
020                     [zoomdir_-1_check]></td>
021                 <td align="right" width="100">Size
022                 <input type="text" name="zoomsize" size=2

```



```

023             value=[zoomsize]></td></tr>
024         <tr><td align="center" colspan="3">Pan
025             <input type="radio" name="zoomdir"
026             value=0 [zoomdir_0_check]></td></tr>
027     </table>
028     <hr size="1">
029     <table>
030         <tr><td colspan="3">

```

Lines 031 through 041 define four checkboxes for a form variable named `layer`. The value of each is the name of one of the layers in the mapfile. If a layer is to be rendered, then the checkbox associated with that layer (the checkbox that has a value equal to the name of the layer) must be checked. Lines 031 and 032 show how MapServer indicates that a layer has been checked. Every layer in the mapfile will have a substitution string associated with it—the name of which is constructed by appending an underscore character (`_`) to the layer name, and then adding the word `check`. For example, Line 031 shows that the string associated with the `"urbanareas"` layer is named `[urbanareas_check]`. These strings can have two values: checked or `""` (null), just as in the case of radio buttons. When MapServer scans the template, it replaces the checkbox substitution strings with the appropriate values.

```

031             <input type="checkbox" name="layer" value="urbanareas"
032             [urbanareas_check]>Urban Areas</td></tr>
033         <tr><td colspan="3"><input type="checkbox"
034             name="layer" value="lakes"
035             [lakes_check]>Lakes</td></tr>
036         <tr><td colspan="3"><input type="checkbox"
037             name="layer" value="states"
038             [states_check]>State Boundaries</td></tr>
039         <tr><td colspan="3"><input type="checkbox"
040             name="layer" value="roads"
041             [roads_check]>Roads</td></tr>
042     </table>
043     </td>
044 </tr>
045 </table>

```

Line 046 provides for a virtual mouse click on the center of the image. Recall that Line 008 is an input image field. If the user doesn't use the mouse to click the image, but clicks the Refresh button instead, MapServer won't know what the center of the new image should be. The CGI variable `imgxy` returns to MapServer the location (in image coordinates) of the mouse click. If you set `imgxy` to the coordinates of the center of the image, then MapServer won't change the center of the extent if the user clicks Refresh rather than the image.

Finally, the form variables and substitution strings in Lines 047 through 049 allow MapServer to maintain the state of the application. Line 047 causes the value of the image extent to be set to the current value of the map extent (i.e., the coordinates of the lower-left and upper-right corners of its bounding box). This allows MapServer to track the current value as the user zooms and pans. As with the initialization file, MapServer must know which mapfile to read. It remembers this by storing it as a hidden form variable when it finds the substitution string (`[map]`), on

Line 048) during its scan. The substitution string [program] in Line 049 is used by MapServer to store the name of the program to invoke when the form is submitted.

```
046     <input type="hidden" name="imgxy" value="320 240">
047     <input type="hidden" name="imgext" value="[mapext]">
048     <input type="hidden" name="map" value="[map]">
049     <input type="hidden" name="program" value="[program]">
050 </form>
051 </body>
052 </html>
```

Once you've saved the mapfile in /home/mapdata/ and entered and saved the initialization file and HTML template in the DocumentRoot (/var/www/htdocs/ on the development system), display the initialization file in your browser:

```
//http://localhost/second_i.html
```

This will display a page (shown in Figure 3-2) that's almost identical to the previous initialization pages (except for the absence of the broken image icon). Click Click to initialize, and you should see a map similar to the one shown in Figure 3-3. You can select and deselect layers for rendering by clicking the checkboxes. Clicking the radio buttons Zoom In, Zoom Out, and Pan will allow you to move around in the map and explore all its features. Depending on where you click and what zoom factor you've set, MapServer should respond with images similar to those in Figures 3-4 and 3-5. It's very easy to use.

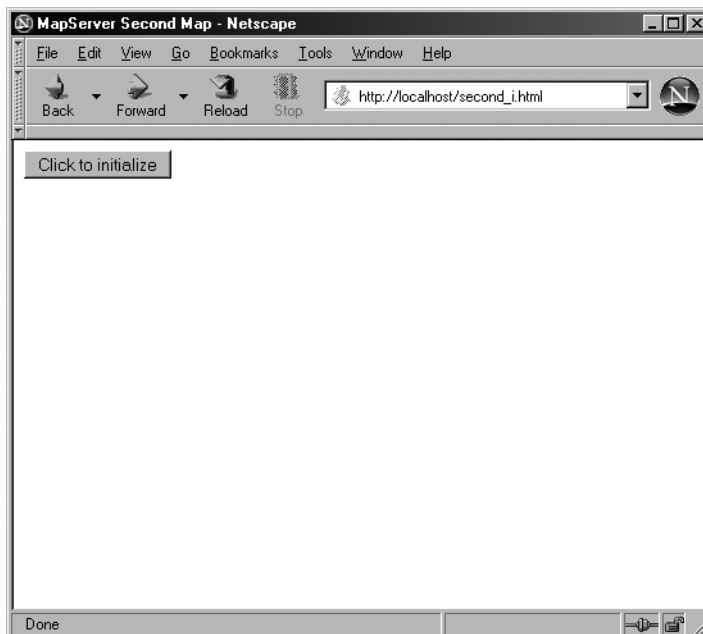


Figure 3-2. *The initialization page for the second map*

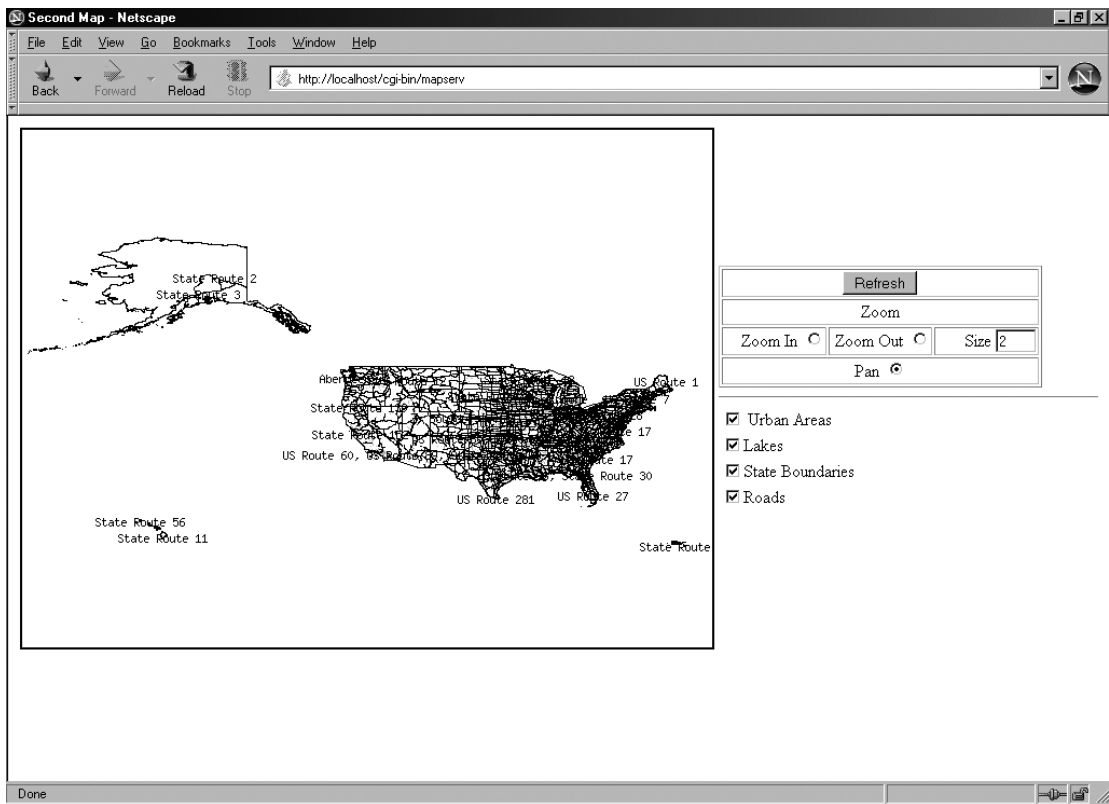


Figure 3-3. Initial display of the second map at full extent

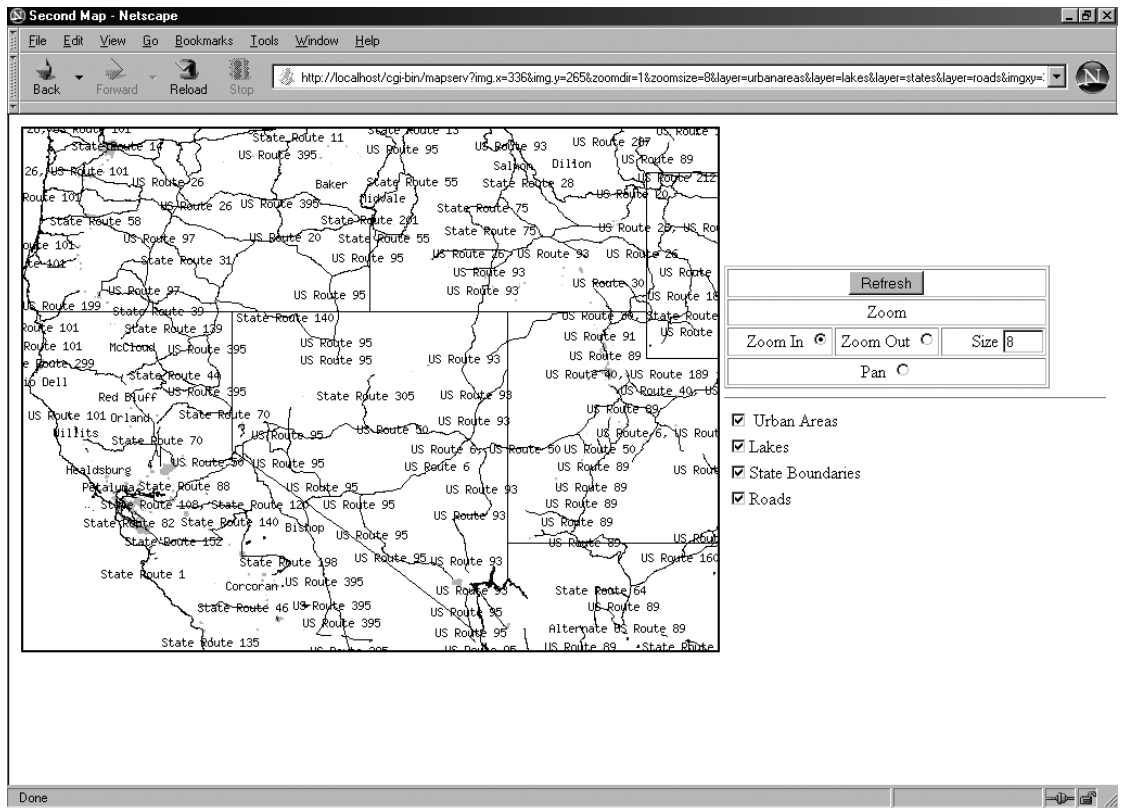


Figure 3-4. The coast of northern California as rendered by the second mapping application

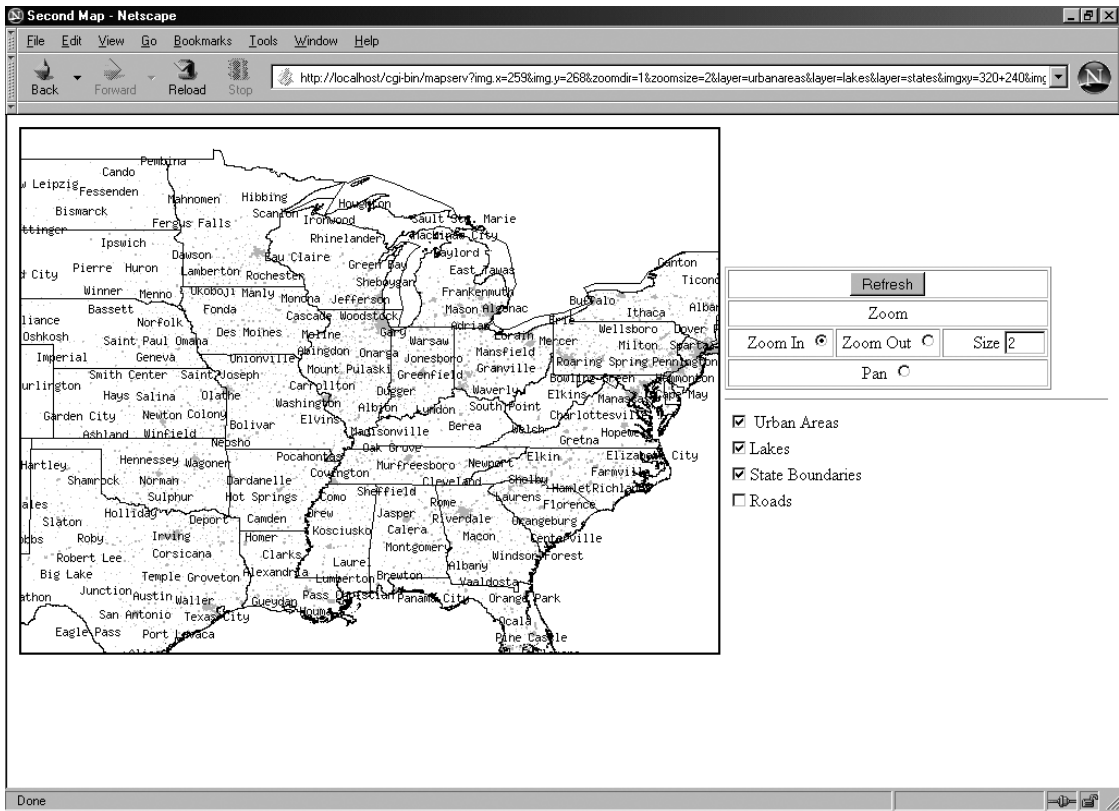


Figure 3-5. The Midwest and eastern United States as rendered by the second mapping application

If the first two applications executed correctly, the only possible sources of error are typos in the mapfile, HTML template, or initialization file—or perhaps a compilation error. If you haven't configured MapServer build to produce GIF images, it will fail.

In the next chapter, I'll address some of the shortcomings of this map, and show you ways to make it more useful and attractive. The changes, however warranted, will be mostly cosmetic—the hard work of getting a functioning, interactive map up and running has been done.

Summary

In this chapter, you've explored the mapfile in some detail and built an application that can actually convey geographic information. You've learned how to classify features using string comparisons and regular expressions. You've also learned to label features with attribute information using the keyword LABELITEM. But perhaps what's most important is that you've learned how to build an *interactive* mapping application. This is a big step. Static digital maps in various formats are available that can be used for the presentation of geographic information (for example, Microsoft Map, which comes packaged with Microsoft Office)—but however elegant or easy-to-use these maps are, the important applications (now and in the future) are all interactive.

Having climbed the hill, it's now time to climb the mountain. The next chapter explores the aesthetic judgments you must make and the methods MapServer provides for you to implement those judgments. Some new concepts will be introduced, but you'll mainly gain experience at manipulating color, shape, and size to produce pleasing images. This hardly seems like map making at all, but then again, many of the difficult problems with map making (within the limits of available technology) are aesthetic.

The image of cartographer as surveyor—with transit over his shoulder and trusty rod man at his side taking the measure of the earth—is focused on the conceptually simple task of data gathering. Of course, this has always been time consuming, expensive, and arduous—freezing in the winter, eaten alive by flies (or worse) in the summer. But this job, however unpleasant, is bounded. A position is a position, and once you have it, you're done. The map maker, on the other hand, is always tempted to wring the next quantum of aesthetic value out of the current data, to change this line or that color. Creating beautiful maps is addictive, and MapServer makes it very easy to practice this addiction.

Code Listings

In the previous sections of this chapter, the code for this application was presented in fragments. While this is a convenient format for performing a line-by-line analysis, it's cumbersome if you're trying to type the code (or even read it). The mapfile, initialization file, and HTML template are presented here, without interruption.

Listing 3-1. *The mapfile second.map*

```
001 # This is our second map file
002 NAME "second"
003 UNITS dd
004 SIZE 640 480
005 IMAGECOLOR 255 255 255
006 IMAGETYPE gif
007 SHAPEPATH "/home/mapdata/"
008 EXTENT -180.00 0.00 -60.00 90.00
009 FONTSET "/var/www/htdocs/fontset.txt"

010 WEB
011     TEMPLATE "/var/www/htdocs/second.html"
012     IMAGEPATH "/var/www/htdocs/tmp/"
013     IMAGEURL "/tmp/"
014 END

015 LAYER
016     NAME "urbanareas"
017     DATA "urbanap020"
018     STATUS on
019     TYPE polygon
020     LABELCACHE on
021     LABELITEM 'NAME'
```

```
022     CLASS
023         NAME "Urban Areas"
024         STYLE
025             COLOR 212 192 100
026         END
027         LABEL
028             COLOR 0 0 0
029             SIZE small
030         END # label
031     END # class Urban Areas
032 END # layer urbanareas

033 LAYER
034     NAME "lakes"
035     DATA "hydrogp020"
036     STATUS on
037     TYPE polygon
038     LABELCACHE on
039     LABELITEM "NAME"
040     CLASSITEM "FEATURE"
041     MAXFEATURES 100
042     CLASS
043         NAME "Lakes"
044         EXPRESSION 'Lake'
045         STYLE
046             SIZE 1
047             COLOR 0 0 255
048         END
049         LABEL
050             MINFEATURESIZE auto
051             COLOR 0 0 0
052             SIZE small
053         END # label
054     END # class Lakes
055 END # layer lakes

056 LAYER
057     NAME "states"
058     DATA "statesp020"
059     STATUS on
060     TYPE polygon
061     LABELCACHE on
062     CLASS
063         STYLE
064             OUTLINECOLOR 0 0 0
065         END
066     END # class
```

```

067 END # layer states

068 LAYER
069     NAME "roads"
070     DATA "roadtrl020"
071     STATUS on
072     TYPE line
073     LABELCACHE on
074     LABELITEM "NAME"
075     CLASSITEM "FEATURE"
076     CLASS
077         NAME "Principal Highway"
078         EXPRESSION /Principal Highway*/
079         STYLE
080             SIZE 1
081             COLOR 0 0 0
082         END
083         LABEL
084             COLOR 0 0 0
085             SIZE small
086         END # label
087     END # class Principal Highway
088 END # layer roads
089 END # mapfile

```

Listing 3-2. *The HTML initialization file second_i.html*

```

001 <html>
002 <head> <title>MapServer Second Map</title></head>
003 <body>
004 <form method=POST action="/cgi-bin/mapserv">
005 <input type="submit" value="Click to initialize">
006 <input type="hidden" name="program" value="/cgi-bin/mapserv">
007 <input type="hidden" name="map" value="/home/mapdata/second.map">
008 <input type="hidden" name="zoomsize" size=2 value=2>
009 <input type="hidden" name="layers"
010     value="urbanareas lakes states roads capitals">
011 </form>
012 </body>
013 </html>

```

Listing 3-3. *The HTML template second.html*

```

001 <html>
002 <head><title>Second Map</title></head>
003 <body>
004 <form name="the_form" method=GET action="[program]">
005 <table width="100%">

```



```

006     <tr>
007         <td width="60%">
008             <input name="img" type="image" src="[img]"
009                 width=640 height=480 border=2></td>
010             <td width="40%" align="left">
011                 <table border="1" width="300">
012                     <tr><td align="center" colspan="3">
013                         <input type="submit" value="Refresh"></td></tr>
014                     <tr><td align="center" colspan="3">Zoom</td></tr>
015                     <tr><td align="right" width="100">Zoom In
016                         <input type="radio" name="zoomdir" value=1
017                             [zoomdir_1_check]></td>
018                         <td align="right" width="100">Zoom Out
019                         <input type="radio" name="zoomdir" value=-1
020                             [zoomdir_-1_check]></td>
021                         <td align="right" width="100">Size
022                         <input type="text" name="zoomsize" size=2
023                             value=[zoomsize]></td></tr>
024                     <tr><td align="center" colspan="3">Pan
025                         <input type="radio" name="zoomdir"
026                             value=0 [zoomdir_0_check]></td></tr>
027                 </table>
028                 <hr size="1">
029                 <table>
030                     <tr><td colspan="3">
031                         <input type="checkbox" name="layer" value="urbanareas"
032                             [urbanareas_check]>Urban Areas</td></tr>
033                     <tr><td colspan="3"><input type="checkbox"
034                         name="layer" value="lakes"
035                         [lakes_check]>Lakes</td></tr>
036                     <tr><td colspan="3"><input type="checkbox"
037                         name="layer" value="states"
038                         [states_check]>State Boundaries</td></tr>
039                     <tr><td colspan="3"><input type="checkbox"
040                         name="layer" value="roads"
041                         [roads_check]>Roads</td></tr>
042                 </table>
043                 </td>
044             </tr>
045         </table>
046         <input type="hidden" name="imgxy" value="320 240">
047         <input type="hidden" name="imgext" value="[mapext]">
048         <input type="hidden" name="map" value="[map]">
049         <input type="hidden" name="program" value="[program]">
050     </form>
051 </body>
052 </html>

```



Modifying a Map's Look and Feel

Since a map's primary purpose is, after all, to present information effectively, a map should be visually attractive. Despite the presence of interactive features like zoom, pan, and layer selection, the map you created in Chapter 3 has a number of serious defects. The major problem is the amount of information crammed into that one image. Huge amounts of spatial information can be compressed into any map, but beyond a certain point, that information is no longer retrievable because the map is too busy. On the other hand, a very sparse map is unwieldy because displaying a low information density requires increasing the physical size of the map. As a creator of maps and a user of MapServer, your goal will be to present enough information on a map so that the viewer is neither inconvenienced nor confused.

Figure 4-1 shows the initial view of the second map. The first thing you'll notice is that all its features and labels are rendered in the same font and color. There are vague indications of state boundaries beneath the rat's nest of overlapping highways and labels, but the density of graphic detail is so high in the east that no individual features can be discerned at all.

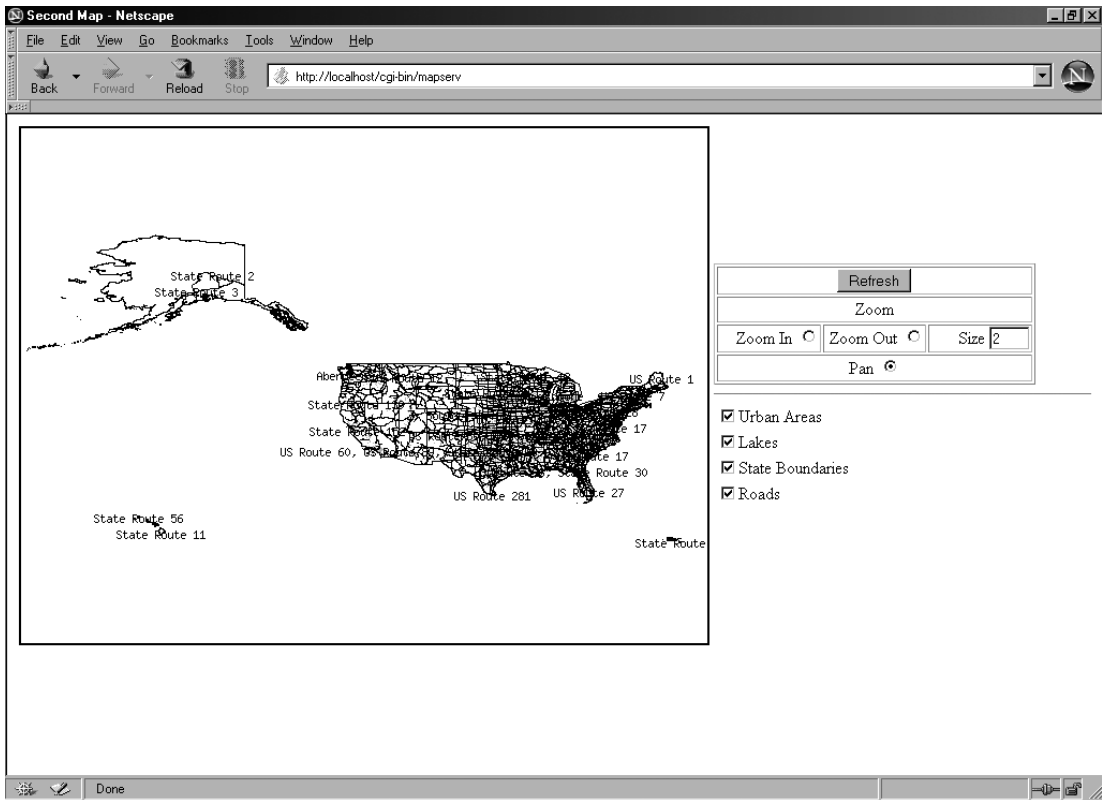


Figure 4-1. *The second mapping application produces a very busy map.*

Reduce the detail by deselecting the Roads layer (uncheck the check box labeled Roads and click Refresh). A faint dusting of urban areas should appear, as shown in Figure 4-2—however, identification is impossible, since the names, being so much larger than the areas they label, are difficult to associate with those areas. Notice also that although the Lakes layer is still being rendered, there don't appear to be any lakes located in the entire continental United States. You might, however, depending on the resolution of your computer monitor, see what appears to be a lake somewhere close to Puerto Rico.

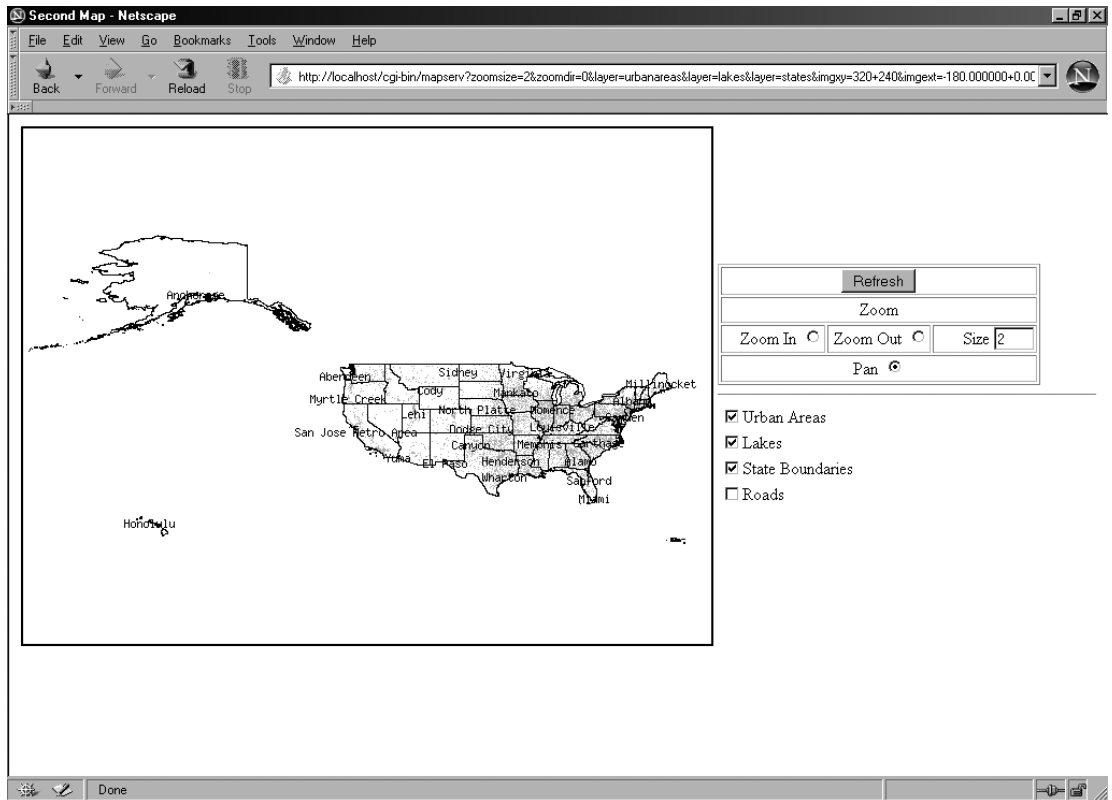


Figure 4-2. The map without the road network is more readable, but its utility is reduced.

Reselect all layers and zoom in by a factor of two. Pan across so that Chicago is roughly in the center of the screen. You can see that the confusion still isn't reduced, as shown in Figure 4-3. Zoom in several more times and lakes begin to appear. The map should look better, as it does in Figure 4-4—however, you should also notice that the labels on the Road layer are littered about, obstructing your view of urban areas and lakes. So far, the road, city, and lake labels all look the same—in the mass of detail, it's almost impossible to draw any conclusions about the geographic relationships between the rendered features.

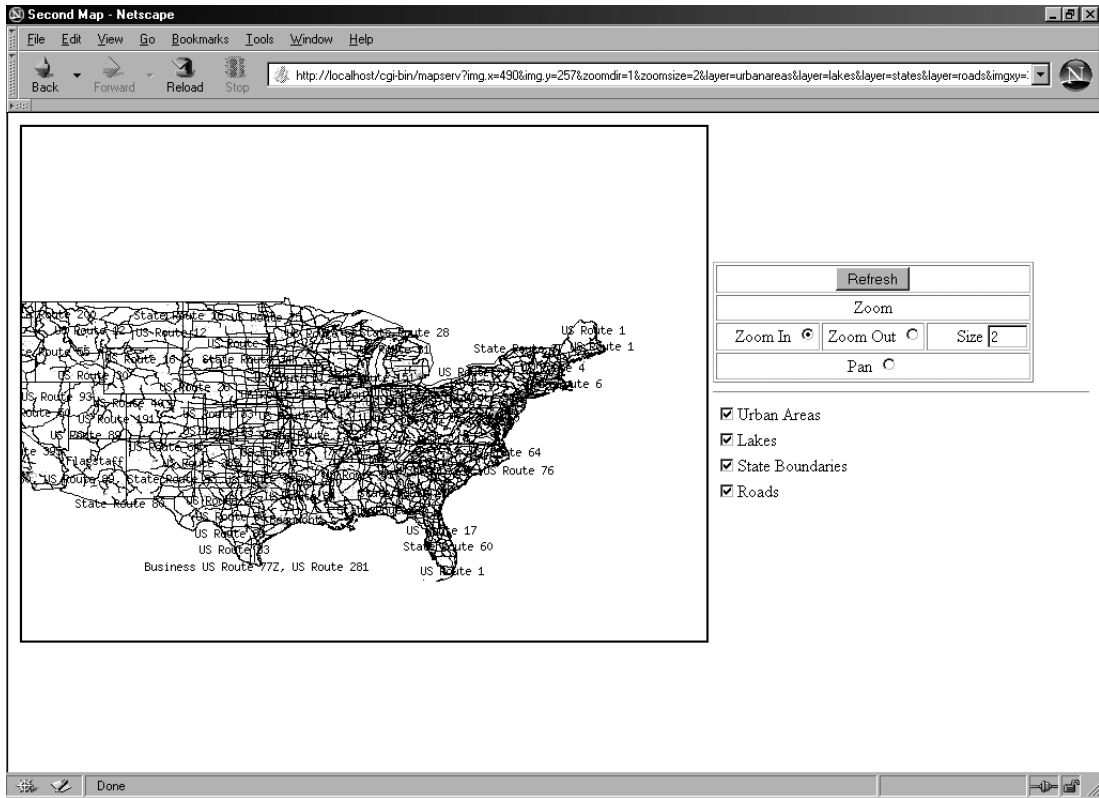


Figure 4-3. Increasing the map scale by a factor of two does little to improve readability.

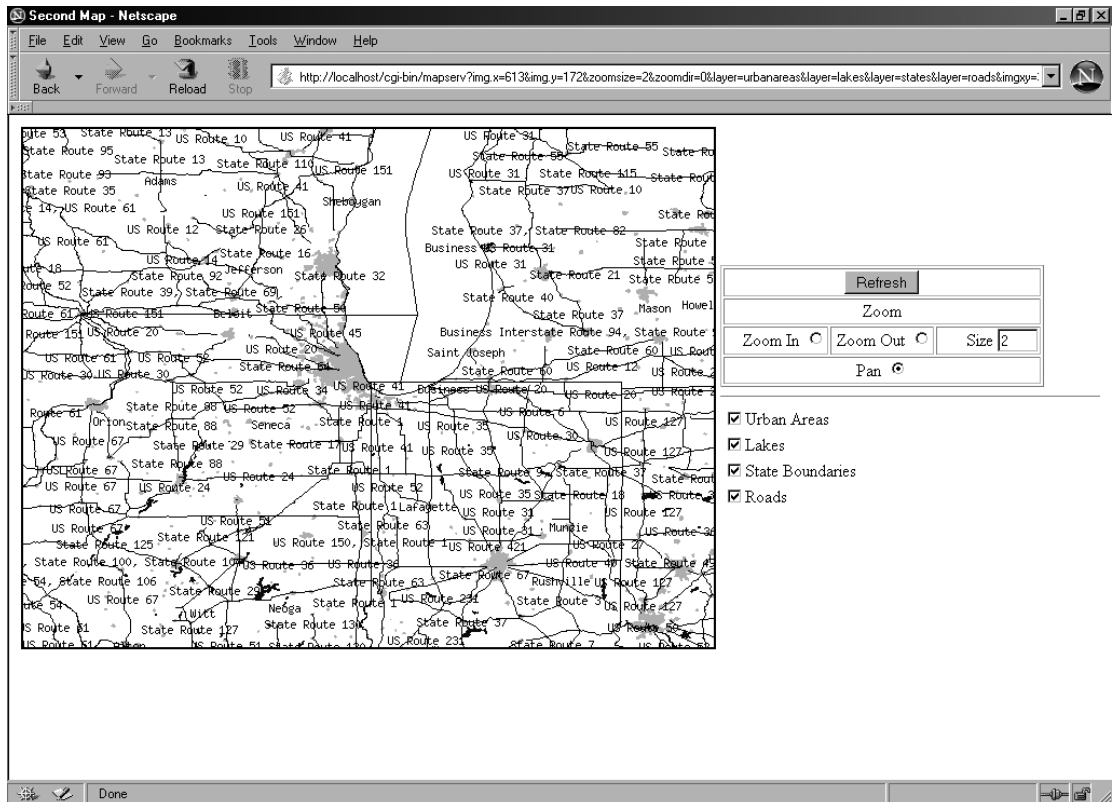


Figure 4-4. Further scale increases reduce feature density, but make identification difficult due to the abundance of undifferentiated labels.

If you continue to zoom in, you'll reach a point at which the map becomes less confusing: features are well separated—both from one another and from labels—and it's possible to see both the roads and the cities. However, there are still a number of problems. For example, if you look at Figure 4-5, you can see how to get from Meadville to Hamilton, but you can't tell where Meadville and Hamilton are located in relation to the rest of the country. While local references are clearer, continental references (the signposts that allow you to relate one place with another over large distances) are no longer visible. At both ends of the scale you lose—the small-scale map is too confusing to use and the large-scale map is so sparse that you lose context.

Also, label orientation problems make it difficult to see which labels refer to which roads. For example, although Meadville and Hamilton are both on US Route 36, they could easily be mistaken for being on US Route 65.

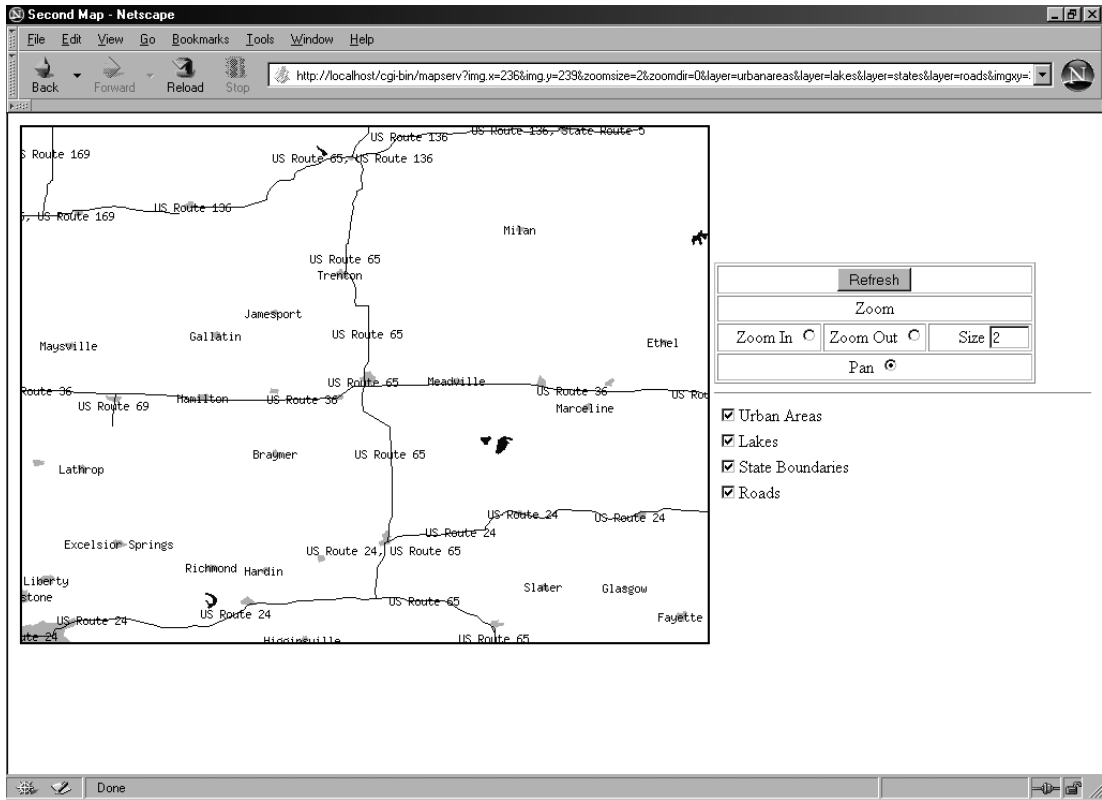


Figure 4-5. *Reduced feature density at large scales removes context and leads to confusion.*

The optimum information density for any map depends on several factors: the type of information you're trying to represent, the spatial distribution of this information, and the graphical design of the map. You have little, if any, control over information type and distribution—whether you need to create a road map or want to show monthly precipitation over an extended area, you're stuck with the spatial data sets that you have. Each type of map possesses different design requirements and constraints. Nevertheless, effective graphic design (over which you have major control) is critical in creating a useful map, no matter what the map represents.

The process of creating a map, whether digital or paper, is not quantitative. You must guide your progress by eye and make your own determinations. Nonetheless, making maps *is* more craft than art. An eye for graphic design is useful but not an absolute requirement—maps are working documents that have a job to do—so a technical understanding of the underlying spatial data is the most important qualification.

In this chapter, I'll demonstrate the features of MapServer that allow you to craft beautiful maps. There are a lot of these features, and the mapfile you'll create will be much larger than the previous ones. Consequently, this chapter is longer than its predecessors.

Because of this, the format of this chapter will be somewhat different. The line-by-line discussion of the code (also available at the Apress website, www.apress.com) will be preceded by a brief description of MapServer syntax for methods addressing the issues mentioned previously. Once I've introduced the syntax, I'll move through most of the mapfile in sequence, skimming quickly or omitting material that has already been covered, but focusing on new keywords (and new uses for old ones). This chapter isn't intended to provide encyclopedic coverage of MapServer. Refer to Chapter 11 for a detailed presentation of mapfile vocabulary and syntax.

The mapfile for this example is `third.map`. Some of the features have been presented before, so I won't spend much time on those. However, there are many new keywords and objects that you haven't yet seen, which will be the focus.

Note Donald Knuth, author of *The TeXbook* (Addison Wesley, 1986), was perhaps the first computer scientist to introduce computer technology into the daily graphical design of documents. The markup language he invented (TeX) allowed anyone with sufficient patience to create “beautiful documents.” These documents could be books, articles, letters, or even grocery lists, (yes, many years of otherwise productive time have been spent on the design of beautiful grocery lists). Likewise, with the computer doing the heavy lifting, you may find that you become consumed by the creation of beautiful maps.

The Graphic Design of Maps

Having described the defects of the second map in some detail, I'll now explain how these defects relate to the features available in MapServer.

The first issues mentioned in the previous section were the lack of color and the amount of detail displayed. Second, urban areas were displayed, but their names were unidentifiable; likewise, although lakes were rendered, they still weren't visible. Also, there were still too many labels obscuring features. After zooming in repeatedly, an uncluttered scale was finally reached, but the displayed extent couldn't be located in the larger context. Additionally, highway labels weren't aligned with the features, making it difficult to identify road names. These defects can be categorized in the following way:

Inappropriate labels. By rendering labels in different colors and fonts, you can give the viewer some cues to help distinguish, for example, labels for highways from labels for towns. By orienting labels appropriately for specific features (e.g., highway names drawn parallel to the road), you provide further cues for understanding which labels are associated with which features. Figure 4-6 shows the region around Kansas City as produced by the mapping application from Chapter 3. Figure 4-7 shows what the same region looks like when rendered by the application in this chapter. In the new map, the label for each class differs from the labels of the other classes. As you can see, urban areas have been given more prominent labels than small cities, and interstate highway labels are more prominent than those for other highways and roads. Notice also that the labels for urban areas and small cities look different from the labels for road features.

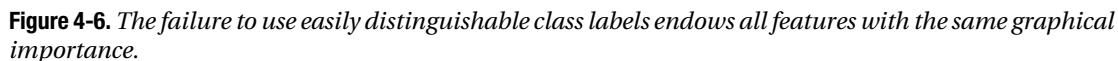


Figure 4-6. *The failure to use easily distinguishable class labels endows all features with the same graphical importance.*

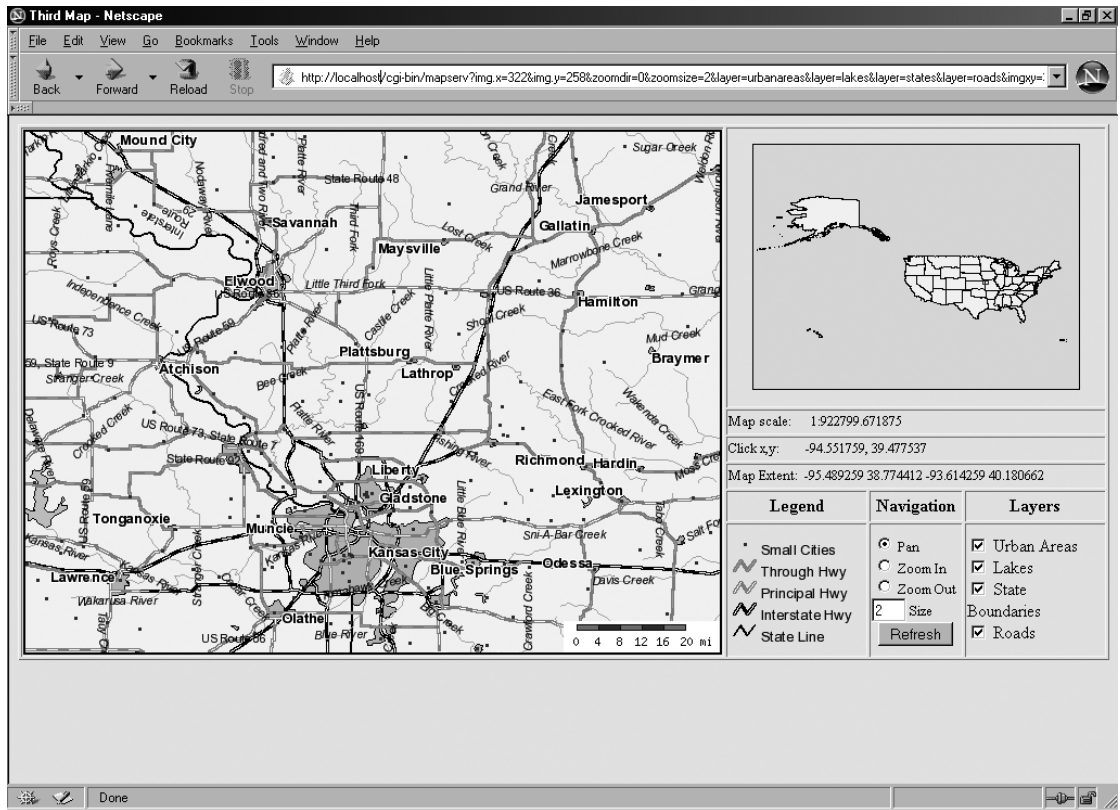


Figure 4-7. Effective labeling makes clear the relative importance of different features.

Inappropriate detail. At some scales, some information doesn't matter. For instance, rendering every urban area and highway when displaying the entire country is pointless. The urban areas would be unidentifiable and the highways so tightly packed together that they couldn't be distinguished from another. By selectively displaying different layers at different map scales, the clutter can be reduced and only information relevant to that scale presented. Figure 4-8 presents a smaller-scale view of the area around Kansas City. The map is dominated by roads and road labels. Rendering the roads while suppressing most of the labels (as shown in Figure 4-9) greatly enhances the readability of the map.

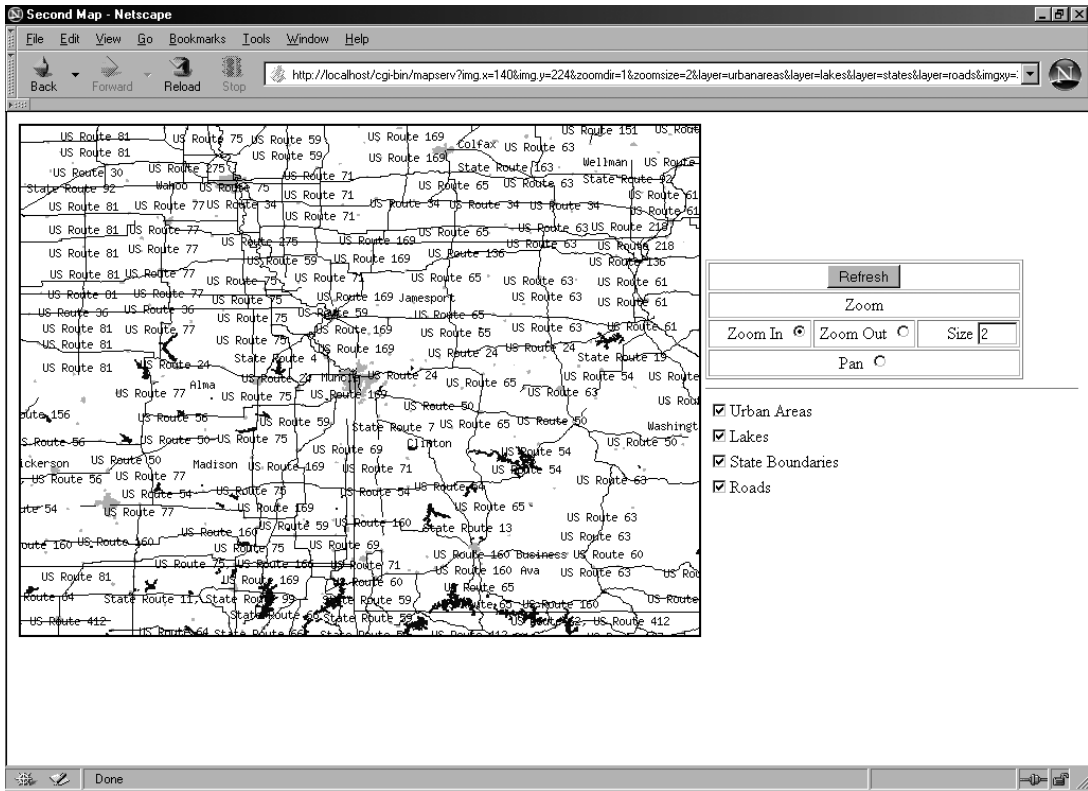


Figure 4-8. Sometimes less really is more. A comprehensive document can impede the transfer of information by obscuring the distinctions that form its basis.

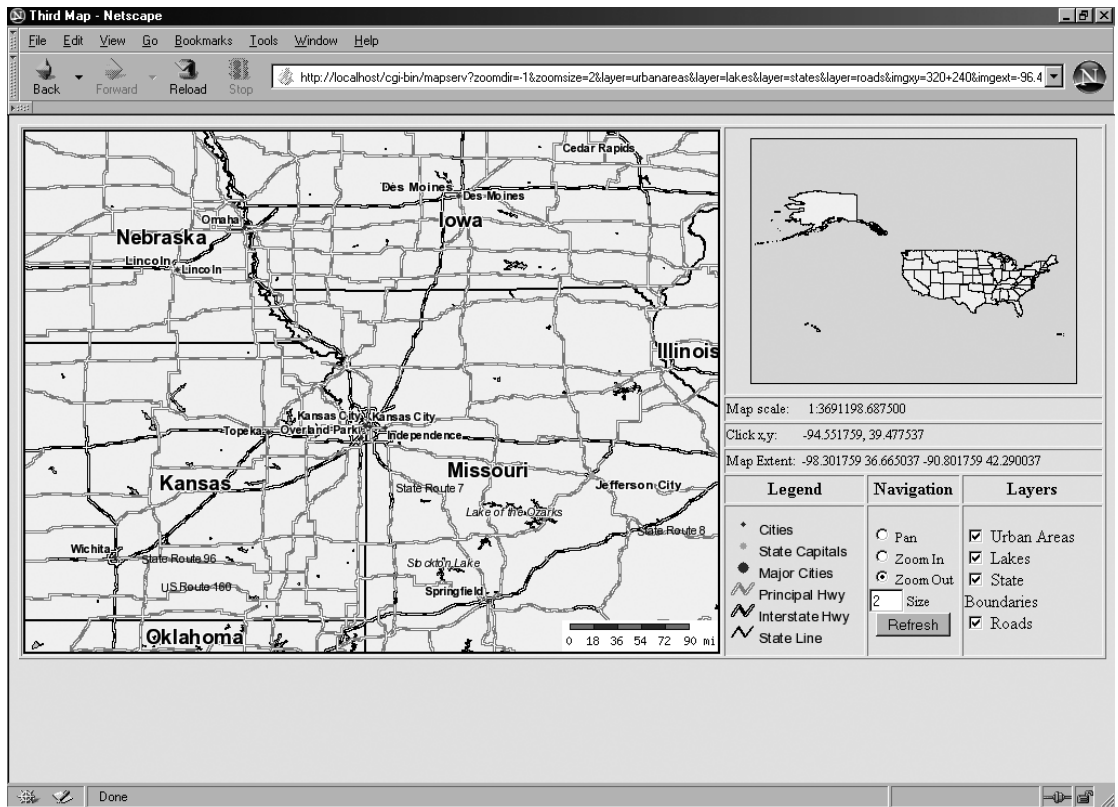


Figure 4-9. Suppressing detail can enhance the information content.

Absence of graphical distinction between features. While a road may be a road, not all highways are the same. Different types of roadways should be distinguished from one another. In fact, features whose attributes differ in any significant way should be distinguished graphically. Figures 4-8 and 4-9 (just shown) also demonstrate the effectiveness of using different symbols to render different classes of features.

Absence of signposts. It's important that viewers know where they are no matter what the scale of the map—but if you include too much large-scale information in your map, clarity will be reduced. Giving the viewer a reference outside the map can provide this large-scale detail without cluttering the map itself. Notice the reference map in Figure 4-9. This clearly places the displayed extent of the main map within a continental context.

Fixing the first three defects takes judgment and an eye for simplicity. Providing signposts requires only the knowledge of the techniques used to create legends, scale bars, and reference maps. These elements can increase the utility of a good map, but they can't fix one that's too busy or too sparse.

These things are as true for paper maps as they are for digital ones. However, the constraints on paper maps are of a different nature. A paper map's size can be made large to accommodate more detail—that is, the map maker can decide that the physical size of the paper must be increased. A road map, for instance, will contain roads, cities, and water features for an extended

area. Features must be labeled, of course, and a legend provided to define the various symbols used. Since an important concern of any road map user is distance, a scale bar is required to allow estimation of distances. Each of these elements takes up space. In addition, space must be inserted between graphical elements to maintain readability. So the paper map maker is faced with deciding between increasing the paper size (making the map more unwieldy) and reducing the size (making the map more difficult to read).

Note For some maps, cartographers don't have the luxury of limiting detail. Maps of this type must be unwieldy to be usable. Aeronautical charts are an example of this kind of map—pilots need the detail not only to plot a course home, but also to avoid obstacles and identify landmarks. Of course, this requires that every pilot learn the art of map folding while controlling a hurtling piece of machinery thousands of feet above the ground.

On the other hand, the size of a digital map (viewed on a computer screen) is fixed by the size of the screen. This means that the map maker must work to provide the appropriate amount of detail at every scale, so that users aren't overwhelmed as they zoom in and out. Selecting which features to render will always be a concern of the digital map maker since there's always more information than there is space to draw it.

The rest of this chapter will be spent looking at ways to address the balance between too much and too little, using the tools provided by MapServer. I'll also cover the use of scale bars, legends, and reference maps.

In the downloadable distribution from the Apress site, the code for this chapter can be found in three files: `third.map`, `third_i.html`, and `third.html`. If you haven't downloaded these files, it would be a good idea to do it now, because they're considerably longer than the previous code examples. However, for purposes of reference, code listings for these three files are provided at the end of the chapter.

Labeling for Clarity

A LABEL object begins with the keyword LABEL and is terminated by the keyword END. It's contained in a CLASS object and it labels the features in that class. (A LABEL object can also be contained in a LEGEND object or a SCALEBAR object. The syntax is almost the same, but I'll discuss this separately when we get to signposts.) A label has several characteristics, including font, color, orientation, position, and size.

Fonts

You can choose to use TrueType fonts or MapServer's built-in bitmapped fonts. The bitmapped fonts are always available and need no external resources, but they limit the use of some other features. TrueType fonts require external resources but allow you to manipulate the label in several useful ways. A comparison of bitmapped and TrueType fonts is shown in Figure 4-10. You must identify to MapServer whether you're using bitmapped or TrueType fonts, which is done with the keyword TYPE. Setting TYPE `bitmap` selects bitmapped fonts and TYPE `truetype`

selects TrueType fonts. If you omit the TYPE keyword, MapServer will default to bitmapped fonts. If you choose to use TrueType fonts, then it's necessary to identify the actual font that will be used (in this book's examples, they're all variations of Arial). By setting the keyword FONT to arial, you let MapServer know that the font pointed to by the string "arial" in the FONTSET file is to be used for this label. Font size is specified differently for bitmapped fonts than for TrueType fonts. When the TYPE is truetype, the numerical values assigned to the keyword SIZE represent point size. Bitmapped fonts set SIZE to one of five values: tiny, small, medium, large, or giant.

Bitmapped tiny
 Bitmapped small
Bitmapped medium
 Bitmapped large
 Bitmapped giant

 Truetype 8pt arial
 Truetype 10pt arial
 Truetype 12pt arial
 Truetype 14pt arial
 Truetype 16pt arial

 Truetype 8pt arial bold
 Truetype 10pt arial bold
 Truetype 12pt arial bold
Truetype 14pt arial bold
Truetype 16pt arial bold

Figure 4-10. *Bitmapped fonts are always available but have limited range; TrueType fonts are site dependent but provide much more flexibility.*

Color

MapServer provides several keywords relating to the color of a label. The value of the keyword COLOR determines the RGB color of the label text itself. However, a 1-pixel-wide outline can be drawn around the text with the value OUTLINECOLOR. This makes the label easier to read against a busy background.

Orientation

In the maps generated by the application developed in Chapter 3, all labels had the same orientation—they were parallel to the bottom of the map. This is fine for features like lakes and urban areas, but for roads and rivers (or any other linear feature), it's preferable for the labels to run along the features they describe. This requires the use of TrueType fonts, since it isn't possible to change the orientation of labels created with bitmapped fonts. The keyword ANGLE can be used to specify the angle at which all the labels in a class are drawn (see Figure 4-11). A positive angle rotates the label the specified number of degrees counterclockwise, while a negative angle rotates it clockwise. For line layers only, the value auto is also available. If auto

is specified, then MapServer will calculate the correct angle to make the label run along the line feature.

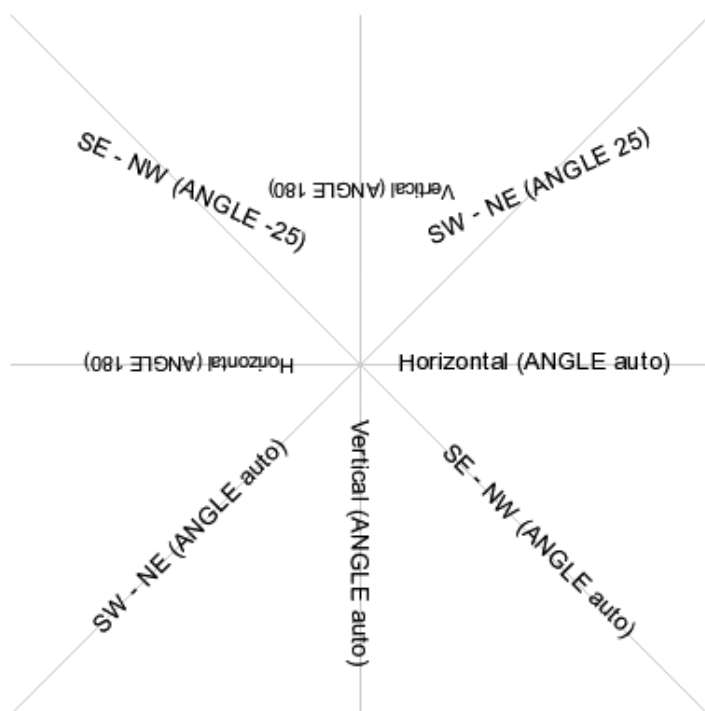


Figure 4-11. *Specifying label orientation directly is sometimes useful, but letting MapServer choose based on feature geometry is usually more effective.*

Positioning Labels

The placement of a label with respect to the feature that it labels is governed by the value of the keyword `POSITION`. Figure 4-12 shows the valid label positions and the associated keyword values. Across the upper part of the square, there are three positions: left, center, and right. Down the side of the square there are also three positions: upper, center, and lower. Each combination of vertical and horizontal position is labeled with the appropriate letters: `ul` for upper-left, `cc` for center-center (at the middle of the square), `lr` for lower-right, etc. Labels for a line layer can only be positioned at `lc` or `uc`, while point and polygon layers can take any of the eight outside locations on the square. MapServer can also be directed to select a position automatically, so that the label doesn't interfere with others. Not surprisingly, the value for this option is `auto`.

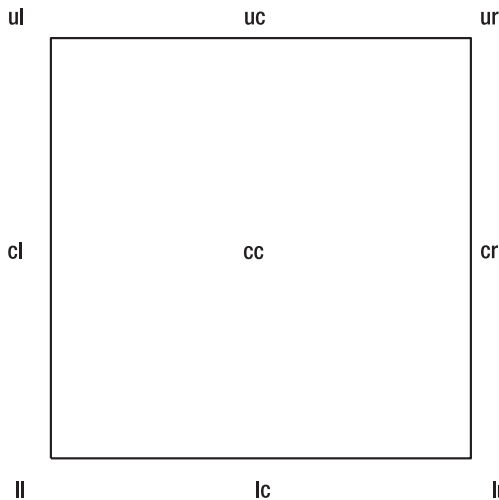


Figure 4-12. *Explicit label positioning is another capability that can be useful in some circumstances, but it's usually better left to MapServer.*

Although MapServer determines where labels are drawn, it possesses no aesthetic sense. It will draw as many labels as it can find room for. This can lead to situations in which a single feature with a large extent (like a highway) may have multiple labels. If the labels are sufficiently far apart, this doesn't lead to problems. When they're too close together, however, you'll want to limit the number of labels that are drawn for the same feature. To do this, you can use the keyword `MINDISTANCE` to specify the number of pixels between duplicate labels.

On the other hand, some features may be so small (such as a small lake at large scale) that the size of the label will dwarf the feature. The keyword `MINFEATURESIZE` sets the size (in pixels) of the smallest feature that will be labeled.

Assigning Font Attributes to Labels

Lines 180 through 190 define a `LABEL` object for a layer that renders lakes. In Line 181, the keyword `TYPE` is assigned the value `truetype`. This indicates to MapServer that the label text is to be rendered using TrueType fonts. In Line 182, the keyword `FONT` is assigned the value `arial`. This font will be used to draw the text. Recall that "arial" is an alias for the actual path to the appropriate font file. The file `FONTSET.txt` defines the mapping from alias to font file. Previously, `TYPE bitmap` was specified. Bitmapped fonts are simpler to use than TrueType fonts, but lack some of the aesthetic possibilities. Bitmapped fonts have fixed sizes while TrueType fonts can be scaled to any font size (specified in "points"). Thus, in Line 183, the size of the label is set to 8 points. In addition, TrueType fonts can be rotated. This means that if a road, for example, is heading from the southeast to the northwest, a TrueType label can be aligned parallel to the road. This is a much neater look than the bitmapped option, in which labels can only be drawn parallel to the bottom of the map.


```
180          LABEL
181          TYPE truetype
182          FONT "arial"
183          SIZE 8
184          OUTLINECOLOR 255 255 255
185          COLOR 0 0 0
186          MINDISTANCE 100
187          POSITION lr
188          MINFEATURESIZE auto
189          WRAP ' '
190      END
```

Line 184 specifies an `OUTLINECOLOR` value of 255 255 255 (white). This value is used as the color to draw a 1-pixel-wide border around the text. This is useful for making the label stand out against a busy background. The keyword `COLOR` specifies the color with which the label itself is to be drawn.

Line 186 specifies the minimum distance in pixels between duplicate labels (i.e., labels for the same feature). By setting the keyword `MINDISTANCE` to a larger value, more space is provided between labels, making the map easier to read.

Positioning Labels

The keyword `POSITION` in Line 187 determines where the label will be placed with respect to the feature being labeled. If the value is set to `auto`, then MapServer draws the label where it won't interfere with other labels. Usually, a label is drawn only if a position can be found that doesn't interfere with other labels. However, if the keyword `FORCE` is set to `true`, then the label is drawn regardless of other labels. If the keyword `MINFEATURESIZE` is set to `auto`, MapServer will only draw labels that are smaller than their features (but any integer value can be specified to override this behavior).

Wrapping Labels

The keyword `WRAP` specifies a character that will cause the label text to wrap to a new line. In this case, a space has been specified. This is done to produce multiline labels, which can be useful if the label text is long.

Caching Labels

The label cache is an important capability. By default, MapServer caches the labels for all the layers and renders them once all the layers have been drawn.

The keyword `LABELCACHE` in Line 170 isn't part of the label itself—it tells MapServer to cache all the labels for layers that specify `LABELCACHE on`, and to render all the labels at the same time. This allows MapServer to ensure that labels don't interfere with one another, and also to select an appropriate position for them. Label caching can be turned off for a particular layer by specifying `LABELCACHE off`. Also note that `POSITION auto` and `MINFEATURESIZE auto` are only available for cached labels.

Adding Label Text

The actual label text is usually found in the attribute table associated with the shapefile. The attribute to use is specified using the keyword `LABELITEM`. The value associated with `LABELITEM` must be the attribute name. In this case, the attribute name is "NAME". It's also possible to define a text string at the class level that will be used as a label. I'll discuss this feature later in the section.

Labels for the layer `interstate1` are defined in Lines 215 through 225. Line 223 contains the keyword `ANGLE`. The value of `ANGLE`, given in degrees, is the angle that the label will make with the bottom edge of the map. For example, `ANGLE 45` would draw labels that are rotated 45 degrees counterclockwise. Setting `ANGLE auto` lets MapServer choose the orientation (for line features, this is parallel to the feature).

```

215          LABEL
216          TYPE truetype
217          FONT "arial"
218          SIZE 8
219          OUTLINECOLOR 255 255 255
220          COLOR 0 0 0
221          MINDISTANCE 200
222          POSITION auto
223          ANGLE auto
224          MINFEATURESIZE 50
225          END

```

Sometimes you need more flexibility in the selection of label text than the `LABELITEM` keyword provides. The keyword `TEXT` provides this flexibility by allowing label text to be taken from multiple sources, including constant strings and multiple attribute values. The values associated with `TEXT` can be of two types. The first is a quote- or parenthesis-delimited string constant. This string constant is used to label every feature in the class. The second type is more complicated—the value is delimited by parentheses, but instead of containing merely a string constant, the parentheses contain references to attribute names as well. The attribute names are delimited by square brackets, and the string constants are inserted between the delimited attribute names.

Line 253 contains the class-level keyword `TEXT`. It's commented so that MapServer ignores it, but it can be useful for exploring the contents of an attribute. In this case, the label text will be taken from the attributes specified. The attribute values for each feature will be concatenated, along with any string constants, and used to label the feature. The keyword `WRAP` can be used with these kinds of labels in exactly the same way as noted previously.

```

253 #          TEXT ([FEATURE],[NAME])

```

There are a number of other `LABEL` keywords that haven't been discussed yet, but they'll be explained in detail in Chapter 11.

Using Scale to Reduce Clutter

On a paper map, features must be rendered without regard to scale. However, with digital maps, you have the flexibility of rendering maps based on scale. At large scales, most maps suppress features that the cartographer considers insignificant. These features might include

villages, local roads, or small lakes. At small scales, it usually doesn't make sense to omit features—a major highway between A and B is likely to be important whether the map scale is large or small. In addition to including or omitting features based on map scale, you can change the symbols used for drawing particular features, change the size and color of the symbols, and even omit or substitute entire layers.

The layer-level keyword `MAXSCALE` sets the maximum scale at which a layer will be rendered. Similarly, the keyword `LABELMAXSCALE` sets the maximum scale at which labels will be rendered. The values of these two keywords don't have to be the same, and probably shouldn't be. To understand why, consider what happens as the scale of a map increases. A viewer might like to see highway names at small scale, but beyond a certain point, the names become too large and merely clutter the map. However, you'll still want to draw the highways at a large scale, since they show how different places are connected (which is more significant than their names).

Corresponding to the maximum scale keywords are the minimum scale keywords `MINSCALE` and `LABELMINSCALE`.

Note Scale plays such an important role in the aesthetic development of a mapping application that you should always print the map scale (using the substitution string `[scale]`) on the web page. This allows you to tune the scale break points to fit the data you have. If the numerical scale isn't a technical requirement of the application, it can always be omitted when the application goes into production. Map scale is displayed on the map shown in Figure 4-9.

`MAXSCALE` and `MINSCALE` are used at several points in the mapfile. In the layer named `urbanareas`, `MAXSCALE` is set to 1999999. Towards the end of the mapfile, in the `largecities` layer, `MINSCALE` is set to 2000000. When the map is rendered at a large scale, cities will then be rendered as points. At smaller scales, when the representative fraction drops below 1:2,000,000, urban area polygons will be drawn instead.

```

460 LAYER
461     NAME "largecities"
462     DATA "citiesx020"
463     STATUS default
464     TYPE point
465     LABELCACHE on
466     LABELITEM "NAME"
467     MINSCALE 2000000
468     CLASS
469         EXPRESSION(([Pop_2000]>100000)and([Pop_2000]<=1000000))
470         NAME "Cities"
471         STYLE
472             SYMBOL "Circle"
473             SIZE 4
474             COLOR 255 0 0
475             BACKGROUNDCOLOR 255 0 0
476     END

```

```

477         LABEL
478             TYPE truetype
479             FONT "arialbd"
480             SIZE 8
481             POSITION auto
482             OUTLINECOLOR 255 255 255
483             COLOR 0 0 0
484         END
485     END
486 END

```

This technique is used again when interstate highways are rendered. There are two interstate layers, named `interstate1` and `interstate2`. The first is drawn when the scale exceeds 1:7,500,000, as defined in Lines 197 through 227 (a fragment of which is shown in the following code snippet):

```

206     MINSCALE 7500001
207     CLASS
208         NAME "Interstate Hwy"
209         EXPRESSION /Limited Access*/
210         STYLE
211             SYMBOL "BigLine"
212             SIZE 1
213             COLOR 0 0 0
214     END

```

The second, as described in Lines 232 through 267, is drawn at smaller scales, with the relevant section shown in the following code:

```

241     MAXSCALE 7500000
242     CLASS
243         NAME "Interstate Hwy"
244         EXPRESSION /Limited Access*/
245         STYLE
246             SYMBOL "BigLine"
247             SIZE 3
248             COLOR 0 0 0
249     END
250     OVERLAYSYMBOL "DashedLine"
251     OVERLAYSIZE 1
252     OVERLAYCOLOR 255 255 255

```

The large-scale rendering uses a 1-pixel-wide black line to render highway segments—keeping the lines thin when the whole country is displayed reduces clutter. At smaller scales, however, a more complicated, fatter symbol is used, which employs overlay symbols to render the interstates so that they're more distinguishable from the smaller roads that are drawn at this scale.

Similarly, principal highways are rendered below 1:4,000,000, thru highways below 1:1,000,000, and other roads below 1:500,000. However, keep in mind that these scale values

aren't magic numbers. They may work well for this application when displayed at a screen resolution of 1024×768, but they're not necessarily optimum.

Classifying Features

To MapServer, a layer consists of a group of features, derived from the same data set, that will be rendered together at the same scale. However, you may not want to draw all the features contained in a given data set, or you may not wish to draw all the features the same way. For example, rendering every feature in a data set that contains the entire road network of the United States would take a considerable amount of time and produce a map that's too cluttered to be useful. Similarly, rendering a dirt road with the same symbol used to render interstate highways produces a map that fails to distinguish graphically between features that are, in the real world, very distinct.

Classes allow you to differentiate between features based on attributes and render features based on class. The previous maps used classes, since every layer must have at least one class. The simplest, default class includes every feature in the data set. In this case, the layer has a single class and every feature in this class is rendered the same way.

However, you'll usually want to avoid using just the default class, and instead choose to classify your features, for the two reasons mentioned previously: first, you may not wish to render every feature; and second, you may want to render features that differ in some attribute with different symbols, colors, or sizes.

Using Expressions to Define Classes

The simplest (and fastest) way to determine class inclusion with MapServer is to use a string comparison. You use the layer-level keyword `CLASSITEM` to identify the name of the attribute that will be used to classify the features. Then, using the class-level keyword `EXPRESSION`, you specify the comparison string. It's good practice to quote the string to ensure that characters are correctly interpreted. This is shown in the code snippet that follows. The `CLASSITEM` attribute of every feature in the data set will be compared with the value of the `EXPRESSION` string. If the `EXPRESSION` string matches the `CLASSITEM` value identically, that feature will be included in the class. Although fast and easy to use, this method isn't very powerful, since the string specified by `EXPRESSION` must match the value of the attribute identified by `CLASSITEM` exactly.

```
CLASSITEM "NAME"
CLASS
    NAME "City"
    EXPRESSION "Seattle"
...
END
```

If you wish to select features based on more complicated matching criteria (but still use only the value contained in the feature specified by `CLASSITEM`), you must use regular expressions. This, however, is an involved topic and is beyond the scope of this book. Refer to any standard Unix text or guide (or consult the man pages at www.rtf.com/man) for a description of regular expression syntax. Regular expressions must be delimited by forward slashes (/). As with string comparisons, the regular expressions specified by the keyword `EXPRESSION` are compared to the value of the `CLASSITEM` attribute, and included in the class if a match is found. The specification

of matching criteria by means of regular expressions is very flexible, but it's still limited to matching the value of a single attribute. For example, the following code

```
CLASSITEM "Feature"
CLASS
    NAME "interstates"
    EXPRESSION /^Limited Access/
...
END
```

will select from a data set those features for which the value of the TYPE attribute begins with the string `Limited Access`. The need to do this might arise, for example, if road types that are part of an interstate highway system are identified by strings like `Limited Access Highway`, `Limited Access Highway Alternate Route`, or `Limited Access Highway Business Route`. (Remember that the metadata associated with spatial data sets is designed to be readable by humans, and therefore attribute values are often more complicated than simple alphanumeric codes.)

Logical expressions allow for more complex classification of features that are based on the values of one or more attributes. No CLASSITEM need be specified (and in fact, will be ignored if present). The keyword EXPRESSION introduces the logical expression, which is delimited by parentheses. The syntax is straightforward: a logical expression consists of an attribute name enclosed in square brackets, a comparison operator, and a value. Table 4-1 shows the comparison operators available. For example, the following code compares the value of the numeric attribute POPULATION with the numeric value 100000:

```
EXPRESSION ( [POPULATION] < 100000 )
```

It will include a feature only if the value of its POPULATION attribute is less than 100,000. Like C and Perl, MapServer uses different operators to compare strings than it does to compare numbers, and you must take care to observe the difference. If an attribute is string valued, then its reference must be enclosed in quotes, as must the comparison value. Single or double will do, as long as they're properly balanced. Consider the following code:

```
EXPRESSION ( '[STATE_FIPS]' eq 'MN' )
```

This will include a feature only if the value of attribute STATE_FIPS is equal to the string MN. Logical expressions can be combined using the conjunction and disjunction operators and and or. Consider also the following example:

```
EXPRESSION (( [POPULATION] < 100000 ) and ( '[STATE_FIPS]' eq 'MN' ))
```

This will match features for which POPULATION is less than 100000 and STATE_FIPS is equal to MN.

Caution Major confusion can arise if a string-valued attribute contains number strings (e.g., "123"). If a numeric comparison is made with a string-valued attribute, there will never be a match (123 will never be equal to "123"), nor will there be an error. Know the data types of your features.

Caution The MapServer mapfile reference document has a typographical error—the numeric “not equal” operator, !=, doesn’t show the bang (!).

Table 4-1. *Logical Expression Comparison Operators*

Operator	Data Type
!=	Numeric
=	Numeric
>	Numeric
<	Numeric
>=	Numeric
<=	Numeric
and	Logical
or	Logical
eq	String
ge	String
gt	String
le	String
lt	String
ne	String

Note Although a single method must be used to define a class, each class in a layer can use a different method.

Using Classes

Now that you know *how* to define a class, let’s look at what this flexibility can do for you in terms of graphical clarity. First, a layer can contain many classes. Recall that a layer is a MapServer object that references a single spatial data set and renders that data set at a specific scale. This means that all the features in a layer will be of the same type (point, line, or polygon, for example).

However, although all the features are the same type, there are still obvious differences—an interstate highway isn't the same as a local unpaved road, and a state capital isn't the same as village. By classifying the features, you can make MapServer distinguish these features graphically. For example, interstate highways can be rendered with a fat line and other highways with thinner lines. Or state capitals might be rendered with a large red dot while other towns are drawn with smaller black dots of sizes that vary according to population. Figure 4-13 shows what happens when you do this. In this example, highways have been rendered four different ways based on highway type. Figure 4-14 shows a large region of the Midwestern United States at a larger scale. In this figure, cities are differentiated based on two criteria: population and whether the city is a state capital. Larger cities are denoted by larger dots and state capitals by green dots.

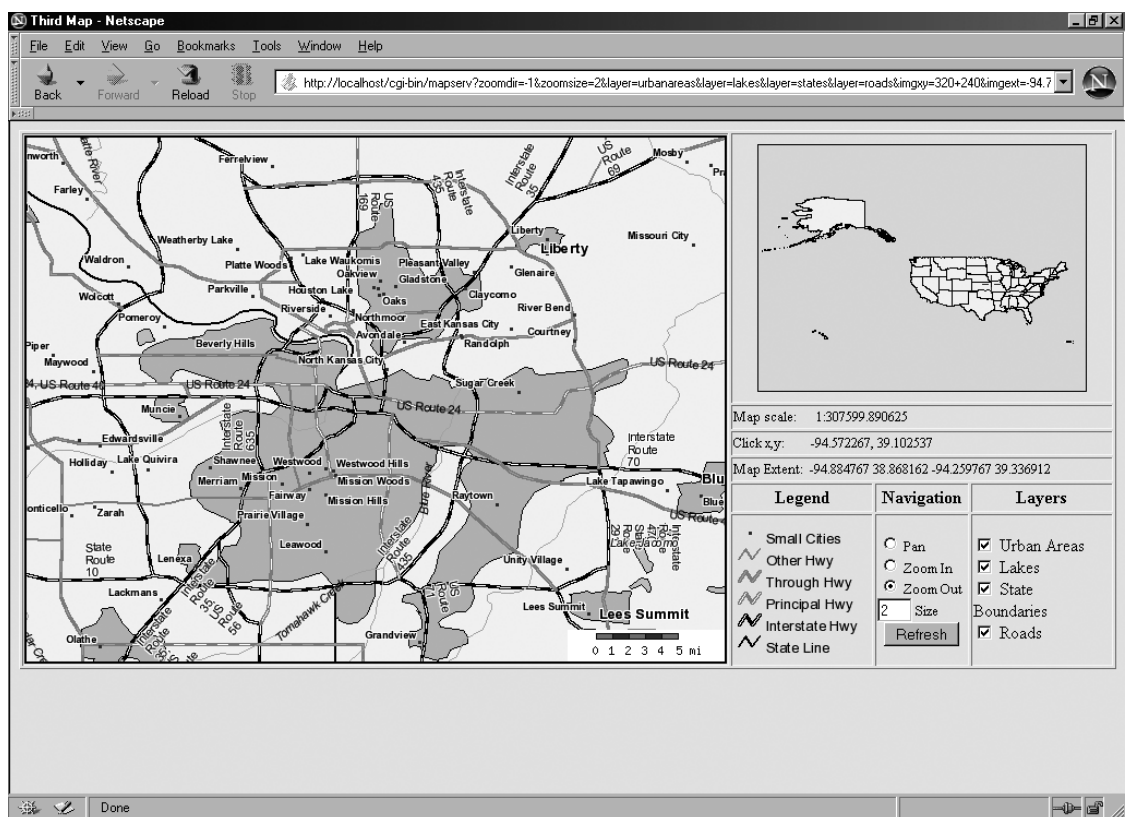


Figure 4-13. Using classes to distinguish features of the same type allows the map maker to indicate graphically things that are perfectly obvious on the ground.

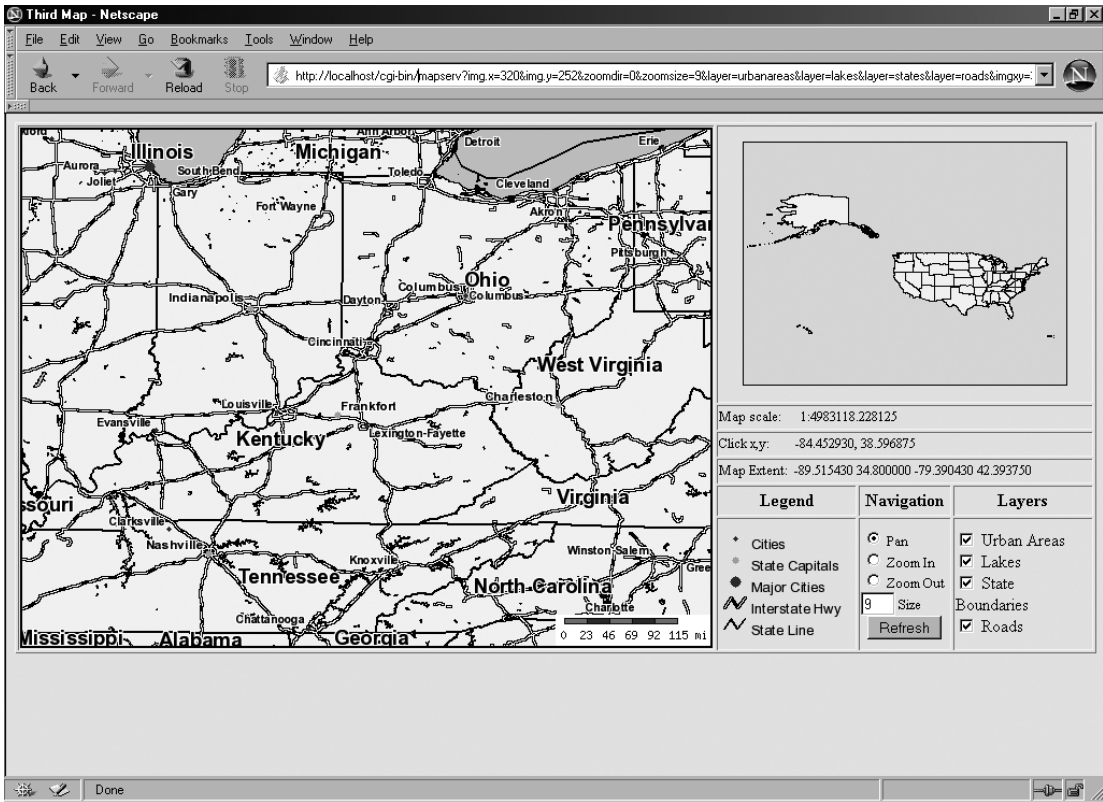


Figure 4-14. Large-scale maps shouldn't be cluttered with details that obscure the relationships between distant places.

In addition to using classes to make graphical distinctions, you might use them to restrict features to be rendered only at specific scales. For example, instead of rendering all the roads in a data set, you could specify that at a scale of 1:7,500,001 and above, only interstate highways will be rendered; and at smaller scales, all roads will be rendered, regardless of type. By creating two layers, one with `MINSCALE` set to 7500001 and the other with `MAXSCALE` set to 7500000, you can select only interstate features for rendering when the scale is greater than 1:7,500,000, and ignore the second layer. When the scale is less than 1:7,500,000, the first layer is ignored and only the second layer (which includes all roads) is rendered. Scale-dependent rendering can also be done at the class level using the keywords `MAXSCALE` and `MINSCALE` within a `CLASS` object.

In this chapter's mapping application, the `majorcities` layer renders two classes of city: state capitals and cities with large populations. The first class, `State Capitals`, is defined in Lines 421 through 437. `CLASSITEM "FEATURE"` (on Line 419) tells MapServer to examine the contents of the `FEATURE` attribute when evaluating the regular expression on Line 323. If the value of `FEATURE` contains the text `STATE` followed by zero or more characters, then that city will be included in the class.

The other class, "Major Cities", is defined by the logical expression on Line 439. If the value contained in the numeric attribute `Pop_2000` is greater than 1,000,000, then the city is included in this class. Since the attribute is numeric, its name is delimited by square brackets only, and the comparison value isn't delimited at all. If the attribute had been string-valued, both the square-bracketed attribute name and the comparison value would need to be quoted.

Note Sometimes the documentation that accompanies spatial data can be wrong. In the present case, the metadata associated with the shapefile `citiesx020` identifies the population field as `POP_2000`, while the actual attribute name is `Pop_2000`. However, as noted previously, MapServer doesn't appear to be case sensitive when dealing with attribute names in shapefiles.

The next two layers each contain a single class defined by a logical expression. The `largecities` layer (Lines 460 through 486) defines a class that contains cities with populations between 100,000 and 1,000,000. The `smallcities` layer defines a class of cities with populations less than 100,000. Although both classes are drawn from the same data set, two layers are used in order to render the two classes at different scales (recall that all the classes in a layer are rendered at the same scale). At the larger scale, the larger cities are drawn and the smaller ones omitted. When the scale moves below 1:1,000,000, both classes are rendered. You want to reduce the amount of detail on the displayed extent of the map to a manageable level—as such, you draw classes that might contain a lot of features when the scale is small because the number actually drawn in the smaller extent is small.

An important point regarding the syntax of logical expressions is shown in the following code:

```
468      CLASS
469          EXPRESSION(([Pop_2000]>100000)and([Pop_2000]<=1000000))
470          NAME "Cities"
```

It has already been stated that logical expressions must be enclosed in parentheses, but parentheses can also be used to group elements of compound expressions. However, this isn't strictly necessary in the example shown.

Using Symbols

In the previous chapter, you saw how to use the class-level keyword `COLOR` to color features. However, MapServer has no way of changing the size of default lines—they're always 1 pixel wide. But by using scalable symbols, you can use the class-level keyword `SIZE` to set the size (in pixels) of the symbol.

A symbol is defined at the map level and is therefore available to all classes in all layers. A symbol starts with the keyword `SYMBOL` and is terminated with the keyword `END`. The keyword `NAME` is used to assign a name to the symbol. The name is used to reference the symbol when it's used. There are several types of symbols: vector, ellipse, pixmap, and truetype.

A vector symbol consists of a series of points that describe the outline of the symbol. The points are defined by using the keyword `POINTS`, followed by coordinate pairs, and terminated by the keyword `END`.

An ellipse symbol uses the same syntax as a vector symbol, but the interpretation is different. The keyword `POINTS` contains only one coordinate pair, and these numbers are interpreted as the relative length (in the x and y directions) of the major and minor axes of the ellipse. If both coordinates are equal, then the ellipse is a circle.

A pixmap symbol uses a GIF or PNG image as a symbol. The file containing this image is identified by the value of the keyword `IMAGE`.

A truetype symbol uses characters from a TrueType font as symbols. The font is identified by the keyword `FONT`, and its value is the alias of a font specified in the `FONTSET`. The particular character is specified by the keyword `CHARACTER`. If the symbol `TYPE` is `truetype`, then the keyword `ANTIALIAS` can be set to `true` or `false` to turn antialiasing on or off for the symbol.

A dash pattern or style can be set with the keyword `STYLE`. The value associated with style is a sequence of integers. The first integer specifies the number of pixels drawn (also referred to as “pixels on”), the next integer specifies the number of pixels of space, the following specifies the number of pixels on, etc.

A symbol can be rendered as an outline or a filled polygon. Setting the keyword `FILLED` to `true` will cause the symbol to be filled with the color specified by the class that uses it.

Although all the symbol definitions can be placed in an external file (identified to MapServer by the value of the keyword `SYMBOLSET`), I’ve chosen to keep things simple by placing them in the mapfile. Three symbols are defined, all of `TYPE ellipse`.

The symbol `BigLine` (Lines 014 through 018) is used for drawing roads. Since the coordinate values of the keyword `POINTS` are equal, using `BigLine` to render a point will draw a circle. Since the default value of `FILLED` is `false`, the circle won’t be filled. However, when drawing a line feature with `BigLine`, successive open circles overlap, effectively filling the symbol. The default value for `STYLE` will produce a continuous line.

```
014 SYMBOL
015     NAME "BigLine"
016     TYPE ELLIPSE
017     POINTS 1 1 END
018 END
```

The next symbol, `DashedLine`, differs from `BigLine` only in `STYLE`. Setting `STYLE 10 10 END` produces a dashed line alternating between 10 pixels on and 10 pixels off.

```
023 SYMBOL
024     NAME "DashedLine"
025     TYPE ELLIPSE
026     POINTS 1 1 END
027     STYLE 10 10 END
028 END
```

When used together, `BigLine` and `DashedLine` produce some interesting effects that serve to distinguish major roadways from other line features. The small-scale highway segments of layer `interstate2` are drawn as 3-pixel-wide black lines (Lines 232 through 267). On top of these lines MapServer draws intermittent, 1-pixel-wide white lines (representing white lane separators). The `OVERLAY` keywords specify the characteristics of the overlaid symbols. Similarly,

principal roads are drawn as brown, 3-pixel-wide `BigLine` symbols, overlaid with white, intermittent, 1-pixel-wide `DashedLine` symbols. The keywords `OVERLAYSYMBOL`, `OVERLAYSIZE`, and `OVERLAYCOLOR` work the same way as `SYMBOL`, `SIZE`, and `COLOR`.

```

245         STYLE
246             SYMBOL "BigLine"
247             SIZE 3
248             COLOR 0 0 0
249         END
250     OVERLAYSYMBOL "DashedLine"
251     OVERLAYSIZE 1
252     OVERLAYCOLOR 255 255 255

```

As shown in the following code snippet, the symbol `Circle` is used to draw a point feature. The default `FILL` is empty, therefore `FILLED true` has been specified so that a solid-colored spot will be drawn.

```

033 SYMBOL
034     NAME "Circle"
035     FILLED true
036     TYPE ellipse
037     POINTS 1 1 END
038 END

```

Using Annotation Layers

In some situations, you may only want to render the labels in a layer, rather than the features. This can be for a number of reasons. You may want to display a raster image (e.g., an aerial photograph or something like it), and you want to avoid rendering a labeled polygon layer on top of it, which would obscure the raster layer. Alternatively, you may be required to render a layer multiple times, but only need to render the labels once. A layer in which only the labels are drawn is called an annotation layer.

In the present case, there are three layers devoted to the state boundary data set. The first layer (Lines 101 through 113) renders the state outlines. Since this is a polygon layer, the color specified by the keyword `COLOR` is the fill color for each of the individual state polygons. There is no border drawn. In order to better see the states, you could use the keyword `OUTLINECOLOR` to set the color of a 1-pixel-wide line that would outline these polygons, but a 1-pixel-wide line isn't very distinctive, so a wider line was chosen.

```

101 LAYER
102     NAME "states"
103     DATA "statesp020"
104     STATUS on
105     TYPE polygon
106     LABELCACHE on
107     LABELITEM "STATE"

```

```

108     CLASS
109         STYLE
110             COLOR 255 246 189
111         END
112     END
113 END

```

In order to create a wider line to outline the states, you can create another layer (Lines 118 through 133) that references the same data set but is defined as a line layer. The color specified for a line layer is the color of the line, not the fill color for the area that the line encloses (if it *is* closed). Furthermore, you can specify a symbol to use for the line and make this symbol as wide as you like. Note here that in Line 128, you're using the `BigLine` symbol defined in the `SYMBOL` object in Lines 014 through 018. Line 129 sets the symbol `SIZE` to 2 pixels wide and Line 130 sets the `COLOR` to black.

```

118 LAYER
119     NAME "states"
120     DATA "statesp020"
121     STATUS on
122     TYPE line
123     LABELCACHE on
124     LABELITEM "STATE"
125     CLASS
126         NAME "State Line"
127         STYLE
128             SYMBOL "BigLine"
129             SIZE 2
130             COLOR 0 0 0
131         END
132     END
133 END

```

These two layers will produce a polygon in the shape of the United States, with a light-brown fill color and state boundaries marked by a fat, black line. If you try to label the line layer, the labels (in this case, the state names) will be lined up along the lower or upper borders of each state. Recall that the label associated with a line feature can only assume the `uc` or `lc` position. However, you want your labels to lie inside each state's boundary. To accomplish this, you define an annotation layer (Lines 521 through 540). The keyword `TYPE` is set to `annotation` in Line 525. The rest of the keywords in the layer should be familiar to you, except for `LABELMINSIZE` in Line 528. `LABELMINSIZE` allows you to specify the minimum scale at which a label will be rendered. This is important because at small scales, the labels of features with large extents can be intrusive. For example, if you're looking at downtown Des Moines at a scale of 1:20,000, you don't need to see the word "Iowa." But when looking at the entire continental United States at a scale of 1:10,000,000, knowing that the postage stamp-sized state at the center is Iowa can be useful. This problem is solved by using `LABELMINSIZE`, which is employed on Line 528.

```

521 LAYER
522     NAME "states"
523     DATA "statesp020"
524     STATUS on
525     TYPE annotation
526     LABELCACHE on
527     LABELITEM "STATE"
528     LABELMINSIZE 2000000
529     CLASS
530         LABEL
531             TYPE truetype
532             FONT "arialbd"
533             SIZE 14
534             OUTLINECOLOR 255 255 255
535             COLOR 0 0 0
536             MINDISTANCE 200
537             MINFEATURESIZE 10
538         END
539     END
540 END

```

Creating Scale Bars

The SCALEBAR object is MapServer's method of creating scale bars (i.e., graphic images that indicate how distances on the map relate to distances on the ground). These images can either stand alone within their own HTML tags defined in the template file, or they can be embedded in the map image itself. The size of the stand-alone image can't be determined before the image is created, so avoid specifying image size in the HTML image tag (). The SCALEBAR object begins with the keyword SCALEBAR, is terminated by the keyword END, and contains keywords that control the appearance and location of the bar. Figure 4-13 displays a scale bar at the lower-right corner of the map image.

For this application, the scale bar is defined in Lines 064 through 081. Inside the scale bar, a LABEL object is defined in order to specify the font size and color of the text associated with the scale. Note that the label in a scale bar can't use TrueType fonts, but otherwise works as I've described previously.

```

064 SCALEBAR
065     LABEL
066         COLOR 0 0 0
067         ANTIALIAS true
068         SIZE small
069     END

```

```
070     POSITION lr
071     INTERVALS 5
072     STATUS embed
073     SIZE 144 5
074     STYLE 0
075     UNITS miles
076     BACKGROUND_COLOR 255 0 0
077     IMAGE_COLOR 255 255 255
078     COLOR 128 128 128
079     OUTLINE_COLOR 0 0 255
080     TRANSPARENT off
081 END
```

The keyword `STATUS` can take three values: `on`, `off`, or `embed`. As with layers, `STATUS on` indicates that a separate scale bar image will be created and `STATUS off` indicates that no image will be created. Keep in mind that although MapServer can create a scale bar image, the image won't display if you don't reference it via the `[scalebar]` substitution string in the HTML template. If you prefer to embed the scale bar in the map itself instead of creating a separate image, set `STATUS` to `embed`.

An embedded scale bar can be located in one of six positions. The position is indicated by setting the value of the keyword `POSITION` to one of the following two-letter codes: `ul`, `uc`, `ur`, `ll`, `lc`, or `lr` ("upper-left," "upper-center," etc.—these values have the same meanings in this context as when they're used to specify label positions, as shown in Figure 4-12).

The keyword `SIZE` is assigned a pair of integer values that represent the width and height (in pixels) of the scale bar image. The value assigned to the keyword `INTERVAL` specifies the number of intervals into which the bar is broken. The keyword `STYLE` can take the values 0 or 1. `STYLE 0` produces a solid bar while `STYLE 1` produces a tick-marked line. The keyword `UNITS` can take the values `feet`, `inches`, `kilometers`, `meters`, and `miles`. Based on the value selected, MapServer will calculate the length of the scale bar intervals and label them appropriately. The scale bar shown on the map in Figure 4-15 uses `STYLE 1`. Compare this with `STYLE 0` in Figure 4-13.

There are several color-related keywords used in the `SCALEBAR` object, and their names don't always indicate their functions. The keyword `IMAGE_COLOR` specifies the background color of the scale bar image, which includes the bar itself and any text labels. (If the keyword `TRANSPARENT` is set to `on`, then the `IMAGE_COLOR` will be transparent.) The keyword `BACKGROUND_COLOR` sets the background color of the bar—not the text labels. The keyword `COLOR` is used to set the foreground color of the scale bar. If more than a single interval is specified, then the `COLOR` and `BACKGROUND_COLOR` will alternate along the length of the bar. Finally, the keyword `OUTLINE_COLOR` specifies the color of the 1-pixel-wide border around the bar (not including any label text).

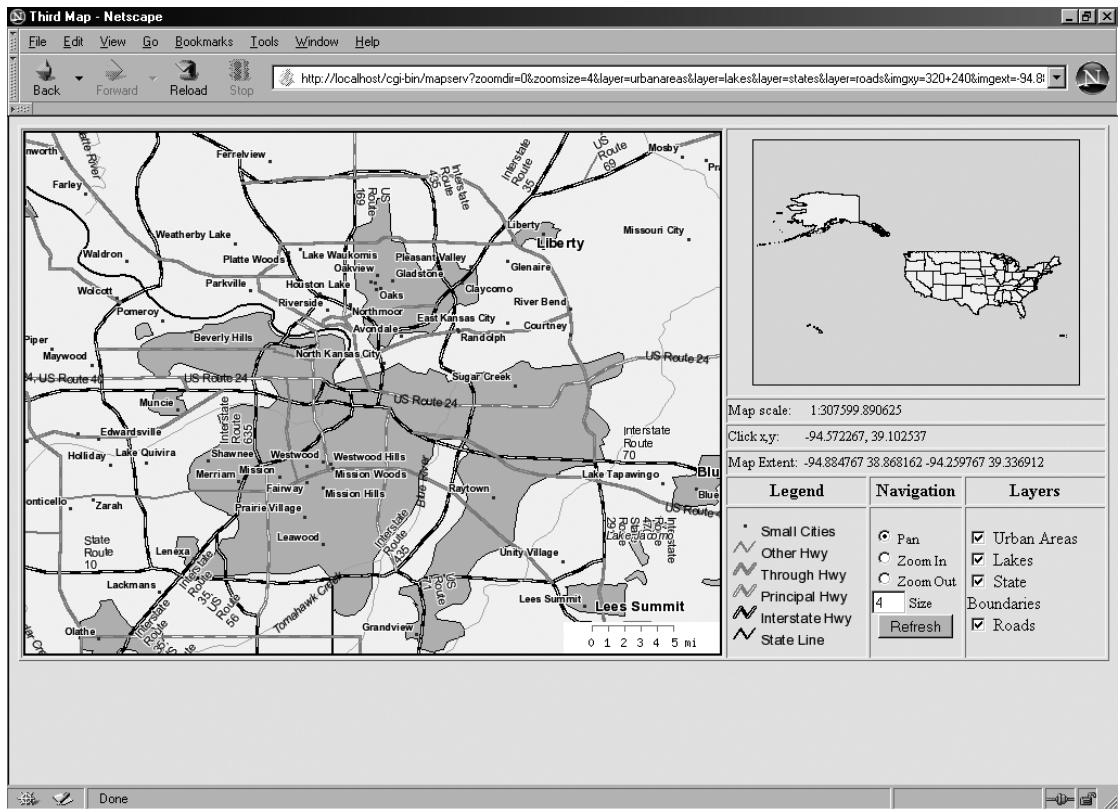


Figure 4-15. A STYLE 1 scale bar presents with a gracile simplicity absent from its more robust STYLE 0 counterpart.

Creating Legends

While scale bars provide a way of relating map features to real-world features, legends present a graphical reminder of what the different map symbols mean. In MapServer, the only symbols that will be shown in the legend are the symbols that are used by *named* classes. Recall that a class doesn't need to be named in order to be rendered—it must, however, have a name in order to be included in the legend. As with scale bar images, legend images can stand alone or they can be embedded in the map image. Also like scale bar images, the size of the stand-alone legend image can't be determined before the image is created, so you should avoid specifying image size in the HTML `` tag. A legend begins with the keyword `LEGEND` and is terminated by the keyword `END`. Figure 4-16 shows an embedded legend. If you look back at previous examples, you'll see that they've all been external legends.

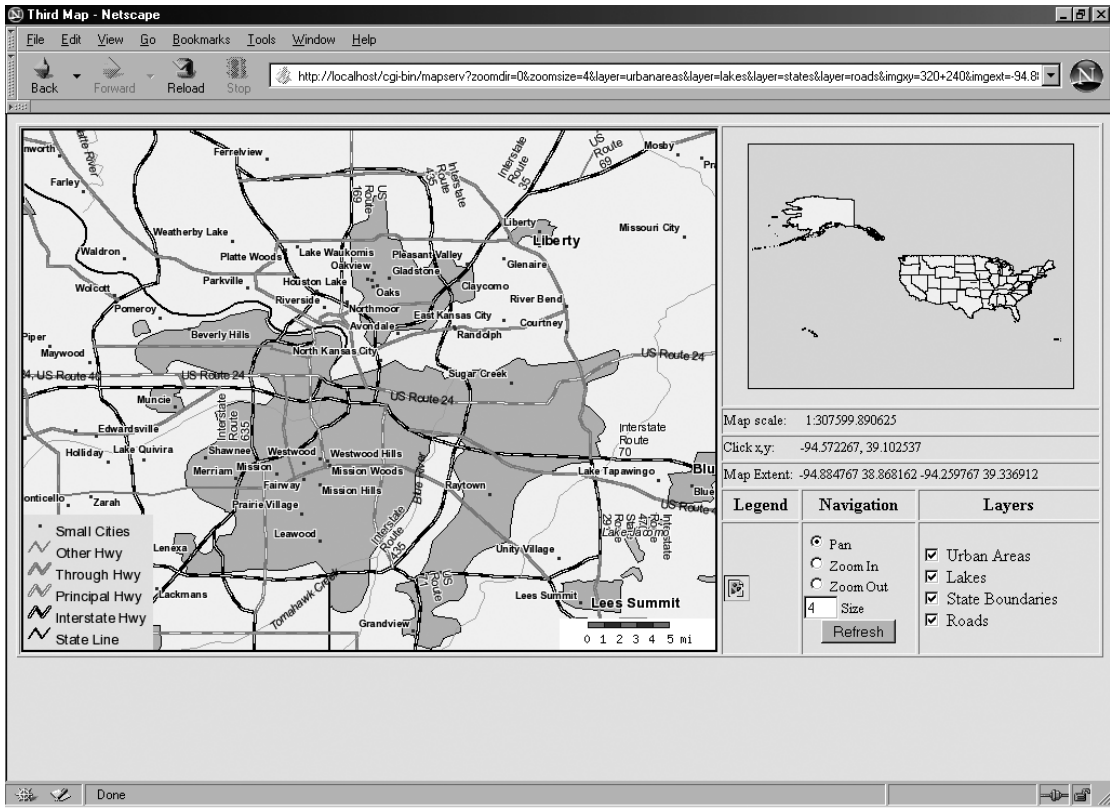


Figure 4-16. An embedded legend is an artifact from the pre-digital past, when cartographers were compelled by the technical limitations of paper to obscure features with useful but graphically extraneous elements. External legends are preferred.

In this application, the legend is defined in Lines 086 through 096. The value of the keyword `STATUS` is on so that MapServer will produce a stand-alone legend image. If `STATUS` were off, no image would be created, and if `STATUS` were set to `embed`, then the legend would be embedded in the map image.

Legends can use TrueType fonts, and in this way, the `LABEL` objects in a legend work the same way as labels in classes. The background color of the legend image is set with the keyword `BACKGROUNDCOLOR`.

```

086  LEGEND
087      STATUS on
088      IMAGECOLOR 230 230 230
089      LABEL
090          TYPE truetype
091          FONT "arial"
092          COLOR 0 0 0
093          SIZE 10

```

```
094          ANTIALIAS true
095          END
096      END
```

Using Reference Maps

One of the problems noted with the second map was that at small scales, it was difficult to place the displayed map extent into a larger context. MapServer solves this problem by allowing the use of a *reference* map, which displays the entire initial extent of the map at all times while indicating the currently displayed extent by outlining it with a rectangular box. (At scales that are so small that the box would be invisible, MapServer uses a crosshair to show the location of the extent.) The reference map may be any size you like, but it's usually best to keep it small. Figure 4-14 shows an example of a reference map with the reference box outlined in red, while Figure 4-15 shows the crosshair.

A reference map begins with the keyword `REFERENCE` and is terminated by the keyword `END`. A reference map needs an image to use as the map. This image is identified by the keyword `IMAGE`, and its value is the path (absolute or relative to the location of the mapfile) to the reference image. The reference image must be a GIF. The size of the image (in pixels) is specified by the keyword `SIZE`, and the extent of the image is set using the keyword `EXTENT`, in the same way that the extent is set for the map itself.

If a reference map is to be generated, then the value of the keyword `STATUS` is set to `on`; if not, it's set to `off`. The color of the reference box is set by the keyword `COLOR`. The box will be filled with this color unless one of the components of the color is set to `-1`, in which case the box will be transparent. The `OUTLINECOLOR` keyword sets the color of the reference box.

The reference map for this application is defined in Lines 052 through 059. The reference image is `usaref.gif` (this file is included in the source code distribution)—it's 300 pixels wide by 225 pixels high. Since at least one of the components of the keyword `COLOR` is set to `-1`, the fill color of the reference box is transparent. The `OUTLINECOLOR` of the box is red.

```
052  REFERENCE
053      IMAGE "/var/www/htdocs/third_usaref.gif"
054      SIZE 300 225
055      EXTENT -180.00 0.00 -60.00 90.00
056      STATUS ON
057      COLOR -1 -1 -1
058      OUTLINECOLOR 255 0 0
059  END
```

Note MapServer itself can be used to create the reference image. By specifying the image size and limited details in a mapfile, MapServer can render a map with appropriate extent and image size. In a later chapter, you'll see how to do this with the utility program `shp2img`, which comes with the MapServer distribution.

Summary

This chapter has led you through the intricacies of designing effective and visually pleasing maps through the manipulation of their graphical elements. When you've mastered the contents of this chapter, you'll be able to create interactive, web-based mapping applications that can display all sorts of data that possesses spatial distribution. Although these maps are useful, you haven't yet wrung *all* the utility from MapServer's CGI interface.

Up to this point, the applications you've created have produced images and allowed you to explore those images at any scale. You haven't yet been able to query the spatial data sets, however, and gain access to the wealth of information stored in them. The next chapter provides a detailed description of the MapServer query facility and takes you through the steps required to build an application that will give you access to all of MapServer's query power.

The Code

This application uses a lot more of MapServer's capabilities and therefore the mapfile is much longer. However, both the HTML initialization file and the template file are substantially the same as those of the previous application. The mapfile is named `third.map`. The initialization and template files are named `third_i.html` and `third.html`, respectively. The listings are as follows:

Listing 4-1. *The mapfile third.map*

```
001 # This is our third map file
002 NAME "third"
003 UNITS DD
004 EXTENT -180.00 0.00 -60.00 90.00
005 SIZE 640 480
006 IMAGECOLOR 189 202 222
007 IMAGETYPE PNG
008 SHAPEPATH "/home/mapdata"
009 FONTSET "/var/www/htdocs/fontset.txt"
010
011 #####
012 # Symbol for drawing fat lines
013 #
014 SYMBOL
015     NAME "BigLine"
016     TYPE ELLIPSE
017     POINTS 1 1 END
018 END
019
020 #####
021 # Symbol for drawing dashed lines
022 #
023 SYMBOL
024     NAME "DashedLine"
025     TYPE ELLIPSE
```

```

026     POINTS 1 1 END
027     STYLE 10 10 END
028 END
029
030 #####
031 # Symbol for drawing spots
032 #
033 SYMBOL
034     NAME "Circle"
035     FILLED true
036     TYPE ellipse
037     POINTS 1 1 END
038 END
039
040 #####
041 # Web object
042 #
043 WEB
044     TEMPLATE "/var/www/htdocs/third.html"
045     IMAGEPATH "/var/www/htdocs/tmp/"
046     IMAGEURL "/tmp/"
047 END
048
049 #####
050 # Reference map
051 #
052 REFERENCE
053     IMAGE "/var/www/htdocs/third_usaref.gif"
054     SIZE 300 225
055     EXTENT -180.00 0.00 -60.00 90.00
056     STATUS ON
057     COLOR -1 -1 -1
058     OUTLINECOLOR 255 0 0
059 END
060
061 #####
062 # Scalebar
063 #
064 SCALEBAR
065     LABEL
066         COLOR 0 0 0
067         ANTIALIAS true
068         SIZE small
069     END
070     POSITION lr
071     INTERVALS 5
072     STATUS embed

```

```

073     SIZE 144 5
074     STYLE 0
075     UNITS miles
076     BACKGROUND_COLOR 255 0 0
077     IMAGE_COLOR 255 255 255
078     COLOR 128 128 128
079     OUTLINE_COLOR 0 0 255
080     TRANSPARENT off
081 END
082
083 #####
084 # Legend
085 #
086 LEGEND
087     STATUS on
088     IMAGE_COLOR 230 230 230
089     LABEL
090         TYPE truetype
091         FONT "arial"
092         COLOR 0 0 0
093         SIZE 10
094         ANTIALIAS true
095     END
096 END
097
098 #####
099 # State boundaries layer - polygon used for shading
100 #
101 LAYER
102     NAME "states"
103     DATA "statesp020"
104     STATUS on
105     TYPE polygon
106     LABELCACHE on
107     LABELITEM "STATE"
108     CLASS
109         STYLE
110             COLOR 255 246 189
111         END
112     END
113 END
114
115 #####
116 # State boundaries layer - line makes a fat boundary
117 #
118 LAYER
119     NAME "states"

```

```

120     DATA "statesp020"
121     STATUS on
122     TYPE line
123     LABELCACHE on
124     LABELITEM "STATE"
125     CLASS
126         NAME "State Line"
127         STYLE
128             SYMBOL "BigLine"
129             SIZE 2
130             COLOR 0 0 0
131     END
132 END
133 END
134
135 #####
136 # Urban areas layer
137 #
138 LAYER
139     NAME "urbanareas"
140     DATA "urbanap020"
141     STATUS on
142     TYPE polygon
143     LABELCACHE on
144     LABELITEM "NAME"
145     MAXSCALE 1999999
146     CLASS
147         STYLE
148             COLOR 212 192 100
149             OUTLINECOLOR 0 0 0
150         END
151         LABEL
152             TYPE truetype
153             FONT "arialbd"
154             SIZE 10
155             OUTLINECOLOR 255 255 255
156             COLOR 0 0 0
157             POSITION auto
158         END
159     END
160 END
161
162 #####
163 # hydrographic layer - lakes
164 #
165 LAYER
166     NAME "lakes"

```

```

167     DATA "hydrogp020"
168     STATUS on
169     TYPE polygon
170     LABELCACHE on
171     LABELITEM "NAME"
172     CLASSITEM "FEATURE"
173     CLASS
174         EXPRESSION ('[FEATURE]' eq 'Lake')
175         STYLE
176             SIZE 1
177             COLOR 189 202 222
178             OUTLINECOLOR 0 0 0
179         END
180         LABEL
181             TYPE truetype
182             FONT "arial"
183             SIZE 8
184             OUTLINECOLOR 255 255 255
185             COLOR 0 0 0
186             MINDISTANCE 100
187             POSITION lr
188             MINFEATURESIZE auto
189             WRAP ' '
190         END
191     END
192 END
193
194 #####
195 # Road layer - interstates only at large scale
196 #
197 LAYER
198     NAME "interstate1"
199     GROUP "roads"
200     DATA "roadtrl020"
201     STATUS on
202     TYPE line
203     LABELCACHE on
204     LABELITEM "NAME"
205     CLASSITEM "FEATURE"
206     MINSIZE 7500001
207     CLASS
208         NAME "Interstate Hwy"
209         EXPRESSION /Limited Access*/
210         STYLE
211             SYMBOL "BigLine"
212             SIZE 1
213             COLOR 0 0 0

```

```

214         END
215         LABEL
216             TYPE truetype
217             FONT "arial"
218             SIZE 8
219             OUTLINECOLOR 255 255 255
220             COLOR 0 0 0
221             MINDISTANCE 200
222             POSITION auto
223             ANGLE auto
224             MINFEATURESIZE 50
225         END
226     END
227 END
228
229 #####
230 # Road layer - interstates only
231 #
232 LAYER
233     NAME "interstate2"
234     GROUP "roads"
235     DATA "roadtrl020"
236     STATUS on
237     TYPE line
238     LABELCACHE on
239     LABELITEM "NAME"
240     CLASSITEM "Feature"
241     MAXSCALE 7500000
242     CLASS
243         NAME "Interstate Hwy"
244         EXPRESSION /Limited Access*/
245         STYLE
246             SYMBOL "BigLine"
247             SIZE 3
248             COLOR 0 0 0
249         END
250         OVERLAYSYMBOL "DashedLine"
251         OVERLAYSIZE 1
252         OVERLAYCOLOR 255 255 255
253 #     TEXT ([FEATURE],[NAME])
254     LABEL
255         TYPE truetype
256         FONT "arial"
257         SIZE 8
258         OUTLINECOLOR 255 255 255
259         COLOR 0 0 0
260         MINDISTANCE 200

```



```

261             POSITION auto
262             ANGLE auto
263             MINFEATURESIZE 50
264             WRAP ' '
265         END
266     END
267 END
268
269 #####
270 # Road layer - principal highways
271 #
272 LAYER
273     NAME "principal"
274     GROUP "roads"
275     DATA "roadtrl020"
276     STATUS on
277     TYPE line
278     LABELCACHE on
279     LABELITEM "NAME"
280     CLASSITEM "Feature"
281     MAXSCALE 4000000
282     CLASS
283         NAME "Principal Hwy"
284         EXPRESSION /Principal Highway*/
285         STYLE
286             SYMBOL "BigLine"
287             SIZE 3
288             COLOR 197 129 65
289         END
290         OVERLAYSYMBOL "DashedLine"
291         OVERLAYSIZE 1
292         OVERLAYCOLOR 255 255 255
293     # TEXT ([FEATURE],[NAME])
294     LABEL
295         TYPE truetype
296         FONT "arial"
297         ANGLE auto # requires ttfonts
298         MINFEATURESIZE 50
299         MINDISTANCE 100
300         ANGLE auto
301         COLOR 0 0 0
302         SIZE 8
303     END
304 END
305 END
306
307 #####

```

```

308 # Road layer - other through highways
309 #
310 LAYER
311     NAME "thru"
312     GROUP "roads"
313     DATA "roadtrl020"
314     STATUS on
315     TYPE line
316     LABELCACHE on
317     LABELITEM "NAME"
318     CLASSITEM "Feature"
319     MAXSCALE 1000000
320     CLASS
321         NAME "Through Hwy"
322         EXPRESSION /Other Through*/
323         STYLE
324             SYMBOL "BigLine"
325             SIZE 3
326             COLOR 197 129 65
327         END
328         OVERLAYSYMBOL "DashedLine"
329         OVERLAYSIZE 1
330         OVERLAYCOLOR 0 255 0
331         LABEL
332             TYPE truetype
333             FONT "arial"
334             ANGLE auto # requires ttfonts
335             MINFEATURESIZE 100
336             MINDISTANCE 100
337             ANGLE auto
338             COLOR 0 0 0
339             SIZE 8
340         END
341     END
342 END
343
344 #####
345 # Road layer - other highways
346 #
347 LAYER
348     NAME "other"
349     GROUP "roads"
350     DATA "roadtrl020"
351     STATUS on
352     TYPE line
353     LABELCACHE on
354     LABELITEM "NAME"

```

```

355     CLASSITEM "FEATURE"
356     MAXSCALE 500000
357     CLASS
358         NAME "Other Hwy"
359         EXPRESSION /Other Highway*/
360         STYLE
361             SYMBOL "BigLine"
362             SIZE 2
363             COLOR 197 129 65
364         END
365         LABEL
366             TYPE truetype
367             FONT "arial"
368             ANGLE auto # requires ttfonds
369             MINFEATURESIZE 100
370             MINDISTANCE 100
371             ANGLE auto
372             COLOR 0 0 0
373             SIZE 6
374         END
375     END
376 END
377
378 #####
379 # Hydrographic layer - streams & rivers
380 #
381 LAYER
382     NAME "rivers"
383     DATA "hydrogl020"
384     STATUS DEFAULT
385     TYPE line
386     LABELCACHE on
387     LABELITEM "NAME"
388     CLASSITEM "FEATURE"
389     MAXSCALE 1000000
390     CLASS
391         EXPRESSION ('[FEATURE]' eq 'Stream')
392         STYLE
393             SYMBOL "BigLine"
394             SIZE 1
395             COLOR 156 182 205
396         END
397         LABEL
398             TYPE truetype
399             FONT "arial"
400             COLOR 0 0 0
401             ANGLE auto

```

```

402             SIZE 7
403             ANTIALIAS true
404         END
405     END
406 END
407
408 #####
409 # Cities layer - State capitals and cities pop. > 1000000
410 #
411 LAYER
412     NAME "majorcities"
413     DATA "citiesx020"
414     STATUS default
415     TYPE point
416     LABELITEM "NAME"
417     LABELCACHE on
418     LABELMAXSCALE 15000000
419     CLASSITEM "FEATURE"
420     MINSCALE 2000000
421     CLASS
422         NAME "State Capitals"
423         EXPRESSION /State*/
424         STYLE
425             SYMBOL "Circle"
426             SIZE 6
427             COLOR 0 255 0
428         END
429         LABEL
430             TYPE truetype
431             FONT "arialbd"
432             SIZE 9
433             POSITION auto
434             OUTLINECOLOR 255 255 255
435             COLOR 0 0 0
436         END
437     END
438     CLASS
439         EXPRESSION ([Pop_2000] > 1000000)
440         NAME "Major Cities"
441         STYLE
442             SYMBOL "Circle"
443             SIZE 10
444             COLOR 255 0 0
445         END
446         LABEL
447             TYPE truetype
448             FONT "arialbd"

```

```

449             SIZE 10
450             OUTLINECOLOR 255 255 255
451             COLOR 0 0 0
452             POSITION auto
453         END
454     END
455 END
456
457 #####
458 # Cities layer - Large cities, pop. < 1000000
459 #
460 LAYER
461     NAME "largecities"
462     DATA "citiesx020"
463     STATUS default
464     TYPE point
465     LABELCACHE on
466     LABELITEM "NAME"
467     MINSCALE 2000000
468     CLASS
469         EXPRESSION(([Pop_2000]>100000)and([Pop_2000]<=1000000))
470         NAME "Cities"
471         STYLE
472             SYMBOL "Circle"
473             SIZE 4
474             COLOR 255 0 0
475             BACKGROUNDCOLOR 255 0 0
476         END
477         LABEL
478             TYPE truetype
479             FONT "arialbd"
480             SIZE 8
481             POSITION auto
482             OUTLINECOLOR 255 255 255
483             COLOR 0 0 0
484         END
485     END
486 END
487
488 #####
489 # City layer - Cities
490 LAYER
491     NAME "cities"
492     DATA "citiesx020"
493     STATUS default
494     TYPE point
495     LABELCACHE on

```

```

496 LABELITEM "NAME"
497 LABELMAXSCALE 500000
498 MAXSCALE 2000000
499 CLASS
500     NAME "Small Cities"
501     EXPRESSION ([Pop_2000] < 100000)
502     STYLE
503         SYMBOL "Circle"
504         SIZE 3
505         COLOR 255 0 0
506     END
507     LABEL
508         TYPE truetype
509         FONT "arialbd"
510         SIZE 7
511         POSITION auto
512         OUTLINECOLOR 255 255 255
513         COLOR 0 0 0
514     END
515 END
516 END
517
518 #####
519 # State boundaries layer - annotation (for labels)
520 #
521 LAYER
522     NAME "states"
523     DATA "statesp020"
524     STATUS on
525     TYPE annotation
526     LABELCACHE on
527     LABELITEM "STATE"
528     LABELMINSCALE 2000000
529     CLASS
530         LABEL
531             TYPE truetype
532             FONT "arialbd"
533             SIZE 14
534             OUTLINECOLOR 255 255 255
535             COLOR 0 0 0
536             MINDISTANCE 200
537             MINFEATURESIZE 10
538         END
539     END
540 END
541 END # mapfile

```



```

030         Zoom Out<BR>
031     <input type="text" name="zoomsize" size=1 value=[zoomsize]>
032         Size<BR>
033     <center><input type="submit" value="Refresh"></center>
034 </td>
035 <td align="top">
036     <input type="checkbox" name="layer" value="urbanareas"
037         [urbanareas_check]>
038         Urban Areas<BR>
039     <input type="checkbox" name="layer" value="lakes" [lakes_check]>
040         Lakes<BR>
041     <input type="checkbox" name="layer" value="states" [states_check]>
042         State Boundaries<BR>
043     <input type="checkbox" name="layer" value="roads" [roads_check]>
044         Roads<BR></font>
045 </td>
046 </tr>
047 </table>
048 <input type="hidden" name="imgxy" value="320 240">
049 <input type="hidden" name="imgext" value="[mapext]">
050 <input type="hidden" name="map" value="[map]">
051 <input type="hidden" name="program" value="[program]">
052 </form>
053 </body>
054 </html>

```




Using Query Mode

In addition to its rendering capabilities, MapServer provides a powerful query facility, supporting both spatial queries (which select features based on location) and attribute queries (which select features based on attribute values). To accomplish this without programmatic support, MapServer makes extensive use of templates in building queries and presenting the results. This leads to some complicated interactions between the mapfile, the templates, and the MapServer program. Added to this is the large number of different query modes, which employ templates in different ways. For these reasons, the query facility is probably the most confusing aspect of MapServer.

In this chapter, you'll explore MapServer's query capabilities and become familiar with its mapfile and template requirements. To gain knowledge of these capabilities, you'll build a query application that uses most of MapServer's query muscle. Because this topic can be so confusing, I'll present the various components of a query first, and then explain them in some detail. Then I'll lead you through the application line-by-line and provide additional details.

How MapServer Processes a Query

Until now, the maps you've created have all been generated in Browse mode. In this mode, when the user clicks the map image, the browser notes the location of the click, the zoom size, and the direction (among other things). This information is sent back to the server, which then forwards it on to MapServer. Based on this response from the browser, MapServer renders the map image, scans the template for substitution strings, and sends this information back to the browser. So, although MapServer has access to the underlying spatial and attribute data, this data is hidden from the user. However, the user can gain access to this information using MapServer's query facility, which can present results in both tabular form and graphically, as a map.

The following sections explain the concepts necessary for an operational understanding of the MapServer query process. While detailed usage and syntax is described when warranted, this section functions primarily as an overview of the main components. A more detailed treatment of some of these topics will be presented later in this chapter, including a more detailed analysis of the code.

Query Types

Browse mode is MapServer's default mode—as such, none of the maps you've created previously have made explicit mode references. But MapServer has numerous modes, 18 of which are query modes of one sort or another. In these modes, users can define the query selection criteria in one of the following ways:

- By selecting a point or region on the map image with the mouse
- By entering the coordinates of a point or region
- By entering an expression specifying attribute criteria
- By entering the shape index (i.e., sequence number) of a feature

In the query modes, the browser sends the coordinates or expression and other form variables back to MapServer, but instead of rendering the map image as it would in Browse mode, MapServer searches one or more layers and populates template files with information derived from features or attributes that match the query parameters. MapServer can perform both spatial queries (based on coordinates) and attribute queries (based on attribute values).

When performing the simplest type of spatial query, MapServer notes the location of a mouse click and selects matching *features* based on proximity to the click point. But MapServer isn't restricted to point queries—supplying a spatial extent (or even an arbitrary shape) instead of a point will return features that are within or close to the specified extent (or shape).

In the attribute query modes, MapServer searches the attribute table for features that match a user-specified expression. This expression employs the same syntax used by the class-level keyword `EXPRESSION` and the layer-level keyword `FILTER`. MapServer also supports staged queries, in which spatial queries are performed on the result set returned by an attribute search.

In order for MapServer to perform a query, the mapfile must contain at least one *queriable* layer. A layer qualifies as queriable if it's active (i.e., its `STATUS` is on or default) and it contains a template reference (specified at either the layer level or the class level by the keyword `TEMPLATE`). If a queriable layer exists, MapServer searches the data set feature-by-feature for those that meet the specified spatial or attribute criteria. If a template exists to present a matching feature, the feature is incorporated into the result set—otherwise it's dropped. The existence of a layer-level template guarantees that every feature in the layer that meets the search criteria will be selected. On the other hand, if a layer template hasn't been specified, classes without templates will return no results.

Caution It's possible to define query criteria that should produce a match but don't. For example, a layer that contains multiple classes but specifies a query template for only one class will be queriable (if it's active). However, it will return results only for members of the class with the template. Features that match the spatial or attribute criteria, but for which there's no query template, will be dropped.

Some modes search only a single layer and return a single result—the modes `QUERY` and `ITEMQUERY`, for example. Others, such as `NQUERY` and `ITEMNQUERY`, search all queriable layers and return all results (these types all include `NQUERY` in their names). These searches can be limited to a single layer by setting the value of the form variable `qlayer` (i.e., the query layer) to the name of the layer to be queried—if this isn't done, MapServer will search all queriable layers. Layers are searched in the sequence in which they're specified in the mapfile. In addition to this, the feature query modes use a select layer (which must be of `TYPE` polygon) specified by the form variable `slayer`. A spatial query is performed on the select layer and returns one or more polygon features. Subsequently, other queriable layers are searched, and any features that are found within (or close to) the selected polygon are returned as well.

Query Templates

As query results are returned, MapServer inserts them into one or more HTML template files and forwards them back to the browser. MapServer doesn't know in advance what the result set of any query might be—and since query modes will return either the first match or all matches depending on the mode, MapServer must structure its output in a manner that's sufficiently general to handle both cases. It does this by specifying a hierarchy of query templates.

Query templates can be defined at the map level, the layer level, the class level, and the join level. The level at which a template is specified affects how and when MapServer will use it. The structure of the query template hierarchy is shown in Figure 5-1. At the map level, a query template is used to present a summary of the entire query, (e.g., information like the total number of matches and the number of layers with matches). Query templates perform a similar function when specified at the layer level, but in this case they give layer-specific summary information. Templates defined at the class level are used to present *individual* feature attributes (e.g., a state's location, population, and state flower) for elements of the result set. (Alternatively, instead of specifying multiple class-level templates, you can specify a layer-level template that's used to report detail results for all classes in the layer.) Finally, if you define a one-to-many join between the spatial data set and an external table, the result for each matching element of the join will be reported using the join-level template.

Joins are a new topic, and a more extensive treatment is provided later in this chapter. However, a brief description is warranted here since joins are discussed in the description of the application. Simply put, a join is a way of attaching the records in an external dBase file to attribute table records in a shapefile. The keyword `FROM` identifies the attribute name in the shapefile, and the keyword `TO` specifies the attribute name in the joined file. If a join has been defined for a layer, then for every selected feature, the joined file is scanned for records with `TO` item values that match the `FROM` item values in the shapefile.

Web/HEADER	Open HTML tags <HTML> <HEAD><TITLE>QUERY</TITLE></HEAD> <BODY> Global summary information
Layer/HEADER	Display layer summary & set up table headings <TABLE> <TR><TH>Item1</TH><TH>Item2</TH>...</TR>
Layer/TEMPLATE	Display attribute details for each match in all classes <TR><TD>[Item1]</TD><TD>[Item2]</TD>...</TR>
Class/TEMPLATE	Display attribute details for each match in a single class <TR><TD>[Item1]</TD><TD>[Item2]</TD>...</TR>
Join/TEMPLATE	Display details for each result of a 1-to-many join [var1][var2]
Layer/FOOTER	Close the table opened in Layer/HEADER </TABLE>
Web/FOOTER	Close the tags opened in Web/HEADER </BODY></HTML>
Web/EMPTY	Create complete page reporting no results <HTML> <HEAD><TITLE>EMPTY</TITLE></HEAD> <BODY> Query returned no results </BODY></HTML>

Figure 5-1. Query templates are HTML fragments that must be assembled to provide a complete web page.

Regardless of the level at which a template is used, MapServer employs the same substitution string method to populate query templates as it uses to populate the main application template. In the case of query templates, the set of substitution strings consists of attribute column names enclosed in square brackets and strings representing summary information generated by MapServer itself. For example, if the attribute names specified in the data set are CITY, STATE, and POPULATION, they're referenced in the layer- or class-level detail template by the substitution strings [CITY], [STATE], and [POPULATION].

Sometimes a query may return no results (i.e., no features or attributes match the query parameters). In such cases, MapServer displays an unformatted message in the browser similar to the following:

```
msQueryByPoint(): Search returned no results. No matching record(s) found.
```

Since this message isn't very aesthetically pleasing, MapServer provides a mechanism to substitute a formatted web page that informs the user of the null result, by way of the WEB-level keyword EMPTY. This page doesn't necessarily have to stand alone, either—it can be structured as a form that allows the user to reformulate the query, or it can even be structured as a frame, pop-up window, or tool tip that merely reports the result while displaying the original query page again.

Maintaining State in Query Mode

In Browse mode, various form variables and substitution strings can be used to maintain the state of the application (e.g., `zoomdir` and `[zoomdir_1_check]`). Matters become a little more complicated when using one of the query modes. The most important thing to note is the absence of substitution strings (like `[nquery_check]` or `[browse_check]`) to accompany the form variable mode. Although references to this set of mode-state substitution strings can still be found, the current release of MapServer doesn't support them. You as the developer must now keep track of mode from one invocation to the next. This is most easily accomplished with a simple JavaScript embedded in the application template. This script is included in the code distribution.

Querymaps

A *querymap* is a map that presents the results of a query by highlighting features that match the selection criteria. Querymaps aren't generated by default—they must be specified in the mapfile by the `QUERYMAP` object. If you don't specify a `QUERYMAP` object in the mapfile, queries can still be performed, but only tabular results will be available.

Since querymaps are referenced in the template in the same way as ordinary map images, they allow the use of pan and zoom controls. The only *visible* difference between querymaps and ordinary map images is the presence of highlighted features in the querymap image. From a developer's point of view, however, a mapping application that allows the user to navigate a querymap image is more complicated than one that doesn't. In order to use a querymap, query results must be saved in a queryfile, which is created during one session and used in subsequent sessions. Note that if MapServer can't find the correct queryfile, it displays an unformatted error message on the screen, like the following:

```
ms LoadQuery(): Unable to access file.(/var/www/htdocs/tmp/Fourth111935397822411.qy)
```

You can save query parameters by specifying `savequery` as the form variable `<input type=hidden name=savequery value=true>` while in a query mode. The parameters are saved in a file that has the same base name as the map image, but with a file extension of `qy`.

If you set the value of a hidden form variable named `queryfile` equal to the substitution string `[queryfile]` (i.e., `<input type=hidden name=queryfile value=[queryfile]>`), the location of the queryfile will be embedded in the web page passed to the user. The next time MapServer is invoked by the form, it will find the variable `queryfile` and load it before performing any other processes. This causes the query parameters to be used to generate a map image with the selected features highlighted. This map image is embedded in the template and forwarded back to the user, who can then use the usual pan and zoom navigation controls.

Map-Only Query Modes

Since the default behavior of query modes produces only tabular results, it's not surprising to learn that there's a set of query modes that produce maps but no tabular output—these are called *map-only* modes. For each query mode that produces tabular output is a corresponding map-only mode, and each of these modes is identified in the same manner: the map-only mode associated with the mode `nquery` is `nquerymap`; the map-only mode associated with the mode `itemquery` is `itemquerymap`, etc.

Map-only modes not only forego the presentation of tabular attribute data—in these modes, MapServer doesn't even scan a template and return the page to the user with the map embedded in it. It returns only the image. This is useful, for example, when an image must be loaded by a URL in an image tag. For example, if the image tag in the following code snippet is embedded in a query template, MapServer will replace the substitution strings with appropriate values to create a valid URL that will cause the browser to load a map image.

```
<image src="[host]/cgi-bin/[program]?  
mode=indexquerymap&  
map=[map]&  
shapeindex=[shpidx]&  
mapext=shape&  
shpext=[shpext_esc]">
```

This is an interesting use for map-only modes, but I won't go into details here. Modes are described in greater detail in the next section, and the substitution strings and form variables will be described in Chapter 11, "MapServer Reference."

You should now have a rough idea of what MapServer queries can do. All the major pieces have been described and put into context. However, effectively using MapServer's query capability requires a clear understanding of the modes, substitution strings, and CGI variables specific to queries.

Since queries are complex and often misunderstood, I'm going to change the usual presentation sequence at this point. Instead of a more extensive discussion of query modes, form variables, and substitution strings, I'll proceed directly to the installation of this chapter's application. Then I'll step through some examples to give you an understanding of each query mode.

You can retrieve all the code used in this book from the Downloads area of the Apress website (www.apress.com). The spatial data set is also available from the same location. The files for this application can be found in the archive `fourth.zip`.

Query Examples

Using MapServer queries requires more than an understanding of mapfile and template syntax. Before reviewing the code, it will be useful to spend some time working through the various query modes to gain an understanding of what each does. This section will provide you with a structured approach that uses a lot of examples and illustrations. Each example will be introduced with a brief note that describes the query parameters and the results. A number of screenshots will be presented to show you what you should be seeing. The examples will follow the sequence of modes in the drop-down box on the main page of the fourth application. The main features of each query mode are summarized in Table 5-1.

Table 5-1. *Query Modes*

Mode	Type	Returns
QUERY	Spatial	Closest feature within tolerance
NQUERY	Spatial	All features within tolerance
ITEMQUERY	Attribute	First matching attribute
ITEMNQUERY	Attribute	All matching attributes
FEATUREQUERY	Spatial	First matching polygon in <code>slayer</code> plus all features within tolerance of match
FEATURENQUERY	Spatial	All matching polygons in <code>slayer</code> plus all features within tolerance of matches
ITEMFEATUREQUERY	Attribute/spatial	First attribute match in <code>slayer</code> plus all features within tolerance of match
ITEMFEATURENQUERY	Attribute/spatial	All attribute matches in <code>slayer</code> plus all features within tolerance of matches
INDEXQUERY	Index	The feature with the specified shape index

The spatial data set in the fourth application consists of a polygon shapefile describing countries of the world and a point shapefile that contains city information. In order to demonstrate the use of joins, another dBase file is used—it contains population information relating to some of the cities. The map has two searchable layers: the first for countries and the second for cities.

QUERY Mode

Type the URL of the initialization page (http://localhost/fourth_i.html) into the address bar of your browser and press Enter. The initialization screen shown in Figure 5-2 will be displayed. Click the Click Me button to proceed to the main query page, shown in Figure 5-3.

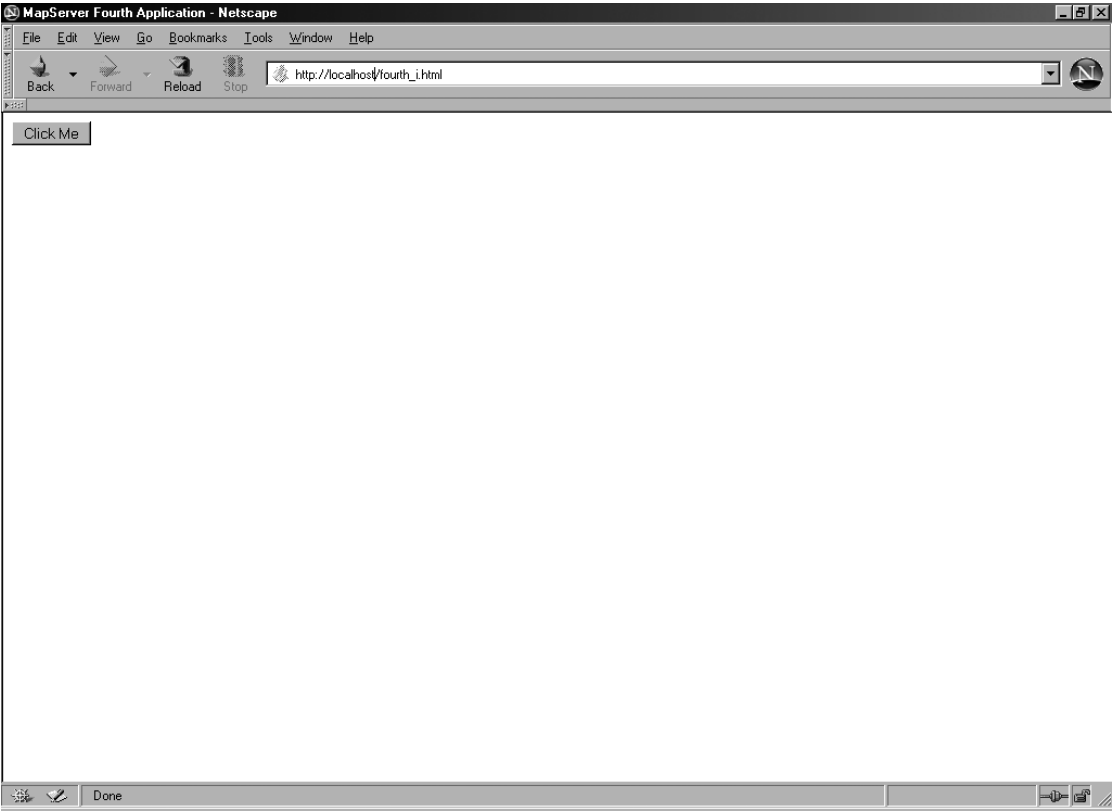


Figure 5-2. *The query initialization page*

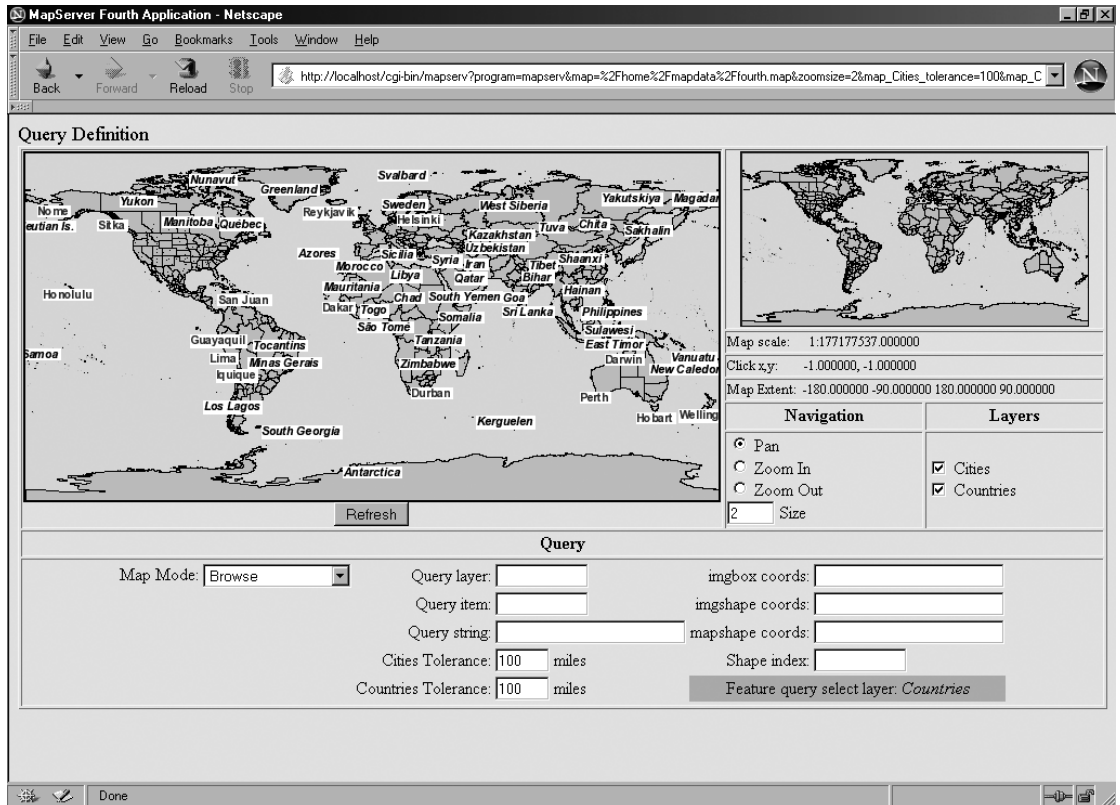


Figure 5-3. Initial display of the query definition page

Select Query from the Map Mode drop-down box. Notice that the TOLERANCE for both the Cities layer and the Countries layer is already set to 100 miles. The values in these two fields are used to set the TOLERANCE for their respective layers. But in this example, the only TOLERANCE that matters is the one associated with the Cities layer. Make sure that this value is set to 100 miles.

Next, click somewhere within the state of Texas. (Texas is used because it's large enough to see at the initial map scale of 1:177,000,000, and also has several cities.) If your mouse click is within 100 miles of a city, you'll be presented with a page that looks like Figure 5-4 (this will vary depending on which city is closest to the mouse click). This page presents the map- and layer-level query information: search extent, click point, and number of query results. Following this is a single line containing information about the city: name, state, latitude and longitude, and perhaps population (which is taken from the joined file).

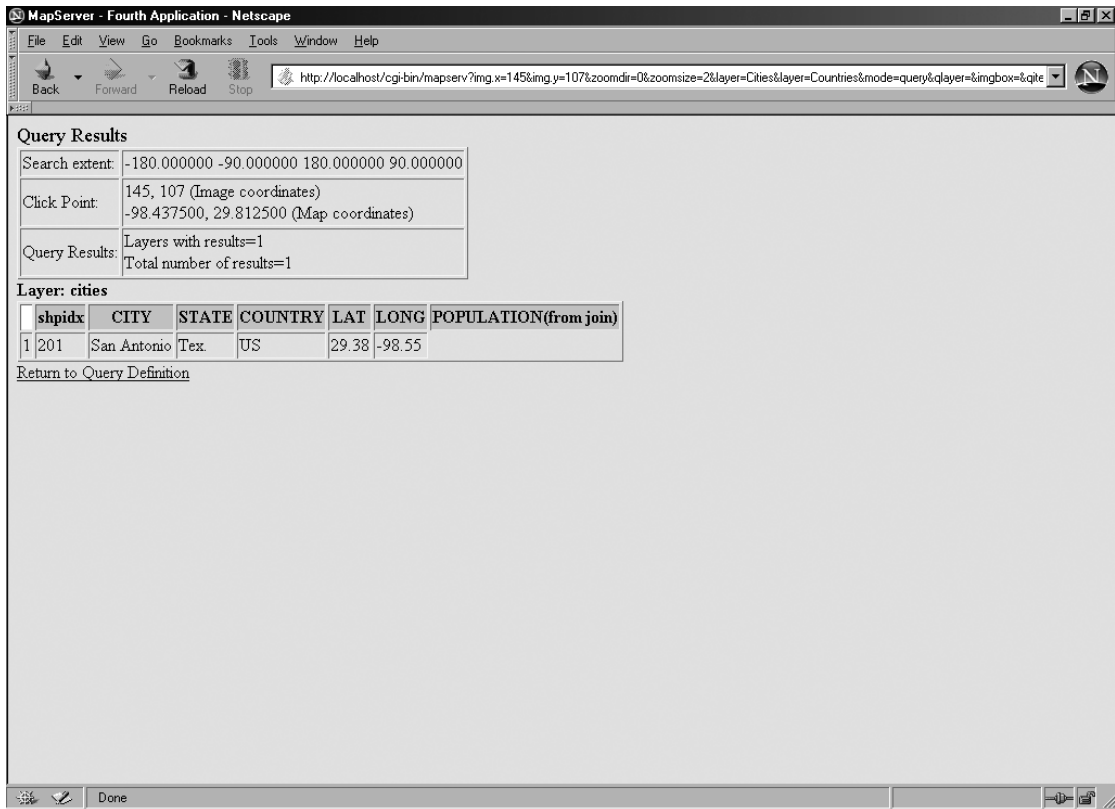


Figure 5-4. Result page for QUERY mode displaying a single city

If the mouse click isn't within 100 miles of a city, a page similar to Figure 5-5 will be displayed. Summary results are presented first, then a querymap image, and finally a single line showing the attributes of the state of Texas. Notice that the querymap has rendered the polygon representing Texas in yellow.

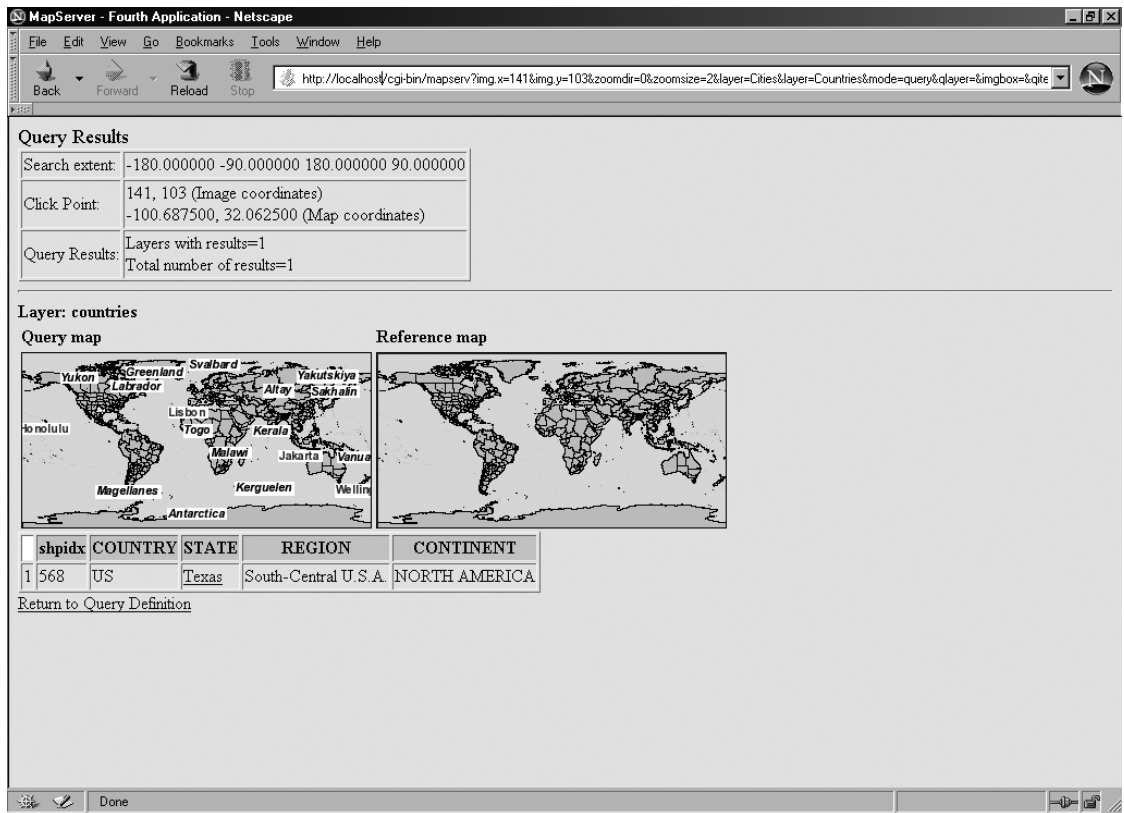


Figure 5-5. Result page for QUERY mode showing a single state

This mode returns only a single result—it will return the city result if a match is found because the Cities layer, having been rendered after the Countries layer, is searched first. If the sequence in which the layers were specified in the mapfile were reversed, then the Countries layer would be searched first and would return a match.

In all query modes in this application, the state name is a link that uses the INDEXQUERYMAP mode to produce a map of the feature. Click on Texas in the STATE column and MapServer should display a map resembling Figure 5-6.

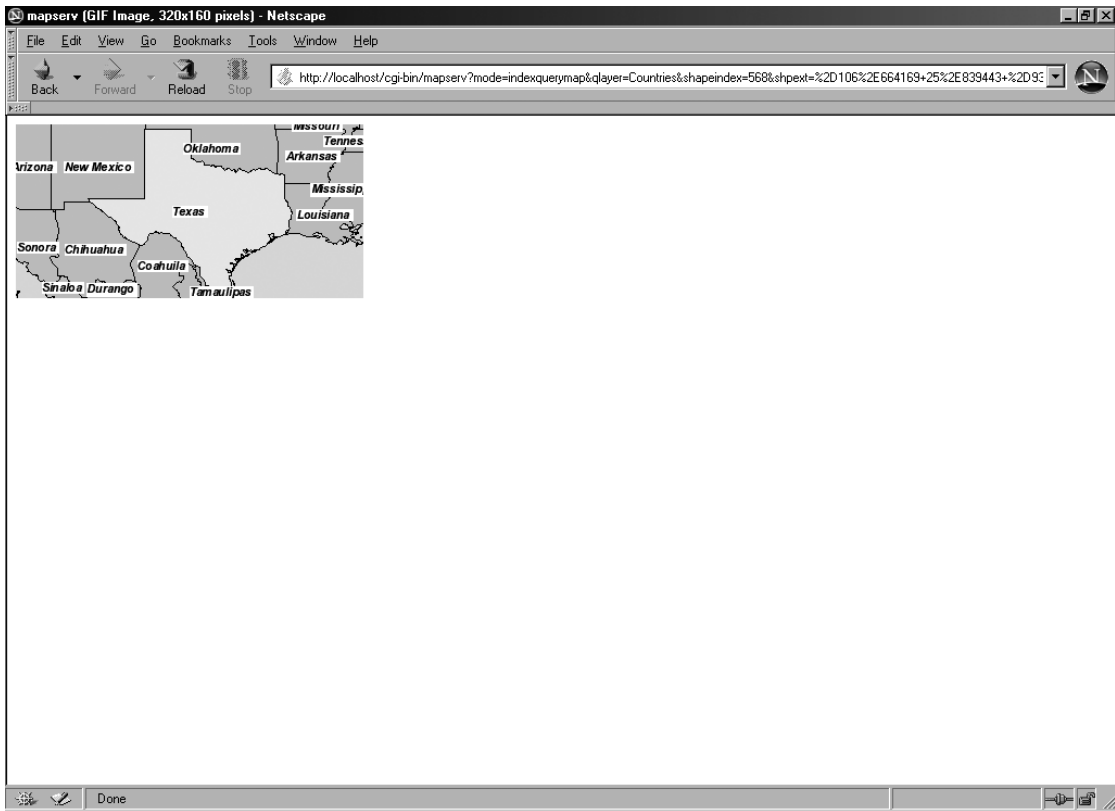


Figure 5-6. INDEXQUERYMAP mode produces only the map image.

NQUERY Mode

From the main query page, select NQUERY mode. Set the TOLERANCE to 300 for the Cities layer and 0 for the Countries layer. Click on Texas again. This time, MapServer should return all matches (see Figure 5-7). Several cities are returned (those within 300 miles of the mouse click), but only one state is returned (Texas, the only state that matches the query). Since this is a spatial query and Texas is a polygon, a click anywhere inside the state will produce a match, but a point outside the state boundary won't match because TOLERANCE is set to 0.

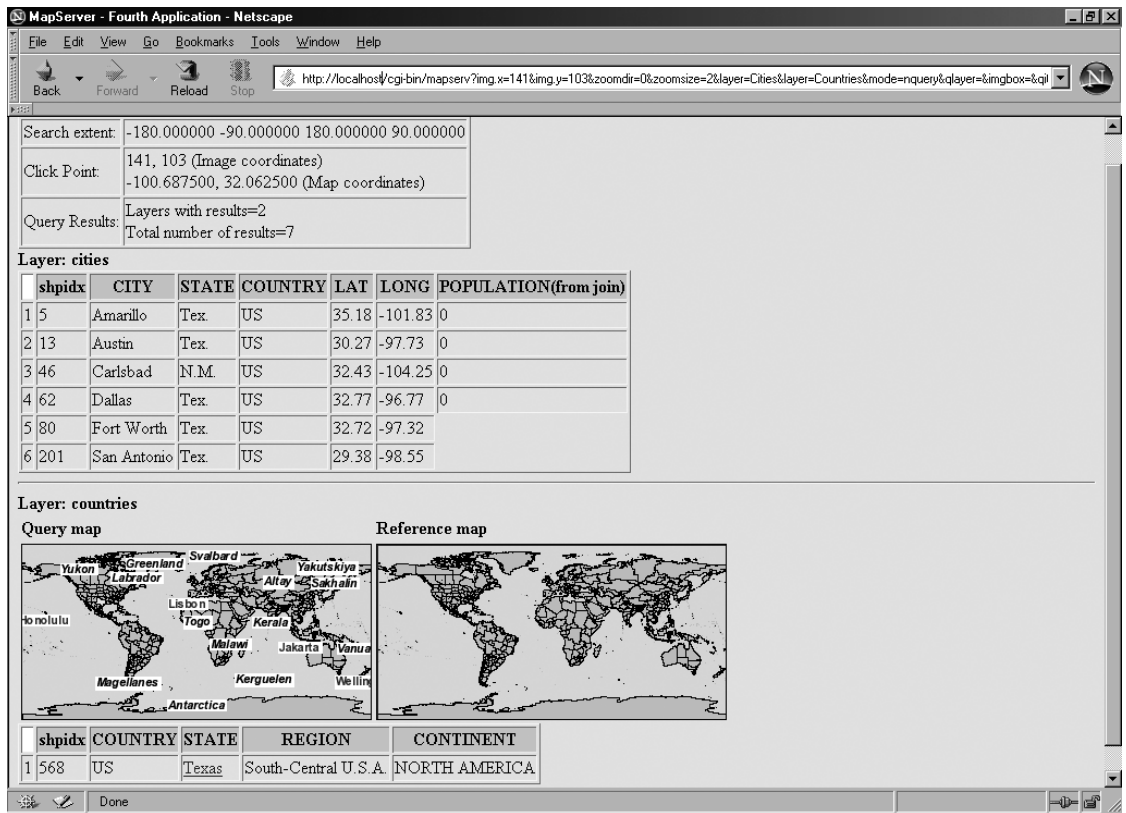


Figure 5-7. *NQUERY mode produces multiple results and displays both states and cities.*

Change the Countries-layer TOLERANCE to 200 miles and repeat the query by clicking on Texas. If you clicked the same spot on the map, then the same cities should be returned as before, but this time you should find (depending on where the mouse click was made) that Oklahoma, New Mexico, and even a Mexican state will be returned (see Figure 5-8). The larger TOLERANCE value specifies a larger search area—the Countries-layer search is no longer cut off at the border of Texas; it now extends 200 miles beyond the border to include other states.

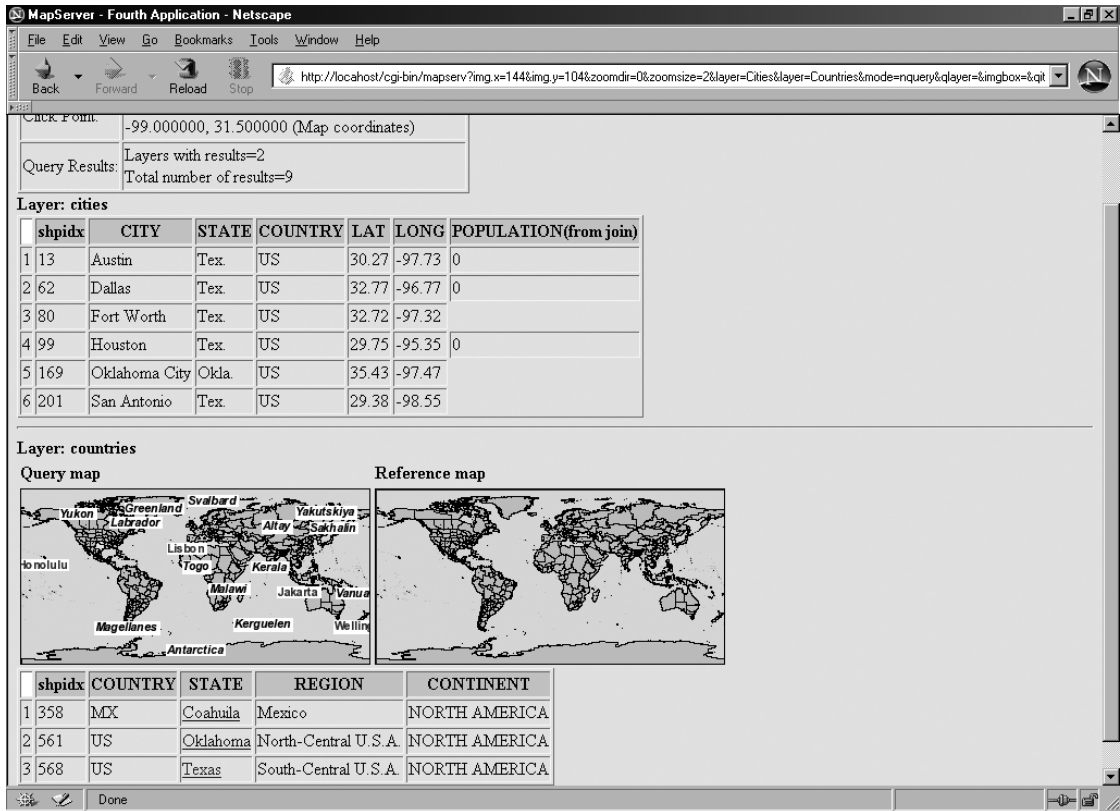


Figure 5-8. In NQUERY mode, a higher layer TOLERANCE returns more matches.

The critical change here is the change in TOLERANCE. Both QUERY and NQUERY are point queries in the current context. That is, a circle with radius equal to the TOLERANCE is drawn about the click point, and any feature falling within that circle will match. However, some queries don't use the TOLERANCE in this way—another use of TOLERANCE will be reviewed in more detail later, in the discussion of feature queries.

ITEMQUERY Mode

Select ITEMQUERY mode from the main query page. Set TOLERANCE to 0 for both layers. This mode selects features based on matching attribute values—as such, no distance measure is used. Set Query layer to Cities and Query item to COUNTRY. (Don't confuse the layer names Cities and

Countries with the attribute names CITY and COUNTRY.) For Query string, use the regular expression /US/. Click Refresh (or click anywhere on the map), and a page similar to Figure 5-9 will be returned. The matching criterion is that COUNTRY is equal to /US/, so the first match will be the first city in the attribute table with COUNTRY equal to US. This turns out to be Albany, New York.

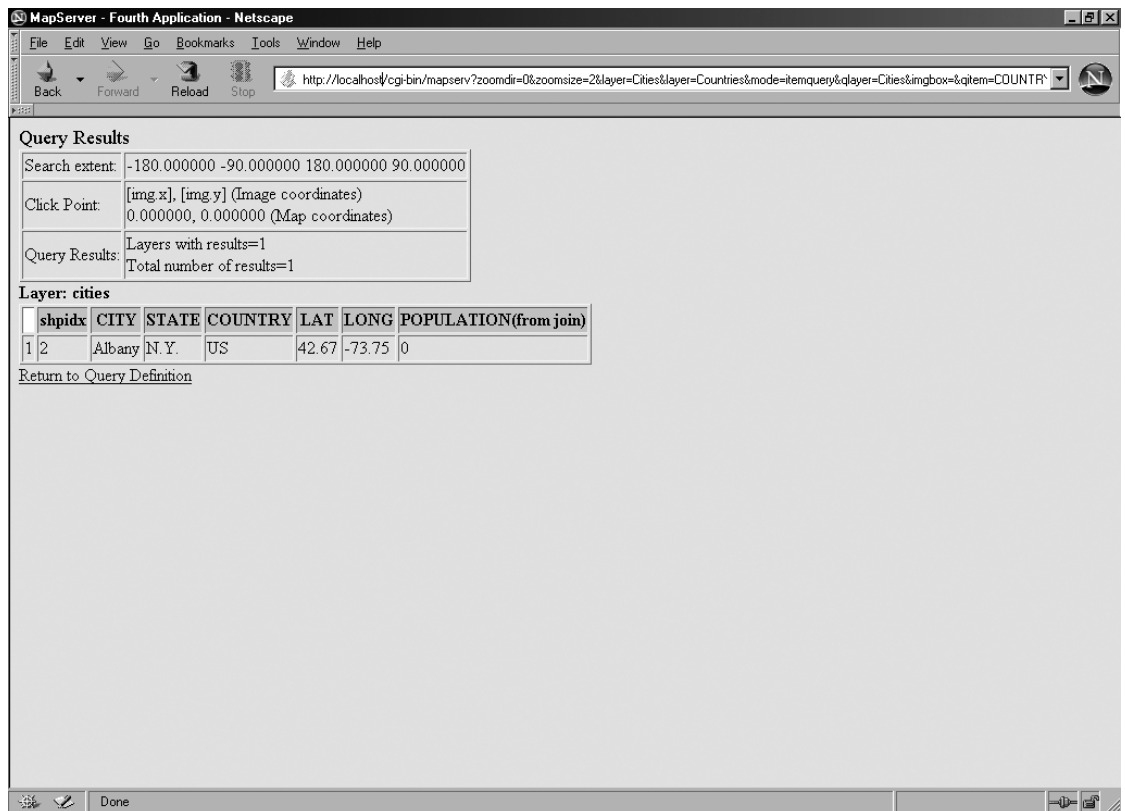


Figure 5-9. ITEMQUERY MODE selects a feature based on an attribute value and returns the first match feature from the Cities layer.

ITEMNQUERY Mode

Next, leave the other query parameters the same, select ITEMNQUERY mode, and click Refresh or the map. Figure 5-10 shows a portion of the result set. If you scroll through the list of cities, you'll notice that population results are returned for both Washington DC and San Jose, California.

83	185	Portland	Ore.	US	45.52	-122.68	0
84	187	Providence	R.I.	US	41.83	-71.40	0
85	189	Raleigh	N.C.	US	35.77	-78.65	0
86	191	Reno	Nev.	US	39.50	-119.82	0
87	193	Richfield	Utah	US	38.77	-112.08	0
88	194	Richmond	Va.	US	37.55	-77.48	0
89	196	Roanoke	Va.	US	37.28	-79.95	0
90	198	Sacramento	Calif.	US	38.58	-121.50	0
91	199	Salt Lake City	Utah	US	40.77	-111.90	
92	201	San Antonio	Tex.	US	29.38	-98.55	
93	202	San Diego	Calif.	US	32.70	-117.17	
94	203	San Francisco	Calif.	US	37.78	-122.43	
95	204	San Jose	Calif.	US	37.33	-121.88	315909
96	205	San Juan	P.R.	US	18.50	-66.17	
97	206	Santa Fe	N.M.	US	35.68	-105.95	
98	209	Savannah	Ga.	US	32.08	-81.08	0
99	210	Seattle	Wash.	US	47.62	-122.33	0
100	212	Shreveport	La.	US	32.47	-93.70	0
101	214	Sioux Falls	S.D.	US	43.55	-96.73	
102	215	Sitka	Alaska	US	57.17	-135.25	0
103	217	Spokane	Wash.	US	47.67	-117.43	0
104	218	Springfield	Ill.	US	39.80	-89.63	0
105	219	Springfield	Mass.	US	42.10	-72.57	0

Figure 5-10. *ITEMNQUERY* selects features based on attribute values and returns all matches from the *Cities* layer.

The San Jose population number, which is derived from a (faulty) join, is completely bogus—this points out a limitation of the join capabilities of MapServer. Recall that a join specifies a FROM and a TO item. If the value of the TO item in a row of a joined table has the same values as the FROM item in an attribute table, the JOIN row will be appended (joined) to the row in the attribute table. This means that if multiple records in the attribute table have the same FROM values, the same TO record will be appended to each. It's not possible to use multiple FROM items to link to an external table based on a match of multiple TO items. In the present case, San Jose, California (which doesn't have population data stored in the table) was linked to San Jose, Costa Rica (which does).

Change *qlayer* from *Cities* to *Countries* and click *Refresh*. This time, MapServer searches the *Countries* layer, returns all the states in the United States, and displays a querymap with the states highlighted in yellow, as shown in Figure 5-11.

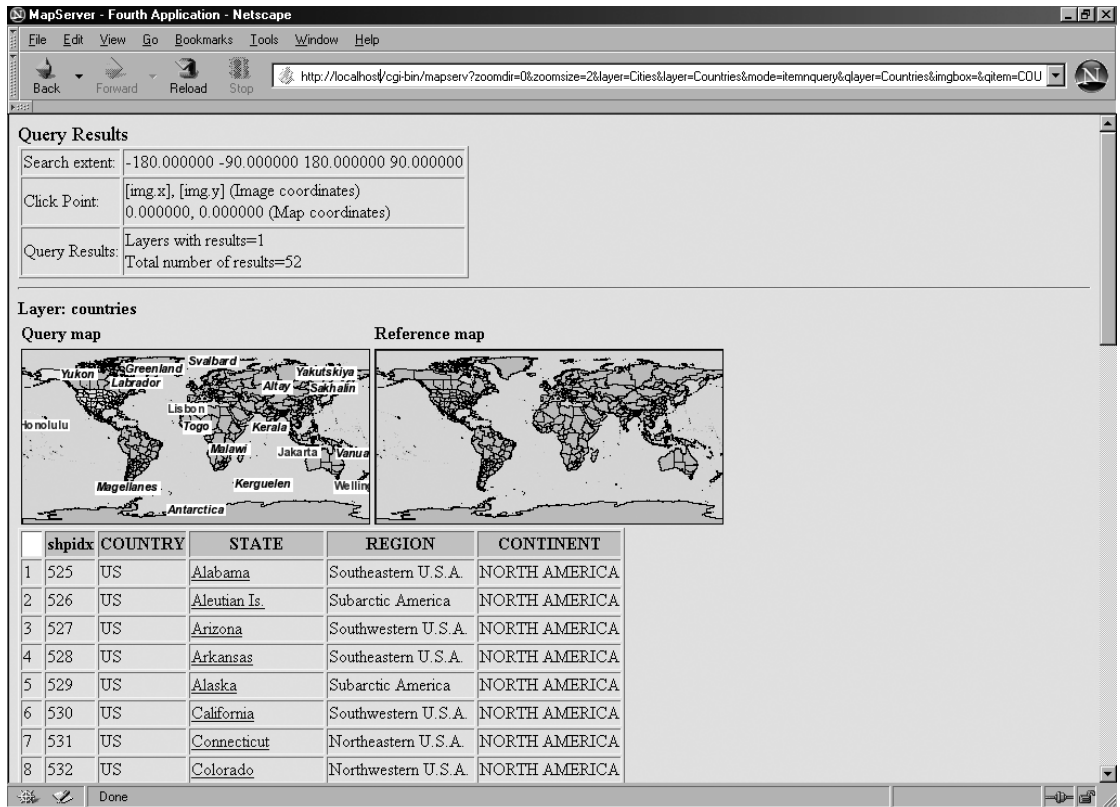


Figure 5-11. ITEMQUERY results when searching only the Countries layer

FEATUREQUERY Mode

Select FEATUREQUERY mode and set the Cities-layer TOLERANCE to 0 and the Countries-layer TOLERANCE to 100 miles. Clear Query layer, Query item, and Query string. This mode performs a spatial search within a single selected polygon. Regardless of the magnitude of TOLERANCE for the selection layer, only the first match will be returned for further processing. Click Texas, and MapServer will return the following results: attributes of cities that it found in Texas and the attributes of the state itself. The selection polygon is highlighted in yellow on the querymap. This is shown in Figure 5-12. Because the TOLERANCE for the Cities layer is set to 0, only those cities inside the Texas boundary are returned. If it had been set to some non-zero value, then cities outside of Texas (but within the TOLERANCE of the boundary) would also have been returned.

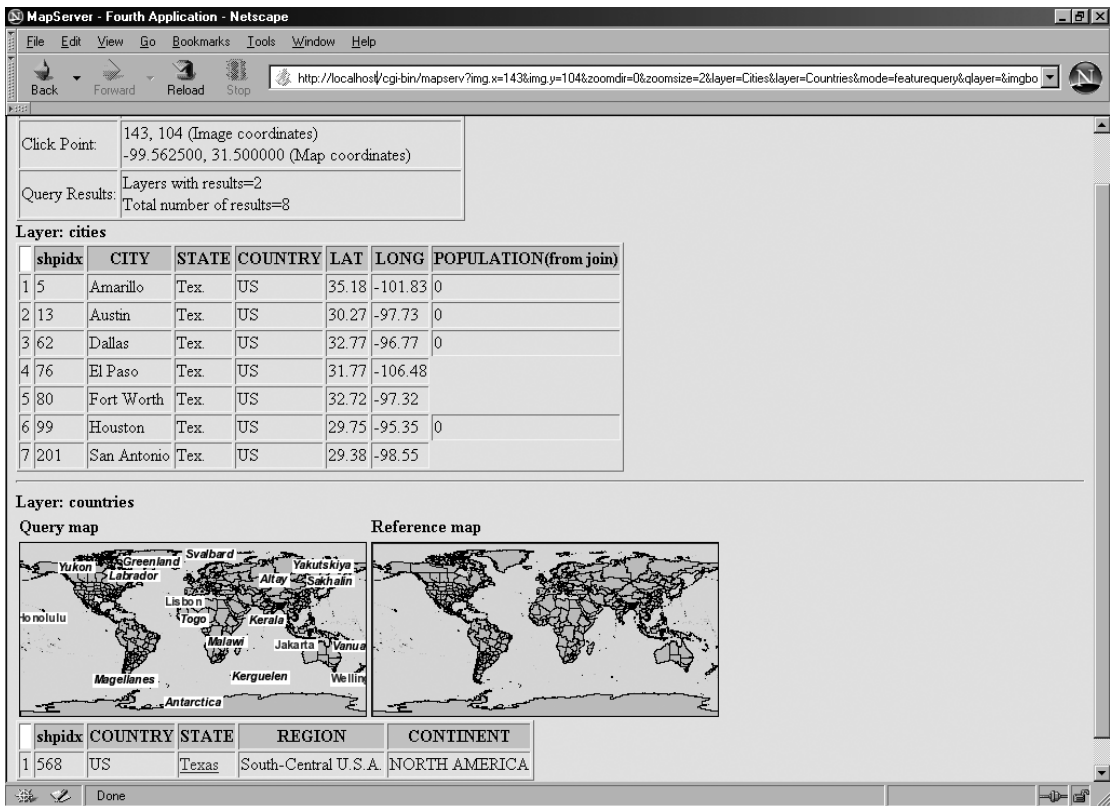


Figure 5-12. *FEATUREQUERY* mode returns a single polygon feature from the selection layer, as well as all features from other layers that are close to or inside the selected polygon.

FEATUREQUERY Mode

Select *FEATUREQUERY* mode, change the *Countries* layer *TOLERANCE* to 200 miles, and leave the other parameters blank. Click Texas again. This time, MapServer returns all of the matches from the selection layer. The result set will depend on where the mouse was clicked, but any polygon in the selection layer with an extent that overlaps a circle of radius equal to the *TOLERANCE* will match, and so be used for querying the *Cities* layer. Clicking in northern Texas will return (in addition to Texas itself) New Mexico and Oklahoma, but clicking in southern Texas will return the Mexican states of Coahuila, Nuevo León, and Tamaulipas. The cities returned will be those falling inside the boundaries of the selected states, as shown in Figure 5-13.

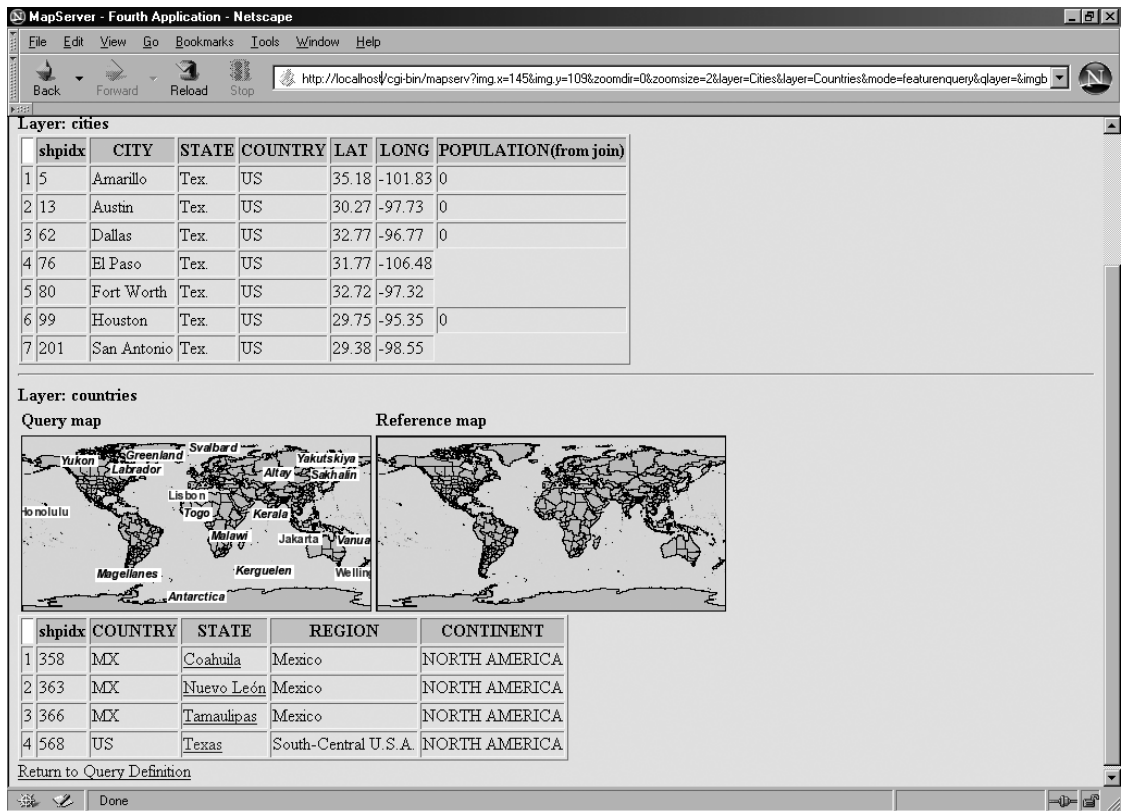


Figure 5-13. *FEATUREQUERY* mode returns all matching polygon features in the selection layer and all features from other layers that are close to or inside the selected polygon.

The interaction of tolerances in this mode is complicated. Set the TOLERANCE to 300 miles for Cities and 0 for Countries. This time, clicking on Texas will return cities within 300 miles of the Texas border and within the state of Texas itself (see Figure 5-14). So, when the selection-layer TOLERANCE is set to 0, the *FEATUREQUERY* mode is functionally equivalent to the *FEATUREQUERY* mode. Table 5-2 shows how the two modes differ.

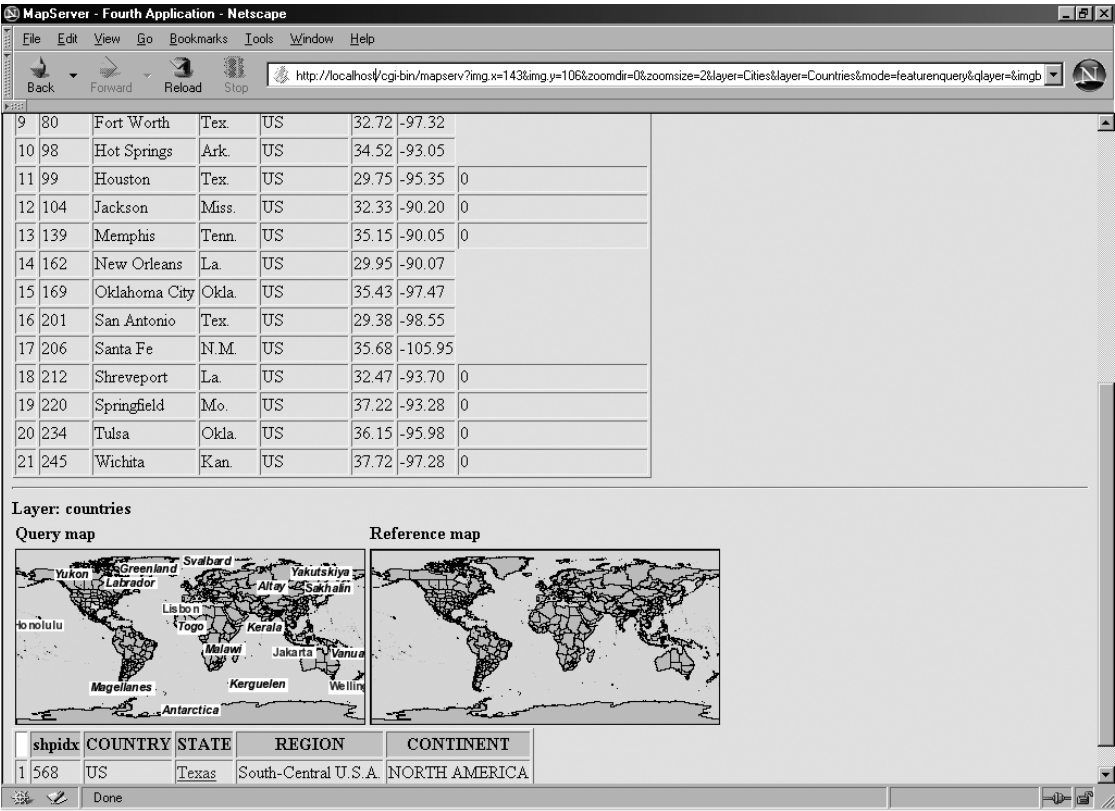


Figure 5-14. *FEATUREQUERY mode with selection layer tolerance set to zero functions the same way as FEATUREQUERY mode*

Table 5-2. *Comparison of Feature Query Modes*

Mode	Selection-Layer TOLERANCE	Search-Layer TOLERANCE	Returns
FEATUREQUERY	0–infinity	0	First matching polygon in the select layer and all features (from other queriable layers) found within the selected polygon
	0–infinity	X	First matching polygon in the select layer, all features found within the selected polygon, and all features within distance X of selected polygon
FEATUREQUERY	0	0–infinity	Same as FEATUREQUERY mode
	Y	0	All polygons in the select layer within distance Y of the click point, and all features within selected polygons
	Y	X	All polygons in the select layer within distance Y of the click point, all features within selected polygons, and all features within distance X of selected polygons

ITEMFEATUREQUERY Mode

Select ITEMFEATUREQUERY and set Query item to STATE and Query string to Texas. Set the TOLERANCE for the Cities layer to 0 and the Countries layer to 200 miles, and click Refresh. MapServer performs an attribute query and searches for a STATE equal to Texas. The 200 mile TOLERANCE is irrelevant since no spatial search is being done on the slayer. Then, because the TOLERANCE of the Cities layer is set to 0, only features within the selected state will be returned. If the TOLERANCE had been set to some non-zero value, then features within that distance of the Texas border would have also been returned (see Figure 5-15).

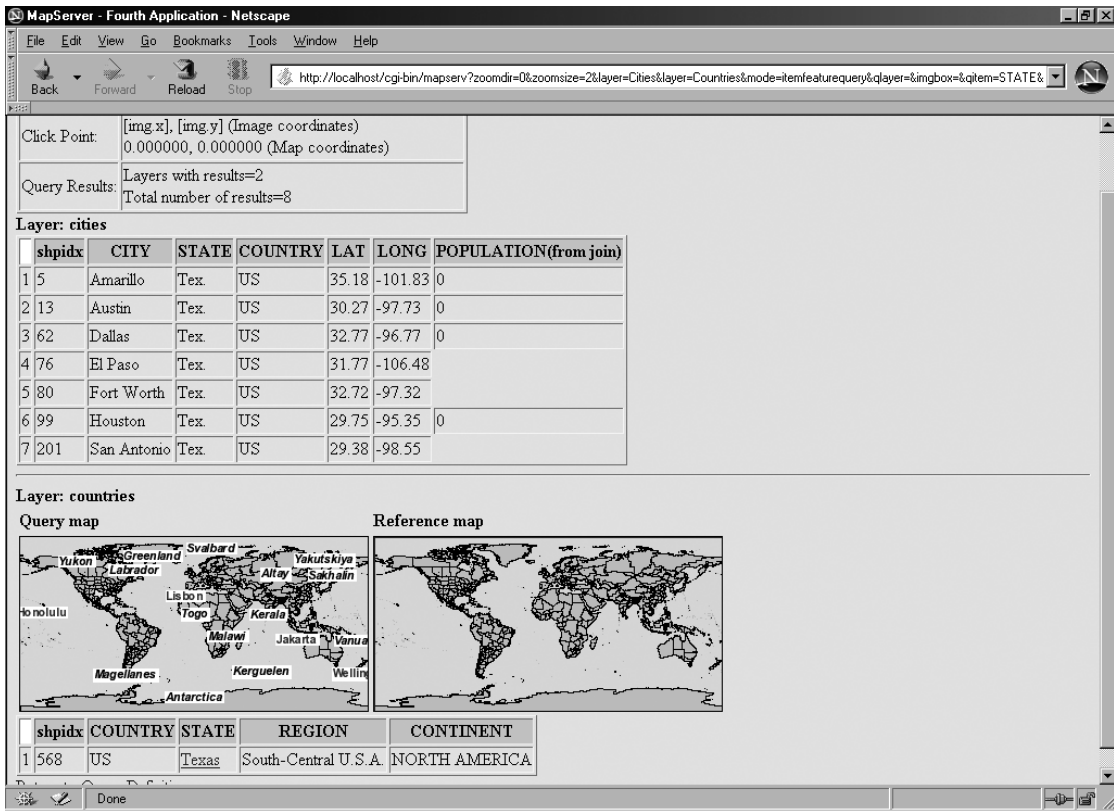


Figure 5-15. *ITEMFEATUREQUERY* mode selects a single polygon feature from the selection layer based on the attribute value and returns all matching features found within or close to the selected polygon.

ITEMFEATUENQUERY Mode

Although this is the most complex mode, you should know what to expect by now. Set Query item to COUNTRY and Query string to US, leave the TOLERANCE values unchanged, and click Refresh. MapServer should return something like the fragment shown in Figure 5-16. The layer upon which the attribute search is performed is specified by the value of slayer. MapServer will return all features in that layer with the attribute specified by qitem (the query item) equal to the string specified by qstring (the query string). The result set consists of all Countries-layer features for which the COUNTRY attribute is equal to US, and all the cities within those features.

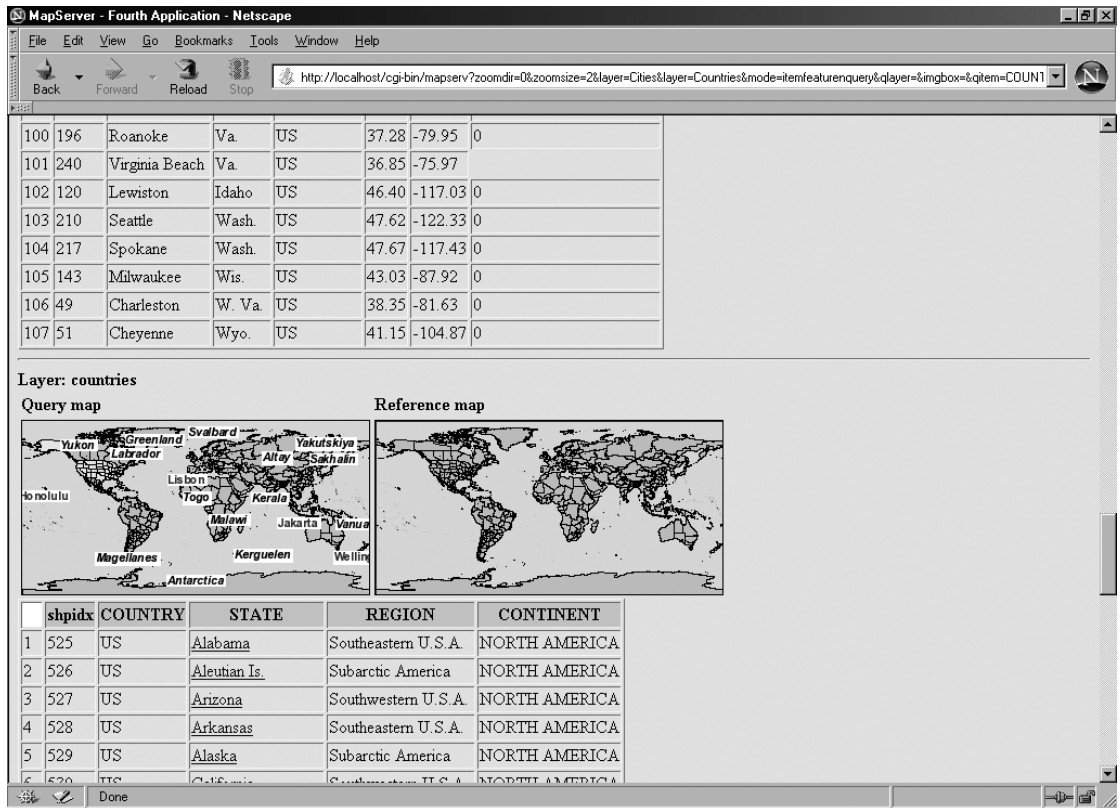


Figure 5-16. *ITEMFEATUREQUERY* mode selects all matching polygon features from the selection layer based on the attribute value and returns all matching features found within or close to the selected polygons.

Next, try increasing the *TOLERANCE* of the *Cities* layer to 200 miles and repeating the query. As Figure 5-17 shows, the list of selected states should be unchanged. However, cities in Canada and Mexico, which were absent from the previous result set, are now returned, since these cities all lie within 200 miles of a US state boundary.

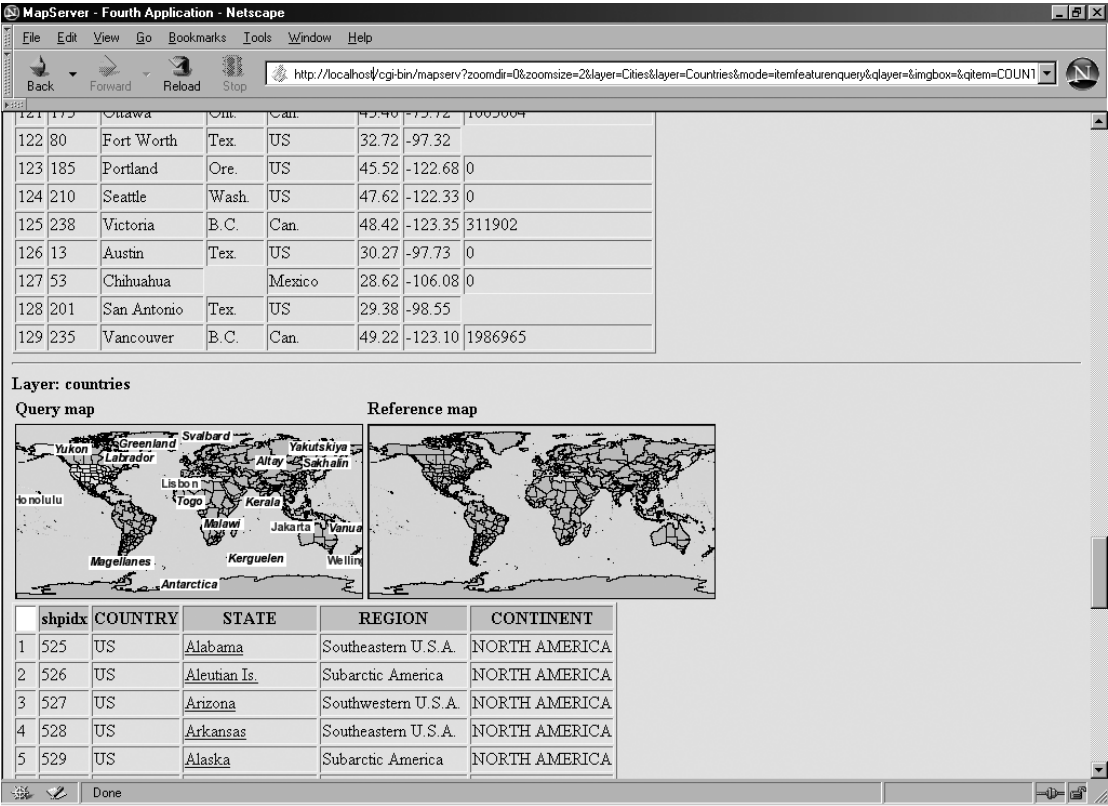


Figure 5-17. *ITEMFEATUREQUERY* mode results with larger *TOLERANCE* for the *Cities* layer

INDEXQUERY Mode

An index query returns a feature identified by its shape index. Set Query layer to Countries and Shape index to 568. Click Refresh, and MapServer will display the result shown in Figure 5-18.

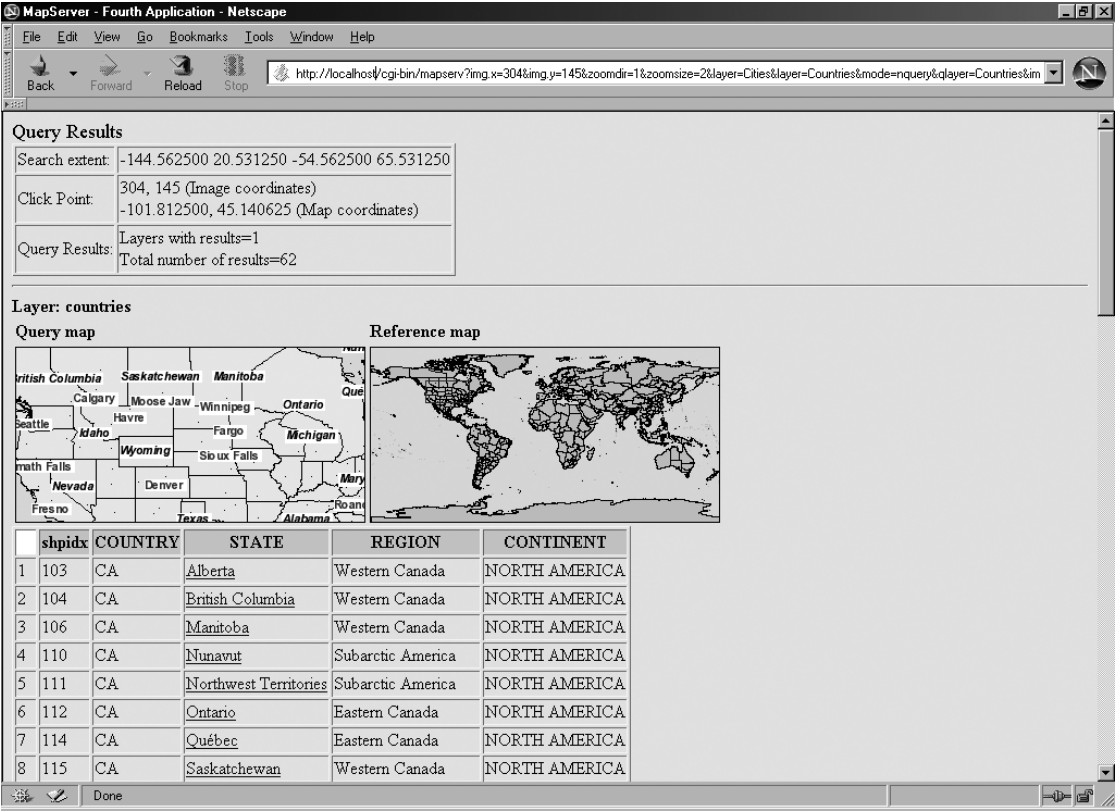


Figure 5-19. An NQUERY mode spatial search using a polygon search region instead of a point

Without changing anything else, add the following coordinates to the end of the list you've already entered:

-110 45 -110 35 -80 35 -80 45 -110 45

MapServer traverses this additional figure in a counterclockwise direction, so it forms a hole in the middle of the previous shape. Click the map and MapServer will return a map similar to Figure 5-20.

The next section describes each of the query modes and explains how they work. Following that, querymaps, joins, substitution strings, and CGI variables will be described.

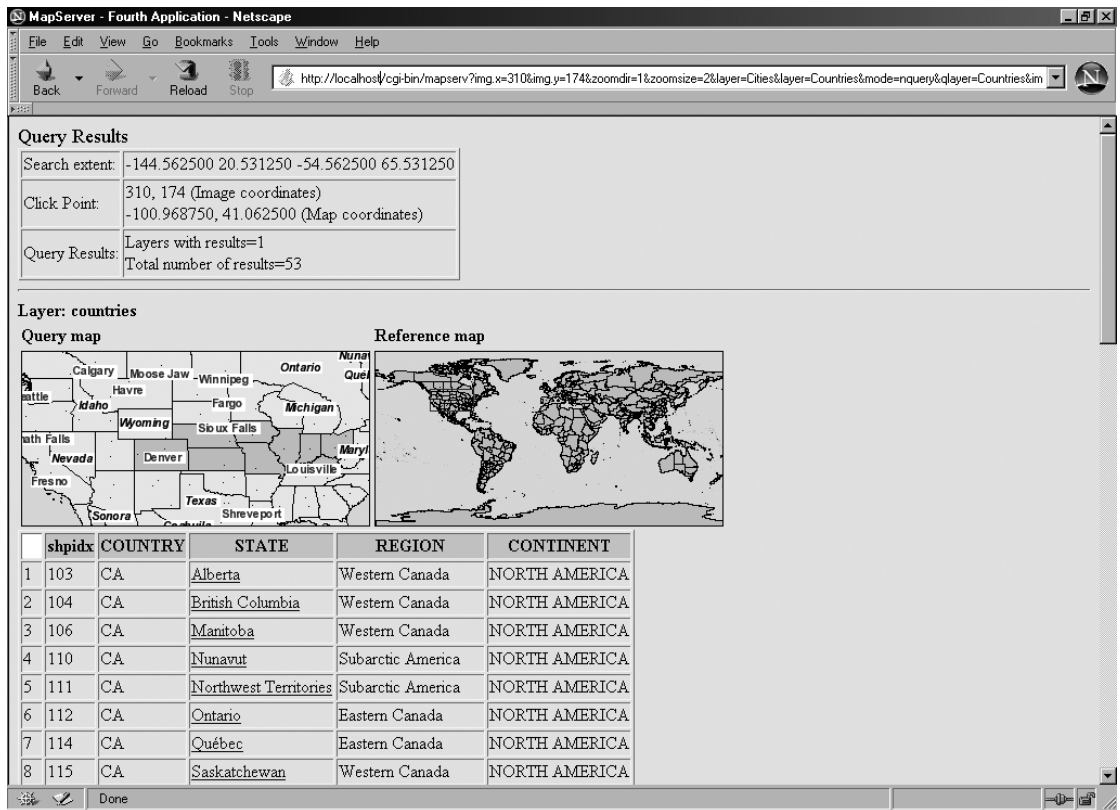


Figure 5-20. An NQUERY mode spatial search using a doughnut-shaped search region.

Query Modes

MapServer keeps track of the current mode with the CGI variable `mode`, but has no corresponding substitution string (`[mode]`) to maintain state. There are more than two dozen modes. Some produce maps, scale bars, and legends, while others change the mode to Browse and set the zoom direction. This chapter however, is devoted to MapServer's query modes. There are 18 of them: nine that can produce both maps and tabular results, and another nine map-only modes that present only the spatial results of queries. The discussion that follows omits the map-only modes since they were described in the previous section.

Tip Don't panic. Although the plethora of modes can be confusing at first, the application that you'll build in this chapter will allow you to experiment with each query mode and see what it does. Once you have the basics, seeing a query in action is a good way to cement your understanding.

QUERY Mode

QUERY mode performs the simplest spatial query: a point query. The image coordinates (in pixels) of a mouse click on the map image are returned to MapServer as the values of the CGI form variables `img.x` and `img.y`. (This means that the HTML tag that contains the map image `<input type="image" name="img" src"...">` must be named `img`). These coordinates are used to search one layer after another until MapServer finds the feature nearest to the click (within a specified `TOLERANCE`) and returns a *single* result: the attributes of that feature. If no matching feature is found within the distance specified by the layer-level keyword `TOLERANCE`, then MapServer looks for the URL specified by the keyword `EMPTY` and sends that page back to the browser. If there's no `EMPTY` page, then MapServer will simply report the null result of the query with an unformatted message to the browser.

Note The maximum distance between the map click and a feature that MapServer will accept as a match in a spatial query is specified by the mapfile layer-level keyword `TOLERANCE`—its units are determined by the keyword `TOLERANCEUNITS`. The default `TOLERANCE` is 3 pixels, but `TOLERANCEUNITS` can take any one of the following values: pixels, feet, inches, kilometers, meters, miles, or dd (decimal degrees).

NQUERY Mode

NQUERY works the same way QUERY does, except that it can perform area searches in addition to point queries, which adds significant power to MapServer's query capabilities. You can specify rectangular search extents, or you can even specify arbitrary shapes and define multiple connected extents—that is, extents with holes. NQUERY mode uses the same method for point queries as QUERY mode: form variables `img.x` and `img.y` are returned and MapServer returns matches within the `TOLERANCE` distance of the mouse click.

Searching an extent is only slightly more complicated because the search region returned to MapServer possesses more coordinates. To search a rectangular extent, the CGI form variable `imgbox` is used to return the space-delimited image coordinates (i.e., pixels) of the top-left and bottom-right corners of the rectangle (this is the reverse of a geographical extent because the pixel count in an image increases from top to bottom). An arbitrary search region is specified in image coordinates by the form variables `imgshape` or, in world coordinates, `mapshape`. For area searches, all matches within the `TOLERANCE` distance of the boundary of the search extent are returned.

As discussed in the Appendix, polygons in shapefiles are described by a sequence of vertices, with the first and last vertices being equal. Further, the inside of a polygon is defined to be the area on the right as the vertices are traversed in sequence. MapServer expects the list of coordinate pairs contained in the form variables `imgshape` or `mapshape` to observe this standard.

Defining two non-intersecting regions is as simple as specifying their vertex coordinates in sequence. In order to create a hole in the interior of a polygon, specify the list of vertex coordinates of the interior shape in reverse order.

Making MapServer aware of the coordinates of the search region can be done in a couple of ways. One method would be to use a Java application to capture drag box coordinates or mouse-click sequences that define the vertices of polygons. However, since Java programming

is out of the scope of this book, I'll show you how to use a simpler method: assigning strings that represent coordinate values to the variables `imgshape` and `mapshape`.

ITEMQUERY Mode

This mode performs the simplest attribute search—looking for matching text in attribute tables and returning the *first* match. Specifying a `TOLERANCE` here doesn't make sense—`TOLERANCE` values are ignored during attribute searches.

The search expression is specified with the CGI variable `qstring`. The search expression assigned to `qstring` will depend on the underlying data source. The syntax is the same as the layer-level keyword `FILTER`. For shapefiles or OGR data sources, this is the same as the MapServer expression (string comparison, regular expression, or logical expression) used by the class-level keyword `EXPRESSION`. However, in the case of SDE, OracleSpatial, or PostGIS, the search expression is a native SQL `WHERE` clause.

It's possible to specify an attribute name to limit the search to a single column by means of the CGI variable `qitem`. If no `qitem` is specified, then all attributes are searched.

If no query layer is specified (by means of the CGI variable `qlayer`), all layers are searched. The HTML substitution strings `[qstring]`, `[qitem]`, and `[qlayer]` correspond to the CGI variables `qstring`, `qitem`, and `qlayer`.

As an example, consider a mapfile containing two layers—a polygon layer named `Countries` and a point layer named `Cities`. The user might search the `Cities` layer for the first record, in which the `STATE` equals Texas. This can be accomplished by setting `qstring` to `/Tex./`, `qitem` to `STATE`, and `qlayer` to `Cities`. (These particular values will work with the data set supplied with the application code.) MapServer will return the first matching feature, which in this case happens to be the record associated with the city of Amarillo.

ITEMNQUERY Mode

This is the same as `ITEMQUERY`, except in this case all matches are returned. If you repeat the last example but select `ITEMNQUERY` instead, MapServer will return all the cities in the database for which `STATE` equals `Tex`.

FEATUREQUERY Mode

`FEATUREQUERY` mode performs a spatial query that uses a feature from one layer to query another layer. Only the first matching feature from the selection layer is selected for use in the subsequent spatial search. However, all the features found within that *selected* feature will be returned. The selection layer is defined by the CGI variable `slayer` and must be of polygon type. The variable `slayer` has a corresponding substitution string, `[slayer]`, used to maintain state.

Since `FEATUREQUERY` is a spatial query, the `TOLERANCE` value assigned to the searchable layers is important. If the `TOLERANCE` of a *searchable* layer is 0, then only those features contained within the selected polygon will match. If the `TOLERANCE` is set to some non-zero value, then features outside of the selected polygon but within the `TOLERANCE` distance of the boundary of the polygon will also match. The `TOLERANCE` value of the `slayer` isn't relevant in this mode since only the first polygon (i.e., the polygon in which the mouse click occurred) is used.

Continuing the previous example, if the user selects `FEATUREQUERY` and clicks the polygon representing the state of Texas, a query is performed that searches the `Cities` layer, returning all the cities in Texas that are contained in the database. If `TOLERANCE` in the `Cities` layer is set

to 0, only cities within the boundary of Texas will be returned. If `TOLERANCE` is set to 100 miles, then cities within 100 miles of the Texas border will be returned as well. But remember, if you click inside the boundary of Texas, only Texas is returned from the selection layer. If you click in some other state (or any polygon), only that state will be returned from the selection layer and used to find cities within (or close to) it.

Note In this chapter's application, the value of `slayer` is `Countries`, and this is hard-coded into the form. Since `Countries` is the only polygon layer in this map, it's the only possible choice for `slayer`.

FEATUREQUERY Mode

This mode is similar to `FEATUREQUERY`, but returns all matching features from the selection layer. In this mode, the `TOLERANCE` value specified for the selection layer is important. In this case, not only is the polygon that received the mouse click returned, but all polygons within the `TOLERANCE` distance of the click point are returned too. The subsequent query of the searchable layers then looks for features contained within any of the selected polygons, or within the `TOLERANCE` distance from the border of the selected polygons. It's important to note that if the `TOLERANCE` of the selection layer is set to 0, then `FEATUREQUERY` is functionally equivalent to `FEATUREQUERY` because then only the polygon containing the click point matches.

For example, let's say the `Countries`-layer `TOLERANCE` is set to 300 miles and the `Cities`-layer `TOLERANCE` is set to 100 miles. If a user selects `FEATUREQUERY` and clicks on Texas, then Texas, New Mexico, Oklahoma, and the Mexican states of Coahuila, Nuevo León, Chihuahua, and Tamaulipas will be selected. And in addition to cities within these states, cities outside the selected states but within 100 miles of the border (like Tulsa, Oklahoma and Springfield, Missouri) will also match.

ITEMFEATUREQUERY Mode

This mode combines an attribute search with a spatial search and returns the first match. It's similar to `FEATUREQUERY`, but instead of using a mouse click and `TOLERANCE` values to determine the selection layer matches, an attribute search is done on the selection layer. In this case, `slayer` must be a polygon layer. The `slayer` parameter determines which layer will be used for the attribute search. The parameter `qstring` contains the search expression. The first feature matched by the attribute search is then used to do a spatial query on the searchable layers, selecting features contained within the polygon feature. All matches are returned.

In the context of the data set supplied with the code, if the user selects `ITEMFEATUREQUERY` and sets the search expression to `(('[STATE]' eq 'Texas'))` or `(('[STATE]' eq 'Kansas'))`, the polygon features representing both Texas and Kansas will match the search expression, but only Kansas will be returned because it's found first. A spatial query is then performed on the `Cities` layer, returning all the cities in Kansas, because the Kansas record comes before Texas in the table.

ITEMFEATURENQUERY Mode

This mode is similar to ITEMFEATUREQUERY, but instead of returning only the first matching feature in `slayer`, it returns them all. If the user selects ITEMFEATURENQUERY with the same search string as before, the Cities layer is searched for cities in both states since both Texas and Kansas are matches.

INDEXQUERY Mode

An INDEXQUERY retrieves a single feature based in the shape index of the feature contained in the CGI form variable `shapeindex`. The shape index is available as the substitution string `[shpidx]`. Typically, INDEXQUERY is used in conjunction with a list containing summary information, including the shape index for each feature. A summary item will be formatted within anchor tags ``, where URL represents a request for MapServer to create and return a map. Clicking on the summary item will submit an INDEXQUERY to MapServer, which retrieves the feature associated with the shape index of that item. This chapter's application contains an example of this use of the INDEXQUERY mode.

After having defined the various query modes, it's time to see just how the results of a particular query are presented. The next section covers templates. Following that, querymaps and joins will be covered. Querymaps allow you to present the spatial selections made by a query, and joins allow you to access attributes from external dBase files.

Query Templates

The templates that are used to present the results of a query aren't complicated. But the fact that there can be several of them, referred to at different levels of the mapfile and used for different purposes, is the second biggest obstacle to overcome (after query modes) if you wish to take advantage of MapServer's query capabilities.

Recall that the main application HTML template specified in the WEB object is an almost complete web page with substitution strings that MapServer will replace with appropriate values. It has `<html>`, `<head>`, and `<body>` tags, and can be rendered by a browser even without string substitution. Query templates, however, aren't like this—they're HTML fragments that must be properly assembled to create a complete, renderable web page.

Query templates are specified in the mapfile. They can be specified at four levels: in the WEB object, in the LAYER object, in the CLASS object, and in the JOIN object. The role that a template plays depends on where it's defined.

Map-Level Query Templates

The WEB object provides three keywords that specify query templates: HEADER, FOOTER, and EMPTY. HEADER and FOOTER are used for multi-result queries. (Most query modes can return multiple results.) The EMPTY template is used for queries that return no results.

In order to display results of a multi-result query properly, MapServer needs to create a complete web page. Since it doesn't know beforehand how many results the query will return, it breaks the task into three steps:

1. Producing a header that contains the correct HTML preamble tags (like `<html>` and `<body>`), any other tags that might be required (Java, JavaScript, etc.), and any summary information available
2. Using appropriate layer and class templates to display the results for each match
3. Producing a footer that contains the HTML tags to close any open tags after the processing is done

The keyword `HEADER` specifies the path (absolute or relative to the mapfile) to the template file, which is processed before everything else when a multi-result query is performed (Step 1 in the previous list). The keyword `FOOTER` identifies the template file that's processed after everything else has been done (Step 3 in the previous list). Both the `HEADER` and `FOOTER` templates can contain substitution strings, and even forms and CGI variables. The same string substitution is performed on them as is performed on the `WEB` template file.

The keyword `EMPTY` specifies the URL of the file that's used if a query returns no results. It's important to note that this isn't a path on a local file system, but a URL relative to the Apache `DocumentRoot`. This file isn't an HTML fragment, but a complete web page that informs the user of the null result, and may offer some advice or a link back to the previous page. If absent, a standard MapServer error message is displayed.

Layer-Level Query Templates

There are only three keywords related to query templates at the `LAYER` level. Unfortunately, they have the same names as the `WEB`-level templates: `HEADER` and `FOOTER`, and the class-level template: `TEMPLATE`—which can lead to some confusion. As before, these query templates are only useful for multi-result queries.

At the `LAYER` level, you can assume that the `HEADER` template defined in the `WEB` object has already set up the initial HTML. But since the number of results for a particular layer isn't yet known, MapServer must prepare the HTML that will be used to present summary data for the current layer or graphics such as `querymaps` (discussed later), or to set up tables that will be used to present query detail. After the detailed results have been presented by means of a class- or layer-level template, you have to clean up and close any open HTML tags and perhaps display some summary information.

The keyword `HEADER` specifies the path (absolute or relative to the mapfile) to the file that will be processed before any results from the layer have been presented. The keyword `FOOTER` identifies the path to the file that will be processed after all the results for the layer have been sent. The keyword `TEMPLATE` specifies the path to the template that's used to present detailed results.

This approach provides a layer-level alternative to the class-level query template. Instead of defining the same template in every class of a layer that contains many classes, you can specify the template once at the layer level. The presence of a layer-level query template indicates to MapServer that a layer is searchable.

Note that you can put anything you want into these templates. There's no reason that layer summary information or graphics must be put in the `HEADER`. If you want something to appear after the detailed results have been displayed, just put the tags and substitution strings in the `FOOTER` template. But note that individual query results will be placed in the class- or layer-level query template.

Class-Level Query Templates

At the class level, there's only one query-related keyword. The keyword `TEMPLATE` specifies the path to the file that will be sent for each query result. You can think of this as a single line in a tabular report, but it need not be so simple. The template could, for example, include a map image for every result along with any text attribute information. But it's important to remember that every result will make use of this template, so a query that might return a thousand results is probably not a good candidate for the inclusion of an image for every match. In addition to its role in formatting and presenting query results, the presence of a class-level template is also a flag that indicates to MapServer that the layer is searchable.

So, to summarize what you've learned about query templates: detailed query results are formatted and presented by means of class-level templates (unless a layer-level template has been defined). Layer-level templates present layer summary data and set up the initial HTML structures (e.g., tables) that are used by the class-level templates. Map-level templates summarize results across all layers and set up the most basic HTML structures required by a web page (like `<html>` and `<body>`). The functions are fairly distinct—if you can keep the names straight, query templates are pretty easy to use.

The QUERYMAP Object

When querying a spatial database, it makes sense that you should be able to see the spatial results of your query—that is, matches should be highlighted on the map image in some way so that, for example, their spatial distribution will be visible. MapServer accomplishes this with the `QUERYMAP` object, which specifies the parameters that determine how matching features are rendered in a map image. For the map-only query modes, a map image is produced but tabular results aren't. For the others, a map image is by default not created. If you want to render a map in these modes, you must define a `QUERYMAP` object.

Note The image created by using a `QUERYMAP` object is retrieved by using the substitution string `[img]`. This is the same string used to retrieve a normal map image. This shouldn't present a problem because the `querymap` image will be displayed on a different page. However, it's possible to use a `querymap` as the main map image, which allows the performance of the usual interactive tasks such as panning and zooming, as well as making spatial queries.

A `QUERYMAP` object begins with the keyword `QUERYMAP` and is terminated by the keyword `END`. The keyword `COLOR` specifies the RGB components of the color used to highlight matching features. If omitted, `COLOR` defaults to yellow (255 255 0). The keyword `SIZE` specifies the width and height (in pixels) of the map image that's created. If omitted, the size defaults to the size specified in the mapfile. The keyword `STATUS` determines whether the `querymap` will be drawn. If the value is `on`, the map will be drawn; if `off`, it won't.

The keyword `STYLE` determines the highlighting behavior of the `querymap`. If set to `normal`, all features will be drawn as specified by layer setting (i.e., no highlighting is done). The value

hilite will cause the matching feature to be drawn in the COLOR specified (or in yellow if it's omitted), while non-matching features will be drawn without highlighting. If the value is selected, then only the matching features (in that layer) will be drawn—the rest won't be rendered. Note that layers not queried are always drawn without highlighting.

The JOIN Object

Associated with each shapefile is an attribute table stored in a dBase file. Because this is a very common format, there's often additional data in external dBase files. MapServer has the ability to join rows in the external files with rows in the attribute table. A join is defined at the layer level—which makes sense, since the JOIN object assumes that the attribute table for the layer is the FROM table.

Note Only dBase (i.e., DBF) files can be joined.

MapServer makes a join by specifying a FROM item in the attribute table and a TO item in the external table. When MapServer performs a query, it scans the attribute table and matches records based on the selection criteria specified (spatial or attribute). For each matching record, it compares the value of the FROM item with values of the TO item in all the records of the external table. If the TO item in the external record equals the FROM item in the selected feature, the items contained in the external record are appended to the attribute table record of the matching feature. This assumes that there's only a single record in the external table that has a TO value equal to the FROM value. If this is the case, then the join is one-to-one and will produce a single result for each matching feature. If there's more than one external record for an attribute record, then the join is one-to-many. A one-to-many join behaves as if there are separate (but identical) matching features for each external record that's joined.

Note In order to display attributes from the joined table in a query template, MapServer uses substitution strings with a particular syntax. This syntax is similar to the syntax for accessing ordinary attributes. Both will be described in the following section, "Substitution Strings and CGI Variables."

A JOIN object begins with the keyword JOIN and is terminated by the keyword END. Joins must have a unique name in order to be referenced in a template. The value of the keyword NAME identifies the JOIN object. The absolute path to the table that's to be joined is specified by the keyword TABLE. The keyword FROM specifies the item in the shapefile that's used to define one side of the join. The keyword TO specifies the item in the external table that defines the other side of the join. The default join type is one-to-one, but the keyword TYPE can be used to specify multiple (one-to-many) or single (one-to-one).

If a join is one-to-many, MapServer needs to know what to do with the multiple joined records. The keyword TEMPLATE identifies a template file that's processed once for each record

in the join. This template can only contain substitution strings for attributes in the joined table. A one-to-one join doesn't require this step—it uses the class-level query template defined in the mapfile. (See Chapter 11 for a description of the JOIN-level template.)

Note Attributes in the joined table are available for display, but they can't be queried.

Substitution Strings and CGI Variables

There are a great many substitution strings and CGI variables available to MapServer query applications. Some have been mentioned already and more will be described here. Some will be left to Chapter 11, “MapServer Reference.”

Query Substitution Strings

The general syntax of query substitution strings is the same as that already discussed—a name delimited by square brackets. This section will only describe substitution strings that will be used in this chapter's application. Note that they're available only when processing the results of queries.

[itemname], [itemname_esc], and [itemname_raw]

To access an item in a shapefile attribute table, use the item name delimited by square brackets. In other words, attribute names are treated the same way as CGI variables. In addition to the value of the item, escaped versions are available for use in URLs, as well as for raw values. For URLs, _esc is appended to the item name, and for raw values, _raw is appended to the item name.

Note When MapServer replaces a substitution string with a value, it doesn't know how that value is going to be used. By default, MapServer replaces some characters (such as < and >) with strings that won't be confused with HTML syntactic elements (such as < and >). The replacements always begin with & and end with a semicolon.

Similarly, the blank space, /, and & are syntactically meaningful constituents of URLs. MapServer can replace the blank space by substituting a +, and it can replace the other characters by concatenating a % symbol and the character's hexadecimal code. For example, the escaped version of the string /this is a URL is %2Fthis+is+a+URL%26. If neither of these options is appropriate, then unescaped, raw values are also available.

[joinname_itemname], [joinname_itemname_esc], and [joinname_itemname_raw]

To access an item in a joined table, append an underscore character and the item name to the join name specified in the mapfile, and enclose the string in square brackets. There are also escaped and raw versions of the joined items.

[nr], [nl], and [nlr]

These substitution strings all present summary data about the query. `[nr]` represents the total number of results, `[nl]` is the number of layers that are returning results, and `[nlr]` is the total number of results in the current layer.

[rn] and [lrn]

`[rn]` is the result number (i.e., a sequence number starting at 1 and counting all results regardless of layer). `[lrn]` is the result number (starting at 1) within the current layer.

[cl]

`[cl]` is the current layer name.

[id]

This last string isn't specifically query related, but it's used in retrieving saved queries, so I'm including it here. Every time MapServer is invoked, it generates a (more or less) unique session ID number from the system time and the process ID. This number is appended to the base map name defined in the mapfile, and used to identify the various components created in a single invocation (such as image names, queryfile names, etc.). This substitution string makes that session ID available for retrieving, for example, saved queryfiles.

Query CGI Variables

The following section describes some of the CGI variables available to MapServer query applications.

img, img.x, and img.y

When used as a substitution string (`[img]`) and embedded in a form, this variable contains the name of the map image to retrieve. However, as a CGI variable, it's used to return the image coordinates (in pixels) of a mouse click. The actual variables returned are `img.x` and `img.y`.

imgext [minx] [miny] [maxx] [maxy]

`imgext` contains the extent of the image that MapServer has already created and that's pointed to by the substitution string `[img]`. It consists of a space-delimited string of coordinates for the lower-left and top-right corners of the extent.

imgxy [x] [y]

`imgxy` specifies the image coordinates (in pixels) of a mouse click. This is used in the main application template to produce a "synthetic" click point at the center of the map image when the user clicks the `Refresh` button instead of the map image.

qlayer [name]

The value assigned to `qlayer` is the name (as used in the mapfile) of the layer to query. If omitted, all layers are queried in sequence.

qitem [name]

The value assigned to `qitem` is the name of the attribute to be queried. If omitted, all attributes are queried.

qstring

The value assigned to `qstring` is the expression used in attribute queries. This can be a simple text string, a regular expression, or a logical expression. The syntax is the same as the class `EXPRESSION` syntax in the mapfile.

queryfile [filename]

The value assigned to `queryfile` is the path to the queryfile that's loaded before any processing. This file is created if `savequery` is set to `true`.

savequery [true] [false]

When set to `true`, `savequery` causes MapServer to save query results in a file. This file is stored in the location pointed to by the map-level keyword `IMAGEPATH` in the mapfile. The name consists of the process ID appended to the map name (as defined in the mapfile), and an extension of `qy`.

slayer

`slayer` specifies the layer to select when performing a `FEATUREQUERY` or `ITEMFEATUREQUERY`. This layer *must* be a polygon layer.

A Query Application

Having reviewed aspects of MapServer that relate to queries, I'll now turn to a detailed analysis of a complete application. It will demonstrate the use of all the query modes described previously, but will be limited to a fairly simple map. The spatial data set consists of a polygon shapefile describing countries of the world and a point shapefile that contains city information. In order to demonstrate the use of joins, another dBase file is used that contains population information relating to some of the cities. The map has two searchable layers: the first for countries and the second for cities. You can retrieve all the code used in this book from the download area of the Apress website (www.apress.com). The spatial data set is also available from the same location. The files for this application can be found in the archive `fourth.zip`.

The Mapfile

The mapfile for this application is named `fourth.map`. The code in this file (with line numbers added) is shown in Listing 5-1. If you haven't downloaded the code, open a file with any text editor and enter the code from the listing, and then save it as `fourth.map`.

Lines 008 through 015 in the following code snippet should be familiar. The map name has been set to `Fourth` in Line 008. Map units are decimal degrees. The size of the map is 640 pixels by 320 pixels, and the background color of the image is set to blue in Line 012. The image type is GIF. The location of the spatial data sets is specified by `SHAPEPATH` in Line 014. The font set is identified in Line 015.

```
008 NAME "Fourth"                # used as base image name
009 UNITS dd                      # units are decimal degrees
010 EXTENT -180.0 -85.0 180.0 85.0 # map extent
011 SIZE 640 320                 # map image size in pixels
012 IMAGECOLOR 200 225 255       # background color
013 IMAGETYPE gif                # image type jpeg/gif/png
014 SHAPEPATH "/home/mapdata/"   # path to data directory
015 FONTSET "/var/www/htdocs/fontset.txt" # pointers to fonts
```

The WEB object in Lines 020 through 029 is very similar to the WEB objects defined in the previous applications. It defines the template file, the image path, and the image URL for this application. Remember that the template specified by the keyword `TEMPLATE` in the WEB object is the main template for the application—it's not a query template.

The WEB object contains three new keywords that are used by a MapServer query application. `HEADER` specifies the name of the top-level query template that's processed before anything else. `FOOTER` specifies the name of the top-level template that's processed after everything else has been sent. Recall that both of these files are just HTML fragments. They need to be assembled with other fragments to form a complete web page. On the other hand, the file specified by `EMPTY` is a complete web page that's sent if a query has no results. (I'll discuss the contents of all the template files used by this application in the next section, "The HTML Template.")

```
019 WEB
020     # A header/footer defined in a web object is displayed
021     # before/after any individual query response is made.
022     # It is displayed only once.
023     #
024     HEADER "/var/www/htdocs/fourth_web_header.html"
025     FOOTER "/var/www/htdocs/fourth_web_footer.html"
026     EMPTY  "/fourth_empty.html"           # URL
027     TEMPLATE "/var/www/htdocs/fourth.html"
028     IMAGEPATH "/var/www/htdocs/tmp/"
029     IMAGEURL "/tmp/"                      # URL
030 END
```

Following the WEB object is the reference map defined in Lines 034 through 041. The reference image used is `fourth_worldref.gif` (included in the source distribution) and its size is 320 pixels by 160 pixels. Setting at least one `COLOR` component to `-1` means that the fill color of the reference box is transparent. Finally, the `OUTLINECOLOR` of the box is red.

```

034 REFERENCE
035     IMAGE "/var/www/htdocs/fourth_worldref.gif"
036     SIZE 320 160
037     EXTENT -180.0 -85.0 180.0 85.0
038     STATUS ON
039     COLOR -1 -1 -1
040     OUTLINECOLOR 255 0 0
041 END

```

Next is the `QUERYMAP` object defined in Lines 045 through 050. Setting `STATUS on` means that the a map image will be rendered. `STYLE hilite` means that the selected feature will be highlighted in the `COLOR` yellow, while unselected features will be rendered according to the parameters specified in the layer objects. The size of the image is 320 pixels by 160 pixels.

```

045 QUERYMAP
046     STATUS on           # draw query map
047     STYLE hilite        # highlight selected feature
048     COLOR 255 255 0     # in yellow
049     SIZE 320 160
050 END

```

Note A querymap image will only be created if a `QUERYMAP` object is defined, but the image won't be displayed unless the appropriate syntax is inserted into one of the query templates.

A polygon layer named `Countries` is specified in Lines 054 through 078. It retrieves its spatial data from a shapefile named `countries`.

```

054 LAYER
055     NAME "Countries"
056     STATUS on
057     TYPE polygon
058     DATA "countries"

```

The keyword `HEADER` in Line 062 identifies the template that will be processed before any results are returned from this layer. The keyword `FOOTER` identifies the template that will be processed after all results from this layer have been returned.

```

062     HEADER "/var/www/htdocs/fourth_countries_header.html"
063     FOOTER "/var/www/htdocs/fourth_countries_footer.html"

```

When a mouse click occurs on a map image while MapServer is in a spatial query mode, the program looks for features that are nearby—if they're close enough, MapServer counts them as matches. The keyword `TOLERANCE` sets the numerical limit of "close enough," and

TOLERANCEUNITS defines the units of TOLERANCE. As noted previously, the default TOLERANCE is 3 and the default TOLERANCEUNITS is pixels, but TOLERANCEUNITS can take any one of the following values: pixels, feet, inches, kilometers, meters, miles, or dd.

```
064      TOLERANCE 1           # must be within 1 tolerance unit
065      TOLERANCEUNITS miles  # units for tolerance values is miles
```

An unnamed CLASS is defined in Lines 66 through 77. There's no class EXPRESSION, so MapServer defaults to selecting all features in the data set, and each feature is rendered in the gray COLOR specified. There's a new class-level keyword, TEMPLATE. This specifies the query template that will be used to present each matching record whenever this layer is queried. Recall that this query template is merely an HTML fragment and must be assembled with other fragments to build a complete web page. Since this layer isn't used to label features—that will be done in an ANNOTATION layer—no LABEL object is defined.

```
066      CLASS
067          # A template defined at the class level is used to
068          # display the results for each response to a query. If a
069          # query results in N hits, then the template will be used
070          # N times. To be queriable a layer must specify a CLASS
071          # level template.
072          #
073          TEMPLATE "/var/www/htdocs/fourth_countries_query.html"
074          STYLE
075              COLOR 199 199 199
076          END
077      END # end class
078  END # end layer
```

A point layer named Cities is defined in Lines 082 through 127. It uses the shapefile cities. The CITY item in the data set will be used to label the features. LABELCACHE is set to on, which allows labels to be placed without interfering with other labels.

```
082  LAYER
083      NAME "Cities"
084      STATUS on
085      TYPE point
086      DATA "cities"
087      LABELITEM "CITY"          # labels use value in column "CITY"
088      LABELCACHE on
```

Query HEADER and FOOTER templates are specified for this layer in Lines 092 and 093 to present layer summary information. TOLERANCE and TOLERANCEUNITS are specified to define how close a mouse click has to be to a feature for a match to be accepted.

```
092      HEADER "/var/www/htdocs/fourth_cities_header.html"
093      FOOTER "/var/www/htdocs/fourth_cities_footer.html"
094      TOLERANCE 1           # must be within 1 tolerance unit
095      TOLERANCEUNITS miles  # units for tolerance values is miles
```

An unnamed CLASS is specified in Lines 096 through 115. Since no EXPRESSION is used to classify the data, each feature in the data set will be rendered as a single black pixel. The keyword TEMPLATE specifies the class-level query template that will be used to present each result.

```

096      CLASS
101          TEMPLATE "/var/www/htdocs/fourth_cities_query.html"
102          STYLE
103              COLOR 0 0 0      # symbol color is black
104          END

```

A LABEL is defined in Lines 105 through 114. It renders labels in bold, red, 8-point Arial. The label text is antialiased and centered below the feature it labels. Each label is drawn with a white BACKGROUND COLOR. Labels for the same feature must be at least 50 pixels apart.

```

105      LABEL
106          TYPE truetype      # use truetype font
107          FONT "arialbd"     # use arial bold
108          SIZE 8             # use 8 point size
109          COLOR 255 0 0      # color text red
110          BACKGROUND COLOR 255 255 255 # render text on white bg
111          MINDISTANCE 50     # labels > 50 pixels apart
112          POSITION lc         # center labels below feature
113          ANTIALIAS true     # antialias the text
114      END # end label
115  END # end class

```

A join is defined in Lines 121 through 126. The external table to which the join is made is specified by the keyword TABLE. Joins are only possible between dBase tables. The join is given a NAME so that it can be referenced in the query template. Next, the two items that are used to link a record in the shapefile to a record in the external table are given by the keywords FROM and TO. FROM identifies an item in the shapefile attribute table, and TO identifies the corresponding item in the external table. In this case, the items have the same name, but this isn't a requirement. The layer is terminated in Line 127.

```

121      JOIN
122          TABLE "/var/www/htdocs/fourth_join.dbf"
123          NAME "test-join"
124          FROM "CITY"
125          TO "CITY"
126      END
127  END # end layer

```

A line layer is defined in Lines 131 through 141. It's based on the same data set as that used for the Countries layer. As in previous applications, line layers are used to draw the boundaries of polygon layers. Recall that only a polygon can have a fill color, but a polygon border can only be rendered as a 1-pixel-wide line. In order to draw a wider border around a filled polygon, it's necessary to render it in two steps—in the first step, the area is filled but no border is specified; in the second step, the border is drawn. In the present case, the default symbol, which is a one-pixel-wide black line, is used. Because of this, you could actually omit this layer by instead

specifying an `OUTLINECOLOR` in the default class in the `Countries` layer. This would also create a 1-pixel-wide line.

```

131 LAYER
132     NAME "Boundaries"
133     STATUS default
134     TYPE line
135     DATA "countries"
136     CLASS
137         STYLE
138             COLOR 0 0 0
139         END
140     END # end class
141 END # end layer

```

Finally, an `ANNOTATION` layer for the `Countries` layer is specified in Lines 144 through 165. The label is rendered as italic, 8-point Arial on a white background. The text is antialiased and labels are centered on the polygon. Labels for the same feature must be at least 50 pixels apart.

```

144 LAYER
145     STATUS DEFAULT           # this layer is always rendered
146     TYPE annotation
147     DATA "countries"
148     LABELITEM "STATE"       # labels use value in column "STATE"
149     LABELCACHE on
150     CLASS                   # class renders line & label
151         STYLE
152             COLOR 0 0 0     # line color is black
153         END
154         LABEL
155             TYPE truetype   # use truetype font
156             FONT "arialbi"  # use arial bold
157             SIZE 8          # use 8 point size
158             COLOR 0 0 0     # color text black
159             BACKGROUNDCOLOR 255 255 255 # render text on white bg
160             MINDISTANCE 50  # labels > 50 pixels apart
161             POSITION cc      # labels in center of feature
162             ANTIALIAS true  # antialias the text
163         END # end label
164     END # end class
165 END # end layer
166 END # end of map file

```

Surprisingly, this mapfile contains only a few new keywords. Nevertheless, these few additions allow you to produce an interactive spatial query application that's quite elegant. Of course, you'll need to provide some query templates to display results, but these are very simple HTML fragments. In fact, the most complicated HTML will be the main application template, but this is really no more complicated than in previous applications. This is examined in the next section.

The Initialization File

This application uses a separate HTML initialization file, as did the previous one. The code for the initialization file is contained in the file `fourth_i.html`, and the contents (with added line numbers) are shown in Listing 5-2. This initialization file differs little from those created in Chapters 3 and 4—it identifies the name of the program and mapfile to use, it sets the value of `zoomsize`, and it specifies that both layers (Countries and Cities) should be drawn.

```

001 <html>
002   <head><title>MapServer Fourth Application</title></head>
003   <body>
004     <form method=GET action="/cgi-bin/mapserv">
005       <input type="submit" value="Click Me">
006       <input type="hidden" name="program"
007         value="mapserv">
008       <input type="hidden" name="map"
009         value="/home/mapdata/fourth.map">
010       <input type="hidden" name="zoomsize
011         size=2 value=2>

```

Lines 012 through 015 show the first instance of MapServer's ability to set mapfile parameters based on CGI form variables. A CGI reference to a mapfile parameter consists of the sequence of named mapfile hierarchy objects leading to the parameter, separated by the underscore character. In this case, the keyword `TOLERANCE` is found in the `Cities` layer, which is contained in the mapfile. So the name used to reference this value is `map_Cities_tolerance`. Here, it has been assigned a value of 100 units. The value of keyword `TOLERANCEUNITS` in that layer of the mapfile determines whether the `TOLERANCE` is measured in pixels, meters, miles, or one of the other possible units. This process is repeated to set the `TOLERANCE` for the `Countries` layer. Lines 018 and 019 specify the selection layer (`slayer`), which is set to `Countries`. The `Countries` layer is the only searchable polygon layer in the mapfile, so no other value makes sense. It can't be changed from the template file.

```

012     <input type="hidden" name="map_Cities_tolerance
013       size=4 value=100>
014     <input type="hidden" name="map_Countries_tolerance
015       size=4 value=100>
016     <input type="hidden" name="layers"
017       value="Cities Countries">
018     <input type="hidden" name="slayer"
019       value="Countries">

```

Lines 020 through 027 initialize the values of several query-related CGI variables to nulls. These variables can later be set interactively in the template file.

The form variable `mapshape` contains the list of vertex coordinates (in map units) that define the shape of the query region. The first coordinate pair must be the same as the last pair. `imgshape` is used in the same way as `mapshape`, but the coordinates are expressed in image (pixel) coordinates. The variable `imgbox` contains the top-left and bottom-right coordinates (in image units) of a rectangular region that will be used as a query region. This is the reverse of the usual extent since image coordinates increase downward.

```

020      <input type="hidden" name="mapshape"   value="">
021      <input type="hidden" name="imgshape"   value="">
022      <input type="hidden" name="imgbox"     value="">

```

Lines 023 through 027 initialize the query expression (qstring), the query layer (qlayer), the query item (qitem), the shape index (shapeindex), and savequery. The variable shapeindex will be used later to select features based on the shape index. Lines 029 and 030 close the open tags.

```

023      <input type="hidden" name="qstring"     value="">
024      <input type="hidden" name="qlayer"      value="">
025      <input type="hidden" name="qitem"       value="">
026      <input type="hidden" name="shapeindex"  value="">
027      <input type="hidden" name="savequery"   value="">
028      </form>
029  </body>
030 </html>

```

The HTML Template

The template file for this application is `fourth.html`. The code is shown in Listing 5-3. Lines 003 through 018 of the file are new. Since MapServer has no HTML mechanism for remembering its previous state, the short JavaScript function `setMode()` is used to reset the select state to reflect the value of the hidden variable `previousmode`, which is a text string that contains the mode name. It does this by scanning the array containing the elements of the mode check box until it finds a match. It then sets the variable `selectedIndex` to the sequence number of the matching mode. In Line 018, the `onload` event is used to trigger the execution of `setMode()`.

```

001 <html><!-- fourth.html -->
002 <head><title>MapServer Fourth Application</title>
003 <script language="JavaScript" type="text/javascript">
004 <!--
005 function setMode()
006 // set map mode to the previous mode
007 {   document.the_map.mode.selectedIndex=0;
008     for (i=0;i<document.the_map.mode.length;i++){
009         if (document.the_map.mode[i].value ==
010             document.the_map.previousmode.value){
011             document.the_map.mode.selectedIndex=i;
012         }
013     }
014 };
015 // -->
016 </script>
017 </head>
018 <body bgcolor="#E6E6E6" onload="setMode()">

```

Lines 019 through 047 should be familiar from previous templates. They present the map and reference images, display some map information, and set up the navigation controls for pan and zoom.

Lines 048 through 054 set the binary check box values for the layer variable that returns a list of layers to be rendered. Similar to the select variables associated with an option value, the CGI variable `Cities_check` represents the state of the check box associated with the `Cities` value of the variable `layer`. If the check box is checked, then MapServer sets the value of `Cities_check` to `Checked`. If not, it's set to null. MapServer substitutes current values for these bracketed substitution strings when the template is scanned before sending back to the user.

Some new material is shown in Lines 059 through 081, which contains a list of modes that the user can now select. This list of modes includes all of MapServer's query modes except for the map-only modes. Recall that the map-only modes produce a map but no tabular output.

```

059         <td rowspan=1 valign="top" align="left">
060             <select name="mode">
061                 <option value="browse">
062                     Browse</option>
063                 <option value="query">
064                     Query</option>
065                 <option value="nquery">
066                     Nquery</option>
067                 <option value="itemquery">
068                     Itemquery</option>
069                 <option value="itemnquery">
070                     Itemnquery</option>
071                 <option value="featurequery">
072                     Featurequery</option>
073                 <option value="featurenquery">
074                     Featurenquery</option>
075                 <option value="itemfeaturequery">
076                     Itemfeaturequery</option>
077                 <option value="itemfeaturenquery">
078                     Itemfeaturenquery</option>
079                 <option value="indexquery">
080                     Indexquery</option>
081             </select></td>

```

Following are several blocks that allow the user to change the values of query-related variables interactively. Lines 082 through 086 set the query layer.

```

082         <td align="right">
083             Query layer:</td>
084         <td align="left">
085             <input type="text" name="qlayer" size=10
086                 value="[qlayer]"></td>

```

Lines 087 through 091 set the imgbox coordinates.

```

087         <td align="right">
088             imgbox coords:</td>
089         <td align="left">
090             <input type="text" name="imgbox" size=25
091                 value="[imgbox]"></td></tr>

```

Lines 095 through 099 set the query item.

```

095         <td align="right">
096             Query item:</td>
097         <td align="left">
098             <input type="text" name="qitem" size=10
099                 value="[qitem]"></td>

```

Lines 100 through 104 set the `imgshape` coordinates.

```

100          <td align="right">
101              imgshape coords:</td>
102          <td align="left">
103              <input type="text" name="imgshape" size=25
104                  value="[imgshape]"></td></tr>

```

Lines 108 through 112 set the query string.

```

108          <td align="right">
109              Query string:</td>
110          <td align="left">
111              <input type="text" name="qstring" size=25
112                  value="[qstring]"></td>

```

Lines 113 through 117 set the `mapshape` coordinates.

```

113          <td align="right">
114              mapshape coords:</td>
115          <td align="left">
116              <input type="text" name="mapshape" size=25
117                  value="[mapshape]"></td></tr>

```

Lines 121 through 127 set the `TOLERANCE` value for the `Cities` layer.

```

121          <td align="right">
122              Cities Tolerance:</td>
123          <td align="left">
124              <input type="text"
125                  name="map_Cities_tolerance" size=4
126                  value="[map_Cities_tolerance]"
127                  miles</td>

```

Lines 128 through 132 set the `shapeindex`.

```

128          <td align="right">
129              Shape index:</td>
130          <td align="left">
131              <input type="text" name="shapeindex" size=10
132                  value="[shapeindex]"></td></tr>

```

Lines 136 through 142 set the `TOLERANCE` value for the `Countries` layer.

```

136          <td align="right">
137              Countries Tolerance:</td>
138          <td align="left">
139              <input type="text"
140                  name="map_Countries_tolerance" size=4
141                  value="[map_Countries_tolerance]"
142                  miles</td>

```


Finally, Lines 149 through 155 set the current values of the hidden form variables that are initialized to nulls in the initialization file. These are used to maintain state. The value of `imgxy` represents a sort of synthetic mouse click. If the user selects a spatial query mode and clicks Refresh rather than the map image, MapServer will need a mouse click location to search. In this case, it's set to the center of the image. `imgext` and `[mapext]` are used to track changes to the map image as the user zooms and pans. The value of `imgext` for the next invocation is set to the `mapext` of the previous invocation. Lines 151 and 152 store the name of the mapfile and the name of the program to use. The variable `slayer` is set permanently to `Countries` since there's no other searchable polygon layer defined in the mapfile. The previous mode is saved, and `savequery` is set to `true`.

```

149     <input type="hidden" name="imgxy" value="320 160">
150     <input type="hidden" name="imgext" value="[mapext]">
151     <input type="hidden" name="map" value="[map]">
152     <input type="hidden" name="program" value="[program]">
153     <input type="hidden" name="slayer" value="Countries">
154     <input type="hidden" name="previousmode" value="[previousmode]">
155     <input type="hidden" name="savequery" value="true">

```

Lines 156 through 157 close the open tags.

```

156 </form>
157 </body>

```

The Query Templates

The query templates provide MapServer the means to present query results. (Indirectly, they also tell MapServer which layers are searchable, since a layer without a class- or layer-level template can't be searched.) Query templates are HTML fragments that MapServer glues together to create a web page that displays the results of a query. Although it might seem to be a complicated way of accomplishing that result, MapServer really has no choice. It doesn't know in advance which layers may have query results, and it doesn't know how many results each layer might have. As such, the job of formatting has to be broken up into chunks. This application uses eight query templates. Several are trivial and all are short.

Map-Level Templates

Map-level query templates are defined within the `WEB` object. There are two templates: a `HEADER` and a `FOOTER`, and a complete web page for reporting a query that returns no results.

The WEB HEADER Template

The map-level query template `fourth_web_header.html` is specified by the keyword `HEADER` (on Line 024 of the mapfile) and shown in Listing 5-4. The template is used by MapServer to display information that summarizes the result set as a whole. Note that it begins with the HTML tags that start every web page: `<html>`, `<head>`, and `<body>`. General but useful information, such as the geographical extent of the displayed map image and the point at which the mouse click occurred, is given in Lines 007 through 011. (You might want to know this since a spatial search won't find a match with any feature located outside the viewable extent of the map image.)

Note that the click point is given in both image coordinates (number of pixels from the left-hand side and top of the image) and map coordinates (degrees of latitude and longitude if the map projection is geographic; meters, kilometers, or miles if it's not).

```

001 <html>
002 <head><title>MapServer - Fourth Application</title></head>
003 <body bgcolor="#E6E6E6">
004   <font size="4">Query Results</font>
005   <table border="1">
006     <tr><td>Search extent:</td>
007       <td>[imgext]<br></td></tr>
008     <tr>
009       <td>Click Point:</td>
010       <td>[img.x], [img.y] (Image coordinates)<br>
011         [mapx], [mapy] (Map coordinates)</td>

```

Query summary results are reported in Lines 015 and 016 using the query-specific substitution strings [nl] and [nr], which represent the number of layers with results and the total number of results, respectively.

```

012   </tr>
013   <tr>
014     <td>Query Results:</td>
015     <td>Layers with results=[nl]<br>
016       Total number of results=[nr]</td>
017   </tr>
018 </table>

```

These results are formatted in a table that's contained entirely within the HEADER template. The table is terminated at Line 018.

The WEB FOOTER Template

The code for the template `fourth_web_footer.html`, defined on Line 025 of the mapfile by the keyword `FOOTER`, is shown in Listing 5-5. Lines 001 through 017 define a URL template that saves the current state of the application. It contains substitution strings and is embedded in an anchor tag. When the user clicks the link, MapServer reloads the original application template with the saved values. It's a cleaner interface than one that would require the user to click the Back button to return to the previous page.

```

001 <a href="http://localhost/cgi-bin/mapserv?
002   program=mapserv&map=%2Fvar%2Fwww%2Fhtdocs%2Ffourth.map&
003   zoomsize=[zoomsize]&
004   map_Cities_tolerance=[map_Cities_tolerance]&
005   map_Countries_tolerance=[map_Countries_tolerance]&
006   layers=[layers_esc]&
007   slayer=Countries&
008   mapshape=[mapshape_esc]&
009   imgshape=[imgshape_esc]&
010   imgbox=[imgbox_esc]&

```

```

011 qstring=[qstring]&
012 qlayer=[qlayer]&
013 qitem=[qitem]&
014 shapeindex=[shapeindex]&
015 savequery=[savequery]&
016 previousmode=[mode]&
017 mode=browse">

```

Because this is a URL, spaces aren't allowed, so escaped versions of the substitution strings [layers_esc], [mapshape_esc], [imgshape_esc], and [imgbox_esc] are used. The string must also be formatted on a single line (which won't fit the page width of this book) or on multiple lines with no intervening spaces.

Line 018 identifies the link to the user, and Lines 019 and 020 close the open tags and terminate the web page.

```

018 Return to Query Definition</a>
019 </body>
020 </html>

```

The EMPTY Page

The web page specified by the keyword EMPTY on Line 026 of the mapfile is shown in Listing 5-6. Lines 004 through 006 indicate an empty result set and tell the user what to do. Although the code shown here is very simple, a more usable interface that leads the user back to the query page would probably be warranted in a production environment.

```

004 <h3> The query returned no results.</h3>
005 <h3> Click on the Back button on your browser to return
006 to the previous page.</h3>

```

The Countries-Layer Templates

The Countries layer contains three templates: a HEADER and FOOTER at the layer level, and a TEMPLATE contained in the default class.

The Layer-Level HEADER Template

The code for the layer-level query HEADER for the Countries layer is found in the file `fourth_countries_header.html` (specified by the keyword HEADER on Line 062 of the mapfile) and shown in Listing 6-7. Note that it *doesn't* begin with opening HTML tags—this is a fragment, not a complete web page. It formats some text, sets up a couple of tables, and refers to two images with the substitution strings [img] and [ref].

```

001 <HR>
002 <b>Layer: countries</b>
003 <table>
004 <tr><th align="left">Query map</th>
005 <th align="left">Reference map</th></tr>
006 <tr><td></td>
007 <td></td></tr>
008 </table>

```

There's more to this image than might first appear. The `[img]` string is usually replaced by an image of the map that's created based on `mapfile` parameters. But since this HTML will be processed in a query mode only, the image displayed will be the `querymap` image described in the `QUERYMAP` object of the `mapfile`. Recall that a `querymap` highlights the selected features. In this application, the `QUERYMAP` object specifies that the map size should be 320×160 pixels, and the selected feature should be highlighted in yellow (`COLOR 255 255 0`).

Note The querymap will have the same extent as the main map from which the query was initiated. Most of the formatting will be the same as well. The QUERYMAP object usually defaults to the main map parameters defined in the mapfile.

The table that begins with the `<table>` tag in Line 009 specifies a row of headings that's terminated by the `</tr>` tag in Line 016. This isn't a valid termination for a web page, and it indicates that there's more to come.

[illegible]

The Class-Level Template

In order to display tabular query results from the Countries layer, MapServer uses a class-level query template (specified by the keyword `TEMPLATE` on Line 082 of the mapfile) that's processed once for each matching feature. The code for this template is found in `fourth_countries_query.html` and shown in Listing 5-8. This template consists of a single row added to the table that's opened in the `fourth_countries_header.html` template.

Every result in a layer is assigned a sequence number, starting at 1. This *layer result number* is referenced by the substitution string [1rn] in Line 001 and causes the layer result number to be displayed for every result.

Line 002 displays the shape index of this feature. Line 003 and Lines 012 through 015 are table elements that contain an item name from the data set underlying the layer. These item names are delimited by square brackets so that MapServer recognizes them as substitution strings. Each substitution string is then replaced by its attribute value from the matching feature.

```
001 <tr><td>[lrm]</td>
002     <td>[shpid]</td>
003     <td>[COUNTRY]</td>
004     <td>
...
012     [STATE]
```

```

013      </a></td>
014      <td>[REGION]</td>
015      <td>[CONTINENT]</td>
016 </tr>

```

Lines 005 through 011 present another example of URL templates. In this case, the shape index for every matching feature is embedded in a link attached to the STATE. When this link is clicked, MapServer performs a query with mode set to `indexquerymap`. A querymap image is created with the specified feature highlighted. The shape extent substitution string `[shpext_esc]` uses the escaped form because it contains embedded blanks. Specifying `mapext=shape` means the extent of the map that's created will be just a little larger than the extent of the feature.

```

005 <a href="http://[host]/cgi-bin/[program]?
006 mode=indexquerymap&
007 qlayer=Countries&
008 shapeindex=[shpidx]&
009 shpext=[shpext_esc]&
010 mapext=shape&
011 map=[map]">

```

The Layer-Level FOOTER Template

When MapServer has finished processing all matching features in this layer, it sends the FOOTER template specified for this layer (specified by the keyword FOOTER on Line 063 of the mapfile) back to the browser. The code for this FOOTER template can be found in `fourth_countries_footer.html` and is shown in Listing 5-9. This code consists of a single line that closes the table opened in `fourth_countries_header.html`.

The Cities-Layer Templates

The Cities layer also contains three templates: HEADER and FOOTER (defined at the layer level), and TEMPLATE (defined at the class level).

The Layer-Level HEADER Template

The code for the layer-level query HEADER template for the Cities layer is found in the file `fourth_cities_header.html` (specified by the keyword HEADER on Line 092 of the mapfile) and shown in Listing 5-10. Again, this is a code fragment, not a complete page. Some formatting is done, a table is opened, and a row of table headings is produced. The template ends with the row termination tag `</tr>` in Line 010.

```

001 <font size+1><b>Layer: cities</b></font>
002 <table border=1>
003 <tr bgcolor=#CCCCC><td bgcolor=#ffffff>&nbsp;</td>
004 <th>shpidx</th>
005 <th>CITY</th>
006 <th>STATE</th>
007 <th>COUNTRY</th>
008 <th>LAT</th>
009 <th>LONG</th>
010 <th>POPULATION(from join)</th></tr>

```

The Class-Level Template

The code for the class-level query template for the Cities layer is found in `fourth_cities_query.html` and shown in Listing 5-11. Its name is specified by the keyword `TEMPLATE` on Line 101 of the mapfile. This template presents attributes from the selected Cities feature as a single table row. Lines 003 through 007 are substitution strings that consist of attribute names from the spatial database enclosed in square brackets. Line 001 displays the layer result number for every result.

```
001 <tr><td>[ln]</td>
002     <td>[shpid]</td>
003     <td>[CITY]</td>
004     <td>[STATE]</td>
005     <td>[COUNTRY]</td>
006     <td>[LAT]</td>
007     <td>[LONG]</td>
008     <td>[test-join_POP]</td></tr>
```

In the Cities layer of the mapfile, a join is defined that joins the cities dBase attribute table with an external table based on the CITY attribute. The external table contains only three fields: the city name, the country name, and the population. The field named CITY is used as the TO value when the join is defined. The field named POP contains a number that represents the approximate population of the city. To reference the value of POP from a template, an underscore character and the name of the field are appended to the name of the join (in this case, `test-join`) and enclosed in square brackets (i.e., `[test-join_POP]`), as shown in Line 011. This template will be processed once for every matching feature in the Cities layer.

The Layer-Level FOOTER Template

After all the results from this layer have been processed, the FOOTER template for the Cities layer will be processed. This template is named `fourth_cities_footer.html` (specified by the keyword `FOOTER` on Line 093 of the mapfile) and shown in Listing 5-12. It contains a single line that closes the table opened in the HEADER template `fourth_cities_header.html`.

Summary

The last several chapters have demonstrated the use of MapServer's CGI interface, and have made clear that applications of real utility can be built with CGI. You should take a moment to consider the elegance of an application that can do so much and be so flexible without the necessity of program writing. However, applications that need a more sophisticated user interface, or that have more complex query capabilities, will require other tools. Server-side scripts and real database management systems are two examples of how the functionality of MapServer might be increased.

The next three chapters will introduce a more powerful way to access MapServer's capabilities: MapScript. MapScript is an API (application program interface) that allows applications written in several language to make calls to MapServer functions and then use the results in whatever way

the developer wishes. MapScript frees the developer from the limitations of the web-based user interface and provides access to other tools (such as DBMSs) so that very powerful applications can be constructed. Such applications can provide map-rendering and spatial-query capabilities as part of a suite of tools.

Three flavors of MapScript will be described: Perl MapScript, Python MapScript, and PHP MapScript. You'll use each to build applications that provide a nearly identical look, feel, and functionality, in order for you to gain a clear understanding of their similarities and differences.

Code Listings

The code for this chapter is presented here without interruption.

Listing 5-1. *The mapfile fourth.map*

```

001 #####
002 #
003 # MapServer Fourth application
004 #
005 #####
006 # Map object
007 #
008 NAME "Fourth"                # used as base image name
009 UNITS dd                     # units are decimal degrees
010 EXTENT -180.0 -85.0 180.0 85.0 # map extent
011 SIZE 640 320                 # map image size in pixels
012 IMAGECOLOR 200 225 255       # background color
013 IMAGETYPE gif                 # image type jpeg/gif/png
014 SHAPEPATH "/home/mapdata/"   # path to data directory
015 FONTSET "/var/www/htdocs/fontset.txt" # pointers to fonts
016 #####
017 # Web object
018 #
019 WEB
020     # A header/footer defined in a web object is displayed
021     # before/after any individual query response is made.
022     # It is displayed only once.
023     #
024     HEADER "/var/www/htdocs/fourth_web_header.html"
025     FOOTER "/var/www/htdocs/fourth_web_footer.html"
026     EMPTY "/fourth_empty.html" # URL
027     TEMPLATE "/var/www/htdocs/fourth.html"
028     IMAGEPATH "/var/www/htdocs/tmp/"
029     IMAGEURL "/tmp/"           # URL
030 END

```

```

031 #####
032 # Reference map
033 #
034 REFERENCE
035     IMAGE "/var/www/htdocs/fourth_worldref.gif"
036     SIZE 320 160
037     EXTENT -180.0 -85.0 180.0 85.0
038     STATUS ON
039     COLOR -1 -1 -1
040     OUTLINECOLOR 255 0 0
041 END
042 #####
043 # Querymap object
044 #
045 QUERYMAP
046     STATUS on                # draw query map
047     STYLE hilite             # highlight selected feature
048     COLOR 255 255 0          # in yellow
049     SIZE 320 160
050 END
051 #####
052 # Country layer
053 #
054 LAYER
055     NAME "Countries"
056     STATUS on
057     TYPE polygon
058     DATA "countries"
059     # A header or footer defined at the layer level is displayed
060     # if that layer is a query layer. It is displayed only once.
061     #
062     HEADER "/var/www/htdocs/fourth_countries_header.html"
063     FOOTER "/var/www/htdocs/fourth_countries_footer.html"
064     TOLERANCE 1               # must be within 1 tolerance unit
065     TOLERANCEUNITS miles     # units for tolerance values is miles
066     CLASS
067         # A template defined at the class level is used to
068         # display the results for each reponse to a query. If a
069         # query results in N hits, then the template will be used
070         # N times. To be queriable a layer must specify a CLASS
071         # level template.
072         #
073         TEMPLATE "/var/www/htdocs/fourth_countries_query.html"
074         STYLE
075             COLOR 199 199 199
076     END

```



```

077     END # end class
078 END # end layer
079 #####
080 # Cities layer
081 #
082 LAYER
083     NAME "Cities"
084     STATUS on
085     TYPE point
086     DATA "cities"
087     LABELITEM "CITY"          # labels use value in column "CITY"
088     LABELCACHE on
089     # A header or footer defined at the layer level is displayed
090     # if that layer is a query layer. It is displayed only once.
091     #
092     HEADER "/var/www/htdocs/fourth_cities_header.html"
093     FOOTER "/var/www/htdocs/fourth_cities_footer.html"
094     TOLERANCE 1              # must be within 1 tolerance unit
095     TOLERANCEUNITS miles    # units for tolerance values is miles
096     CLASS
097         # A template defined at the class level is used to display
098         # the results for each response to a query. If a result set
099         # contains N, then the template will be used N times.
100         #
101         TEMPLATE "/var/www/htdocs/fourth_cities_query.html"
102         STYLE
103             COLOR 0 0 0      # symbol color is black
104         END
105         LABEL
106             TYPE truetype    # use truetype font
107             FONT "arialbd"    # use arial bold
108             SIZE 8            # use 8 point size
109             COLOR 255 0 0     # color text red
110             BACKGROUNDCOLOR 255 255 255 # render text on white bg
111             MINDISTANCE 50    # labels > 50 pixels apart
112             POSITION lc        # center labels below feature
113             ANTIALIAS true    # antialias the text
114         END # end label
115     END # end class
116     # To use information stored in a DBF file external to a shape
117     # file requires a JOIN. You must identify the external file
118     # with the keyword TABLE. The NAME is the reference to use in
119     # the template file. To link the shape to the external DBF,
120     # FROM and TO specify the fields that must match.

```

```

121 JOIN
122     TABLE "/var/www/htdocs/fourth_join.dbf"
123     NAME "test-join"
124     FROM "CITY"
125     TO "CITY"
126 END
127 END # end layer
128 #####
129 # Line layer for Country boundaries
130 #
131 LAYER
132     NAME "Boundaries"
133     STATUS default
134     TYPE line
135     DATA "countries"
136     CLASS
137         STYLE
138             COLOR 0 0 0
139         END
140     END # end class
141 END # end layer
142 #####
143 # Annotation layer for Countries
144 LAYER
145     STATUS DEFAULT          # this layer is always rendered
146     TYPE annotation
147     DATA "countries"
148     LABELITEM "STATE"      # labels use value in column "STATE"
149     LABELCACHE on
150     CLASS                  # class renders line & label
151         STYLE
152             COLOR 0 0 0    # line color is black
153         END
154         LABEL
155             TYPE truetype  # use truetype font
156             FONT "arialbi" # use arial bold
157             SIZE 8         # use 8 point size
158             COLOR 0 0 0    # color text black
159             BACKGROUNDCOLOR 255 255 255 # render text on white bg
160             MINDISTANCE 50 # labels > 50 pixels apart
161             POSITION cc     # labels in center of feature
162             ANTIALIAS true  # antialias the text
163         END # end label
164     END # end class
165 END # end layer
166 END # end of map file

```

Listing 5-2. *The HTML initialization file fourth_i.html*

```

001 <html>
002   <head><title>MapServer Fourth Application</title></head>
003   <body>
004     <form method=GET action="/cgi-bin/mapserv">
005       <input type="submit" value="Click Me">
006       <input type="hidden" name="program"
007         value="mapserv">
008       <input type="hidden" name="map"
009         value="/home/mapdata/fourth.map">
010       <input type="hidden" name="zoomsize
011         size=2 value=2>
012       <input type="hidden" name="map_Cities_tolerance
013         size=4 value=100>
014       <input type="hidden" name="map_Countries_tolerance
015         size=4 value=100>
016       <input type="hidden" name="layers"
017         value="Cities Countries">
018       <input type="hidden" name="slayer"
019         value="Countries">
020       <input type="hidden" name="mapshape" value="">
021       <input type="hidden" name="imgshape" value="">
022       <input type="hidden" name="imgbox" value="">
023       <input type="hidden" name="qstring" value="">
024       <input type="hidden" name="qlayer" value="">
025       <input type="hidden" name="qitem" value="">
026       <input type="hidden" name="shapeindex" value="">
027       <input type="hidden" name="savequery" value="">
028     </form>
029   </body>
030 </html>

```

Listing 5-3. *The template file fourth.html*

```

001 <html><!-- fourth.html -->
002 <head><title>MapServer Fourth Application</title>
003 <script language="JavaScript" type="text/javascript">
004 <!--
005 function setMode()
006 // set map mode to the previous mode
007 {   document.the_map.mode.selectedIndex=0;
008     for (i=0;i<document.the_map.mode.length;i++){
009       if (document.the_map.mode[i].value ==
010         document.the_map.previousmode.value){
011         document.the_map.mode.selectedIndex=i;
012       }
013     }

```

[illegible]

```

061         <option value="browse">
062         Browse</option>
063         <option value="query">
064         Query</option>
065         <option value="nquery">
066         Nquery</option>
067         <option value="itemquery">
068         Itemquery</option>
069         <option value="itemnquery">
070         Itemnquery</option>
071         <option value="featurequery">
072         Featurequery</option>
073         <option value="featurenquery">
074         Featurenquery</option>
075         <option value="itemfeaturequery">
076         Itemfeaturequery</option>
077         <option value="itemfeaturenquery">
078         Itemfeaturenquery</option>
079         <option value="indexquery">
080         Indexquery</option>
081     </select></td>
082     <td align="right">
083         Query layer:</td>
084     <td align="left">
085         <input type="text" name="qlayer" size=10
086             value="[qlayer]"></td>
087     <td align="right">
088         imgbox coords:</td>
089     <td align="left">
090         <input type="text" name="imgbox" size=25
091             value="[imgbox]"></td></tr>
092 <tr>
093     <td></td>
094     <td></td>
095     <td align="right">
096         Query item:</td>
097     <td align="left">
098         <input type="text" name="qitem" size=10
099             value="[qitem]"></td>
100     <td align="right">
101         imgshape coords:</td>
102     <td align="left">
103         <input type="text" name="imgshape" size=25
104             value="[imgshape]"></td></tr>
105 <tr>
106     <td></td>
107     <td></td>

```

```

108         <td align="right">
109             Query string:</td>
110         <td align="left">
111             <input type="text" name="qstring" size=25
112                 value=[qstring]></td>
113         <td align="right">
114             mapshape coords:</td>
115         <td align="left">
116             <input type="text" name="mapshape" size=25
117                 value="[mapshape]"></td></tr>
118     <tr>
119         <td></td>
120         <td></td>
121         <td align="right">
122             Cities Tolerance:</td>
123         <td align="left">
124             <input type="text"
125                 name=map_Cities_tolerance size=4
126                 value=[map_Cities_tolerance]>
127                 miles</td>
128         <td align="right">
129             Shape index:</td>
130         <td align="left">
131             <input type="text" name="shapeindex" size=10
132                 value=[shapeindex]></td></tr>
133     <tr>
134         <td></td>
135         <td></td>
136         <td align="right">
137             Countries Tolerance:</td>
138         <td align="left">
139             <input type="text"
140                 name=map_Countries_tolerance size=4
141                 value=[map_Countries_tolerance]>
142                 miles</td>
143         <td align="center" colspan=2 bgcolor="#F5A5A5">
144             Feature query select layer:
145             <em>Countries</em></td></tr>
146     </table>
147 </td></tr>
148 </table>
149 <input type="hidden" name="imgxy" value="320 160">
150 <input type="hidden" name="imgext" value="[mapext]">
151 <input type="hidden" name="map" value="[map]">
152 <input type="hidden" name="program" value="[program]">
153 <input type="hidden" name="slayer" value="Countries">
154 <input type="hidden" name="previousmode" value="[previousmode]">

```

```

155     <input type="hidden" name="savequery" value="true">
156 </form>
157 </body>
158 </html>

```

Listing 5-4. *The map-level query template fourth_web_header.html*

```

001 <html>
002 <head><title>MapServer - Fourth Application</title></head>
003 <body bgcolor="#E6E6E6">
004     <font size="4">Query Results</font>
005     <table border="1">
006         <tr><td>Search extent:</td>
007             <td>[imgext]<br></td></tr>
008         <tr>
009             <td>Click Point:</td>
010             <td>[img.x], [img.y] (Image coordinates)<br>
011                 [mapx], [mapy] (Map coordinates)</td>
012         </tr>
013         <tr>
014             <td>Query Results:</td>
015             <td>Layers with results=[nl]<br>
016                 Total number of results=[nr]</td>
017         </tr>
018     </table>

```

Listing 5-5. *The map-level query template fourth_web_footer.html*

```

001 <a href="http://localhost/cgi-bin/mapserv?
002 program=mapserv&map=%2Fvar%2Fwww%2Fhtdocs%2Ffourth.map&
003 zoomsize=[zoomsize]&
004 map_Cities_tolerance=[map_Cities_tolerance]&
005 map_Countries_tolerance=[map_Countries_tolerance]&
006 layers=[layers_esc]&
007 slayer=Countries&
008 mapshape=[mapshape_esc]&
009 imgshape=[imgshape_esc]&
010 imgbox=[imgbox_esc]&
011 qstring=[qstring]&
012 qlayer=[qlayer]&
013 qitem=[qitem]&
014 shapeindex=[shapeindex]&
015 savequery=[savequery]&
016 previousmode=[mode]&
017 mode=browse">
018 Return to Query Definition</a>
019 </body>
020 </html>

```


Listing 5-9. *The layer-level query template fourth_countries_footer.html*

```
</table>
```

Listing 5-10. *The layer-level query template fourth_cities_header.html*

```
001 <font size+1><b>Layer: cities</b></font>
002 <table border=1>
003 <tr bgcolor=#CCCCC><td bgcolor=#ffffff>&nbsp;</td>
004 <th>shpid</th>
005 <th>CITY</th>
006 <th>STATE</th>
007 <th>COUNTRY</th>
008 <th>LAT</th>
009 <th>LONG</th>
010 <th>POPULATION(from join)</th></tr>
```

Listing 5-11. *The class-level query template fourth_cities_query.html*

```
001 <tr><td>[l<br>
002 <td>[shpid</td>
003 <td>[CITY</td>
004 <td>[STATE</td>
005 <td>[COUNTRY</td>
006 <td>[LAT</td>
007 <td>[LONG</td>
008 <td>[test-join_POP]</td></tr>
```

Listing 5-12. *The layer-level query template fourth_cities_footer.html*

```
</table>
```



Using Perl MapScript

Thus far, all the mapping applications you've considered have been based on MapServer operating in CGI mode. While these examples have demonstrated some of MapServer's powerful features, they've been limited to displaying or querying only information that could be found in its spatial datasets. If you wanted to query an external database based on the result of some spatial query, you were out of luck—MapServer won't do that. In addition, the web-based user interface precludes the creation of smart applications that can respond more flexibly to user inputs than the *click*, *process click*, *draw map* sequence of operations that constitutes MapServer's repertoire. MapServer responses are limited to presenting maps and displaying query results by means of templates.

However, there's another, more powerful way to use MapServer. This method is called *MapScript*. MapScript provides access to all of MapServer's underlying functionality, making it available as a convenient API (application program interface). This is just a fancy way of saying that all the procedures and function calls that the MapServer executable makes to its supporting libraries are also available to other programs.

MapScript lives in the object-oriented world, so the API is more properly characterized as a collection of classes with methods and attributes. And since different languages have different structures and syntax, the API exhibits some language-specific differences. In addition to this, there are two parallel maintenance efforts. PHP MapScript is maintained manually, while the Perl and Python versions make use of a software interface generator to automate the process.

Note If you're unfamiliar with Perl's object-oriented features, consider picking up a copy of *Beginning Perl, Second Edition*, by James Lee and Simon Cozens. (Apress, 2004).

There are three versions of the API that you'll be looking at in this book. This chapter covers Perl MapScript, while the next two chapters will cover Python MapScript and PHP/MapScript, respectively. The material presented here won't be an exhaustive review of MapScript—that topic alone is worthy of an entire book. However, you'll be given the tools to build some working MapScript applications—and with those demonstrations under your belt, you'll have the background to dig into the details of the API and broaden your understanding of MapScript.

Perl is a programming language created by Larry Wall in the late 1980s. It was initially used to automate system administration tasks. An interpreted language with powerful string-handling

functions, Perl provides an excellent environment for performing these tasks. It has become, however, more than a simple scripting language for parsing log files and generating reports. It's widely used in the online environment to support interactive websites (via CGI) and build custom network applications. Modules exist that provide database interfaces to Oracle, DB2, and MySQL, and there are modules that provide numerical analysis routines and even the ability to manipulate MP3 files. Perl is open source and freely available for download and use.

Building and Installing Perl MapScript

Although the Perl MapScript code is supplied with the MapServer distribution, it's not compiled by default when MapServer is built. Building MapScript is easy—but before proceeding to that step, I'll provide a brief description of the Perl installation procedure (in the unlikely event that you don't have a functioning Perl interpreter).

Building Perl

Perl is available in a source code distribution and a variety of platform-specific precompiled binaries. The installation described here will compile and install from the source distribution. If you prefer (or are compelled) to use one of the binary distributions, consult the documentation provided with the download for installation instructions. There are many configuration options available (e.g., support for threads and large files, 64-bit support, etc.)—if you need this kind of functionality, read the `INSTALL` document. The installation described here will use the default options, which in most cases will suffice.

Retrieve the source distribution from www.perl.com and untar it into `/usr/local/src/`. Then execute the following command:

```
cd /usr/local/src/perl-5.8.X
```

5.8.X represents the version number of the distribution you downloaded. Execute the following commands:

```
./Configure -de
make
make test
```

The configuration option `-de` causes configuration defaults to be used for all options. Note that unlike previous configuration script names, this one begins with an uppercase C. You can see a list of available options by running `./Configure -h`. Some platform-specific scripts can be found in the `hints/` directory—but read the file `README.hints`. If the contents mean nothing to you, stick with the defaults. If `make test` completes successfully and displays the message `All tests successful`, execute the following commands:

```
make install
perl -V
```

This installs libraries and displays some detailed information about your configuration. The last step isn't required, but will show you where everything is installed. With a functioning Perl interpreter, you can go on to build Perl MapScript.

Building Perl MapScript

Change directories to the root of the MapServer source tree (`/usr/local/src/` in the development environment), then execute the following commands:

```
cd mapscript/perl
perl Makefile.PL
make
make install
```

That's all there is to the installation process. Assuming all has gone well, proceed to the next section.

The Perl MapScript “Hello World” Application

The first MapScript application will be kept as simple as possible. It will replicate the functionality (if you can call it that) of the “Hello World” MapServer application. This accomplishes two goals: it tests the MapScript build and it demonstrates the key steps in creating an image and displaying it in a browser. The functionality will be duplicated by means of a simple trick—instead of building a map from scratch, you'll use all the specifications in the `hello.map` file.

Note The previous applications have all used MapServer operating in CGI mode. In this chapter, you'll create a CGI script (written in Perl) to provide the same functionality via the Perl MapScript API. However, MapScript isn't required to run in CGI mode, and it can be used for far more than Web-enabled mapping applications.

When MapScript is instructed to create a map object by reading a mapfile, all the map parameters, layers, classes, and attributes are translated into MapScript objects. Default values are chosen for attributes that are left unspecified. If the mapfile produces a map as part of a CGI application, then the MapScript version will produce the same map. In addition, all the MapScript objects can be modified programmatically. By building an application this way, you can focus attention on the new MapScript functions without getting bogged down in mapfile details.

The code for this section is contained in the file `perlms_hello.pl` in the code distribution available from the Apress website, and is reproduced in Listing 6-1. The first two lines should be familiar to any Perl programmer—they invoke the Perl interpreter (the path to Perl might differ in your environment) and keep you honest by using the `strict` package. The `strict` package requires, among other things, that all variable names be given a scope using the function `my()`. Perl will choke at compilation time if you try to use a variable without a scope while using `strict`.

```
001 #!/usr/bin/perl
002 use strict;
```

Line 003 makes the MapScript module available to the script, and Line 004 makes the CGI module available. The CGI module provides several functions that are useful in a web application, such as access to form variables and creation of HTML tags. Here, it's the latter functionality you'll make use of. Although it's overkill in this application, it will be invaluable in the next. Invoking CGI with `":cgi"` forces the use of more object-oriented syntax and provides a small increment in performance by limiting the number of named objects passed to the script when CGI is first loaded. Line 005 creates a new CGI object referenced by `$resp`.

```
003 use mapscript;
004 use CGI ":cgi";
005 my $resp = new CGI;
```

Line 008 creates a unique file name for the map image by concatenating the string `perlms_hello`, a six-character string of random digits, and the image file extension `.png`. For instance, this might produce the name `perlms_hello135246.png`.

```
008 my $image_name = sprintf("ms-hello%0.6d",rand(1000000)).".png";
```

Line 011 creates a reference to a new map object, `$map`, by importing map specifications from the mapfile `hello.map` in `/home/mapdata/`. `$map` now possesses all the characteristics of the map specified in the mapfile, but there's not yet an image to display. Line 014 uses one of the constructor methods associated with the `imageObj` class to create the image, and returns a reference to it in `$img`. Line 015 uses the `imageObj` method `save()` to write the image to the appropriate place on disk.

```
011 my $map = new mapscript::mapObj("/home/mapdata/hello.map");
014 my $img = $map->draw();
015 $img->save("/var/www/htdocs/tmp/".$image_name);
```

Lines 018 through 025 generate the HTML tags needed to display your map image. Lines 018 and 019 write the header (i.e., the content type and a blank line) and opening HTML tags (`<html>`, `<header>`, `<title>`, and `<body>`) via the CGI methods `header()` and `start_html()`. A *here-document* is used to generate a form starting at Line 020. Line 021 opens a `<form>` tag, which identifies the action as this script. Line 022 creates an input field of the image type, with `src` pointing to the image just created. It's an input field, so you can click on it to execute the script again. Line 023 closes the `<form>` tag. Line 024 contains the string that terminates the here-document. Line 025, which makes use of the CGI method `end_html()`, closes all the tags opened by `start_html()`.

```
018 print $resp->header();
019 print $resp->start_html(-title=>'MapScript Hello World ');
020 print <<END_OF_HTML;
021 <form name="pointmap" action="perlms_hello.pl" method="POST">
022 <input type="image" name="img" src="/tmp/$image_name">
023 </form>
024 END_OF_HTML
025 print $resp->end_html();
```

All the HTML generated is sent to the browser, where the “Hello World” image is displayed with a tiny red dot between the words “Hello” and “World.” That’s all there is to it. Load the URL of this script (http://localhost/cgi-bin/perlms_hello.pl in the development environment) and execute it. You should see the same yellowish rectangle (with the words “Hello World” printed at the center) as you saw when you loaded `hello.html`. See Figure 6-1.

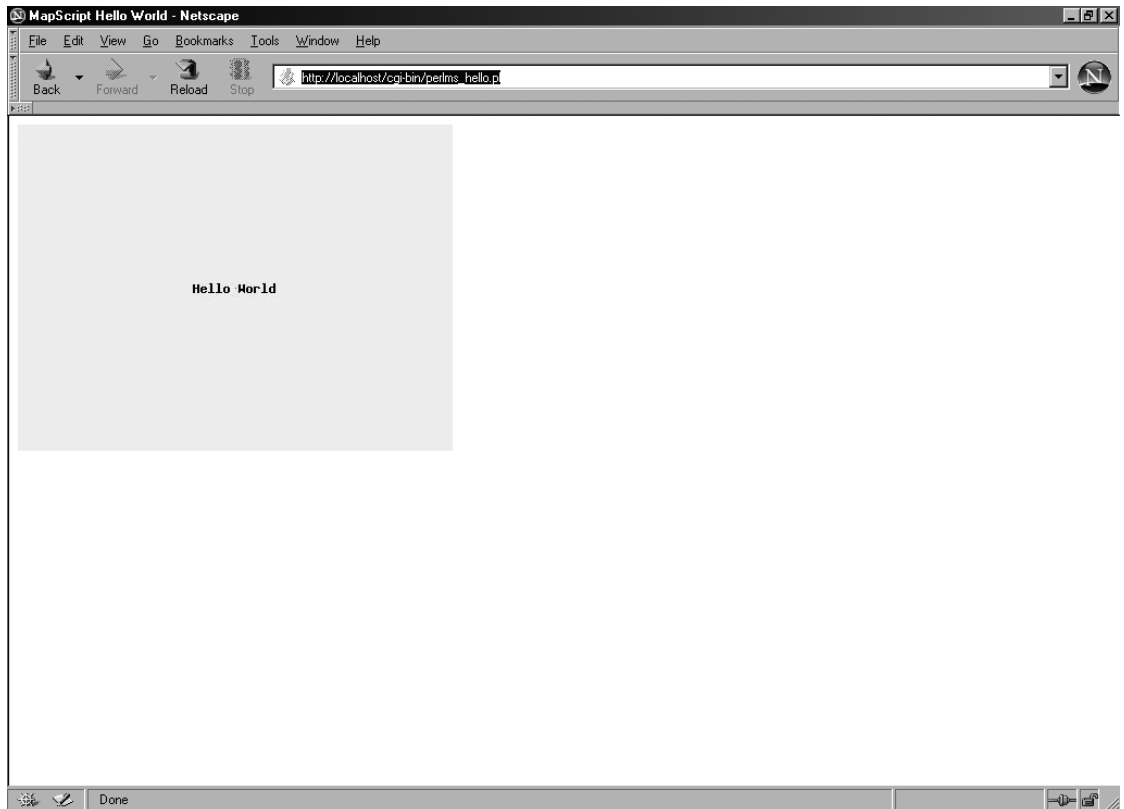


Figure 6-1. *The Perl MapScript version of the “Hello World” application*

If the expected outcome doesn’t occur, start the troubleshooting process by checking the Apache error log to determine the cause of the failure. An error message like

```
[Sat Jan 1 01:00:00 2005] [error] [client localhost]
file permissions deny server execution: /var/www/cgi-bin/perlms_hello.pl
```

indicates that the script can’t be executed because it doesn’t have the correct permissions (usually, the permission should be set to 755 for Unix-like operating systems). The following error message occurs when the script can’t find the mapfile:

```
Can't call method "draw" without a package or object reference at
/var/www/cgi-bin/perlms_hello.pl line 14.
[Fri Jun 3 02:33:36 2005] [error] [client 142.161.105.145]
Premature end of script headers: /var/www/cgi-bin/perlms_hello.pl
```

Next, check the script for typos—saving a map image under one name (or directory) and trying to access it under another name will surely fail. The MapScript environment *isn't* the same as the MapServer environment. Finally, try rebuilding MapScript and watching the build closely for any error messages.

Listing 6-1. *The Perl MapScript “Hello World” application*

```
001 #!/usr/bin/perl
002 use strict;
003 use mapscript;
004 use CGI ":cgi";
005 my $resp = new CGI;
006 # Create a unique image name every time through
007 #
008 my $image_name = sprintf("ms-hello%0.6d",rand(1000000)).".png";
009 # Create a new instance of a map object
010 #
011 my $map = new mapscript::mapObj("/home/mapdata/hello.map");
012 # Create an image of the map and save it to disk
013 #
014 my $img = $map->draw();
015 $img->save("/var/www/htdocs/tmp/".$image_name);
016 # Output the HTML form and map image
017 #
018 print $resp->header();
019 print $resp->start_html(-title=>'MapScript Hello World ');
020 print <<END_OF_HTML;
021 <form name="pointmap" action="perlms_hello.pl" method="POST">
022 <input type="image" name="img" src="/tmp/$image_name">
023 </form>
024 END_OF_HTML
025 print $resp->end_html();
```

A Practical Perl MapScript Application

The application in the previous section confirms that MapScript was built properly and demonstrates some fundamental MapScript functions, but it doesn't provide a very useful or interesting map. In this section, you'll explore MapScript functionality in greater depth and produce an interactive map. You'll employ the same method as last time—that is, the initial definition of map parameters will be taken from a mapfile, but subsequently you'll make changes to some of these parameters so that the application behaves the same way as its CGI-based counterpart. The output of the script is shown in Figure 6-2.

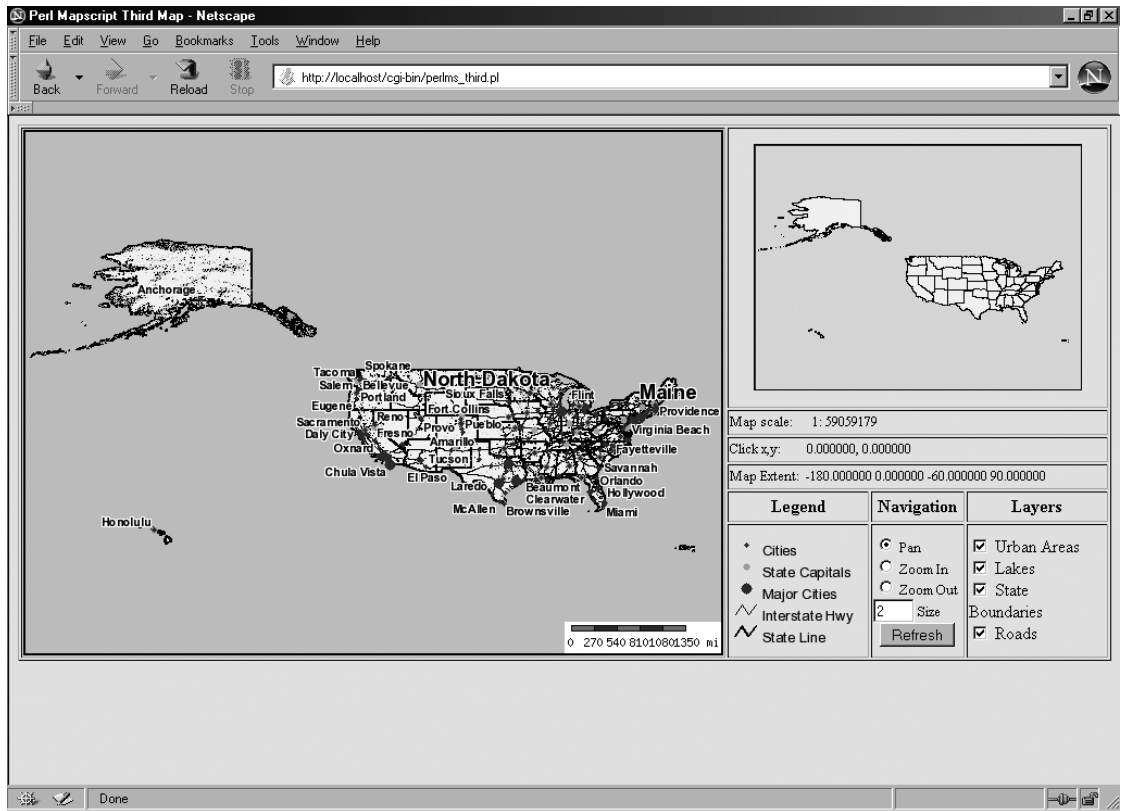


Figure 6-2. The Perl MapScript version of the third application, *perlms_third.pl*

The mapfile used will be *third.map*. The code for this example is found in the file *perlms_third.pl* and is available in the source distribution downloadable from the Apress website. The code is shown in Listing 6-2.

This script begins the same way as the previous example—that is, with the *strict*, *MapScript*, and *CGI* modules loaded. Then there's a section that sets up some variables and objects with default values. The web page generated by this script will invoke the script—it's identified in Line 007 so you don't lose track of what's executing through *perlms_third.pl*. Next, Lines 009 through 011 identify the path to the mapfile, the mapfile itself, and the path to any images created by this script.

```
001 #!/usr/bin/perl
002 use strict;
003 use MapScript;
004 use CGI ":cgi";
007 my $script_name = "/cgi-bin/perlms_third.pl";
009 my $map_path = "/home/mapdata/";
010 my $map_file = "third.map";
011 my $img_path = "/var/www/htdocs/tmp/";
```


Initially, the script is invoked from the Location bar of the browser. This means that a form to pass CGI parameter values to the script doesn't yet exist. You have to set default values for these parameters so that MapScript knows what to do the first time the script is invoked. This is done in Lines 016 through 033.

The navigation defaults in Lines 013 through 016 set the initial zoomsize and the values of the select variables \$pan, \$zoomin, and \$zoomout. Recall that an input variable of type RADIO can have several values associated with it. Only one of these can be selected at a time. If the state of a value is CHECKED, then that value is returned. You initialize the navigation radio buttons so that the first time the user sees the web page containing the map, the map will be in Pan mode. The other values are set to empty strings.

```
013 my $zoomsize=2;
014 my $pan="CHECKED";
015 my $zoomout="";
016 my $zoomin="";
```

Lines 018 through 021 set the CHECKED state for several layers. HTML will be generated that will allow the user to select which layers should be displayed. In this case, the variable will be of type CHECKBOX—which is similar to a radio button, except it allows either none, some, or all of the layers to be selected.

```
018 my $urbanareas = "CHECKED";
019 my $lakes = "CHECKED";
020 my $states = "CHECKED";
021 my $roads = "CHECKED";
```

When the user clicks some point in the map image, the pixel coordinates of that mouse click are returned to the script. However, should the user click the submit button to refresh the image, a *virtual* click point needs to be created so that MapScript has some point of reference when it zooms in or out when refreshed. Lines 022 and 023 place this point at the center of the image. (Recall from the mapfile third.map that the image is 640 pixels wide by 480 pixels high.)

```
022 my $clickx = 320;
023 my $clicky = 240;
```

Next, Lines 024 and 025 define two MapScript objects. One is pointObj(), which contains a pair of coordinates—a new pointObj() is created with the reference \$clkpoint. You'll use this object to refer to the click point (real or virtual) and assign values to its coordinates later. The other object is a rectangle object, rectObj(). A rectangle object consists of two coordinate pairs: the coordinates of the lower-left and upper-right corners of the rectangle. A rectObj() object is MapScript's way of referencing a map extent. \$old_extent refers to the extent of the map that has already been displayed in the browser (or in the case of the first invocation, it refers to the default extent).

```
024 my $clkpoint = new mapscript::pointObj();
025 my $old_extent = new mapscript::rectObj();
```

Line 026 defines the default extent as an array. The values of the array elements are the extent coordinates specified in the mapfile. On first invocation, this extent will be saved on the web page as a hidden variable. Subsequent invocations will assign current values to the coordinates.

Line 027 defines the maximum extent of the map. The MapScript `zoomPoint()` method (employed in the following code snippet) won't zoom out farther than this. This extent should also equal the extent specified in the mapfile, or else strange behavior occurs.

```
026 my @extent = (-180, 0, -60, 90);
027 my $max_extent = new mapscript::rectObj(-180, 0, -60, 90);
```

Line 030 creates a CGI object referenced by `$parms`. As mentioned, loading the CGI module with the modifier `":cgi"` requires you to use formal object-oriented syntax when using any CGI objects.

Line 033 creates a new `mapObj` based on the contents of the mapfile specified previously (i.e., `third.map`). The extent of this map is the extent specified in the mapfile, and the layers rendered are those for which the STATUS is on or default.

```
030 my $parms = new CGI;
033 my $map = new mapscript::mapObj($map_path.$map_file);
```

Line 036 determines whether the script has been invoked by a form or not. All the code up to this point is executed every time the script runs, and default values have been assigned to most variables. If the script has been invoked by the form, then the CGI method `param()` will return a hash of form variable names and values so the block of code following the `if` statement will be executed. Otherwise, a null value will be returned and execution will drop through to Line 125 without executing any conditional code.

```
036 if ( $parms->param() ) {
```

Assume that this is the first invocation so that the execution will drop through. Line 125 creates a unique identifier for the various images associated with this map by formatting a random number as a six-digit string. Lines 125 through 131 define file names and URLs for the map image, the reference map image, and the legend image.

```
125 my $map_id = sprintf("%0.6d",rand(1000000));
126 my $image_name = "third".$map_id.".png";
127 my $image_url="/tmp/".$image_name;
128 my $ref_name = "thirdref".$map_id.".gif";
129 my $ref_url="/tmp/".$ref_name;
130 my $leg_name = "thirdleg".$map_id.".png";
131 my $leg_url="/tmp/".$leg_name;
```

Line 134 uses the `imageObj()` constructor `draw()` to create the map image. Line 135 draws any cached labels on top of this image with the `mapObj` method `drawLabelCache()`. Finally, the map image is saved using the `imageObj` method `save()`. Lines 139 through 144 perform similar steps to create and save reference map and legend images to disk.

```
134 my $image=$map->draw();
135 $map->drawLabelCache($image);
136 $image->save($img_path.$image_name);
139 my $ref = $map->drawReferenceMap();
140 $ref->save($img_path.$ref_name);
143 my $leg = $map->drawLegend();
144 $leg->save($img_path.$leg_name);
```

The next step is superfluous the first time the script is executed. Line 147 retrieves the extent of the map just saved to disk and converts it to a space-delimited list of coordinates. The first time through, this extent is the same as the default extent defined previously. Subsequent invocations, however (after zooming and panning), will have different extents.

```
147 my $new_extent = sprintf("%3.6f", $map->{extent}->{minx})." "
148                 .sprintf("%3.6f", $map->{extent}->{miny})." "
149                 .sprintf("%3.6f", $map->{extent}->{maxx})." "
150                 .sprintf("%3.6f", $map->{extent}->{maxy});
```

The `mapObj` possesses an extent. Here, you want to access each of the four coordinates of the extent individually, so you use the chain of references (`$map->{extent}->{minx}`, for example). However, if you had wished to retrieve the extent as an instance of `rectObj`, you would have used the following syntax:

```
my $some_extent = new mapscript::rectObj();
my $some_extent = $map{extent};
```

Line 153 retrieves the map scale from the map object. Lines 156 through 159 invoke the function `img2map()` to convert the mouse-click point from image coordinates to map coordinates (which in the present case are measured in decimal degrees).

```
153 my $scale = sprintf("%10d", $map->{scale});
154 my ($mx, $my) = img2map($map->{width}, $map->{height},
155                        $clkpoint, $old_extent);
156 my $mx_str = sprintf("%3.6f", $mx);
157 my $my_str = sprintf("%3.6f", $my);
```

You've created and saved the three images that are required (map, reference map, and legend), and you've calculated the scale and the new extent, so you're now ready to generate the web page. For header and initial HTML, you'll use the facilities provided by the CGI module: `header()` and `start_html()` on Lines 162 and 163.

```
162 print $parms->header();
163 print $parms->start_html(-title=>'Perl Mapscript Third Map');
```

Since the web page had already been formatted as a template for the CGI-based MapServer application, it was easy to import that file directly into a here-document. Substitution strings in the template code were then replaced with the appropriate variable names. (Here, you're performing the same tasks manually that MapServer would perform every time it scanned the template file.) In Line 168, the string `[program]` is replaced with the variable `$script_name`. In Line 171, `[img]` is replaced with the variable `$image_url`. In Line 175, `[ref]` is replaced with `$ref_url`. Similar replacements occur for `[legend]` and `[scale]`. The new extent of the map—the string `$new_extent`—is stored as the value of the hidden variable `extent` in Line 184 and displayed in Line 185.

```

167 <body bgcolor="#E6E6E6">
168 <FORM METHOD=POST ACTION="$script_name">
171 <input name="img" type="image" src="$image_url"
175 <img SRC="$ref_url" width=300 height=225 border=1>
179 Map scale:&nbsp; &nbsp; &nbsp; &nbsp; 1:$scale</font></td></tr>
181 Click x,y:&nbsp; &nbsp; &nbsp; &nbsp; &nbsp; $mx_str, $my_str</font>
184 <input type="hidden" name="extent" value="$new_extent">
185 Map Extent:&nbsp; &nbsp; $new_extent</font></td></tr>
189 <tr><td rowspan="2"></td>

```

The values of the navigation defaults are inserted in Lines 191 through 197.

```

191 <INPUT TYPE=RADIO NAME="zoom" VALUE=0 $pan>
192 Pan<br>
193 <INPUT TYPE=RADIO NAME="zoom" VALUE=1 $zoomin>
194 Zoom In<br>
195 <INPUT TYPE=RADIO NAME="zoom" VALUE=-1 $zoomout>
196 Zoom Out<br>
197 <INPUT TYPE=TEXT NAME="zsize" VALUE="$zoomsize" SIZE=2>

```

The layer selection check boxes are set up in Lines 203 through 215.

```

203 <td align="top">
204 <input type="checkbox" name="layer"
205 value="urbanareas" $urbanareas >
206 Urban Areas<BR>
207 <input type="checkbox" name="layer"
208 value="lakes" $lakes >
209 Lakes<BR>
210 <input type="checkbox" name="layer"
211 value="states" $states >
212 State Boundaries<BR>
213 <input type="checkbox" name="layer"
214 value="roads" $roads >
215 Roads<BR></font>

```

The web page is now sent back to the browser via the Apache web server, and the user will see a page that's almost identical to the page displayed by the CGI-based MapServer application.

Now, let's assume that the user looks at the map, changes the zoom state from Pan to Zoom In, and clicks somewhere on the map image. The Apache server then receives the request from the browser and executes the script. The first 36 lines of the script are executed just as before, but now, when execution reaches the `if` statement on Line 036, the function `param()` returns a non-null value—a hash of form variable names and values. So, execution of the `if` block proceeds to Line 039.

If the user had clicked Refresh, then `param('refresh')` would return a non-null reference—but in this case, the user clicks on the map, so the image coordinates of the click point are returned. Since the name of the input field containing the map image is `img`, these coordinates are returned as the values of form variables `img.x` and `img.y`. Lines 045 through 046 save these values in the two variables `$clickx` and `$clicky`. You previously created an instance of `pointObj` and `$clkpoint`—now you use the `pointObj` method `setXY()` in Line 050 to set its coordinate values to `$clickx` and `$clicky`.

```
039   if ( $parms->param('refresh') ) {
040       $clickx = 320;
041       $clicky = 240;
042   } else {
043       $clickx = $parms->param('img.x');
044       $clicky = $parms->param('img.y');
045   }
046   $clkpoint->setXY($clickx,$clicky);
```

Line 053 retrieves the list of layers that the user chose to display by clicking the appropriate check boxes. The string variable `$layers` will contain a space-delimited list of layer names. In Line 055, the Perl string-matching function is used to search for the name `urbanareas` in the list. If it's found, then the variable `$urbanareas` is set to `CHECKED`. Remember, if you want this layer to be checked when you generate the web page again, you must set the value of `$urbanareas`. Next, the `mapObj` method `getLayerByName()` is used to retrieve a pointer (`$this_layer`) to the layer named `urbanareas`. This pointer is then used to access the status of the `urbanareas` layer and set it to on by assigning the value 1 to layer status. On the other hand, if the name `urbanareas` isn't found, then `$urbanareas` is set to the empty string and the layer status is set to off by assigning the value 0. This is repeated for the other layers.

```
053   my $layers = join(" ", $parms->param('layer'));
054   my $this_layer = 0;
055   if ($layers =~ /urbanareas/){
056       $urbanareas = "CHECKED";
057       $this_layer = $map->getLayerByName('urbanareas');
058       $this_layer->{status} = 1;
059   } else {
060       $urbanareas = "";
061       $this_layer = $map->getLayerByName('urbanareas');
062       $this_layer->{status} = 0;
063   }
```

In Line 084, the form variable `extent` is retrieved and its four components split into the array `@extent`. The elements of `@extent` are then used by the `mapObj` method `setExtent()` to set the extent of the map. Recall that when the script is executed, the extent of the map is set to the default value specified in the `mapfile`. It isn't until this point that the extent saved in the form on the previous invocation as the variable `extent` is parsed and used to set the extent to its previous value.

```

084   if ( $parms->param('extent') ) {
085       @extent = split(" ", $parms->param('extent'));
086   }
089   $map->setExtent($extent[0],$extent[1],$extent[2],$extent[3]);

```

A `rectObj()` object containing the current extent is required by the `zoomPoint()` method used in the following code. So, the elements of `@extent` are used to set the values of the components of the `rectObj` `$old_extent` in Lines 092 through 095.

```

092   $old_extent->{minx} = $extent[0];
093   $old_extent->{miny} = $extent[1];
094   $old_extent->{maxx} = $extent[2];
095   $old_extent->{maxy} = $extent[3];

```

Line 102 calculates the zoom factor to pass to the `zoomPoint()` method. The variable `$zoom_factor` is the product of the form variables `zoom` and `zsize`. Recall that `zoom` is set to 0 if `$pan` equals "CHECKED", -1 if `$zoomout` equals "CHECKED", and 1 if `$zoomin` equals "CHECKED". Lines 103 through 116 then set the values of the navigation variables that are to be saved in the form. There are a couple of things to note. Line 104 sets `$zoom_factor` to 1 if `$zoom_factor` equals 0, since the `zoomPoint()` method can't accept a zoom factor of 0. Line 117 sets `$zoomsize` to the absolute value of form variable `zsize`, just in case a user should enter a negative value. While this won't break the script, it will make it behave in the opposite manner—that is, a negative zoom size will result in a zoom-in if `Zoom Out` is selected, and a zoom-out if `Zoom In` is selected. Finally, Line 120 employs the `zoomPoint()` method to center the map on the click point and then zoom in or out according to the value of `$zoom_factor`.

```

102   my $zoom_factor = $parms->param("zoom")*$parms->param("zsize");
103   if ($zoom_factor == 0) {
104       $zoom_factor = 1;
105       $pan = "CHECKED";
106       $zoomout = "";
107       $zoomin = "";
108   } elsif ($zoom_factor < 0) {
109       $pan = "";
110       $zoomout = "CHECKED";
111       $zoomin = "";
112   } else {
113       $pan = "";
114       $zoomout = "";
115       $zoomin = "CHECKED";
116   }
117   $zoomsize = abs( $parms->param('zsize') );
120   $map->zoomPoint($zoom_factor,$clkpoint,$map->{width},
121       $map->{height},$old_extent,$max_extent);
122 }

```

You've now created the map, zoomed or panned, and changed its extent. At this point, you drop out of the block conditional on the presence of form variables, and prepare to both draw the images and generate the HTML that's to be forwarded to the browser.

The only part of this code you haven't looked at is the function `img2map()` in Lines 226 through 242. When you click on the map, the coordinates are returned in image coordinates. The click point coordinates required by the `zoomPoint()` method are image coordinates. But when viewing the map, you probably want to know the position of the click point in terms of map coordinates. There's no MapScript method to perform this calculation, so you're required to write one out.

The method is simple—the width (and height) of the map is known in both pixels (`$width` and `$height`) and map coordinates (from the extent `$ext`). Lines 229 through 232 separate the coordinates of the extent into the maximum and minimum values that are used to calculate the width and height of the extent in map units.

```
229     my $minx = $ext->{minx};
230     my $miny = $ext->{miny};
231     my $maxx = $ext->{maxx};
232     my $maxy = $ext->{maxy};
```

Line 233 ensures that you have a valid click point before proceeding with the calculation. Lines 234 and 235 retrieve the coordinates of the click point from `pointObj`, and Lines 236 and 237 calculate the number of map units per pixel.

```
233     if ($point->{x} && $point->{y}){
234         $x = $point->{x};
235         $y = $point->{y};
236         $dpp_x = ($maxx-$minx)/$width;
237         $dpp_y = ($maxy-$miny)/$height;
```

Note Map coordinates aren't restricted to decimal degrees. If you project your spatial data (which I'll discuss further later), then map coordinates will be actual distances like miles, kilometers, or feet—not angular measures.

Now, if a point is `$x` pixels from the left edge of the image, you obtain its longitude by multiplying `$x` by the number of degrees per pixel, and adding this to the longitude of the west (or left) side of the extent, as in Line 238. You do something similar for the height-to-latitude conversion, but keep in mind that row count increases downward for image coordinates, so you must calculate the degrees of latitude per pixel, multiply by the number of pixels from the top of the image, and *subtract* that number from the maximum extent (as in Line 239). Finally, the map coordinates are returned to the calling routine in Line 241.

```
238         $x = $minx + $dpp_x*$x;
239         $y = $maxy - $dpp_y*$y;
240     }
241     return ($x, $y);
```

Summary

In this chapter, you've examined some basic methods of Perl MapScript and created an application that duplicates the functionality of a MapServer CGI application. You've seen how to create a map object from a mapfile and manipulate some of its internal attributes and objects by means of MapScript methods. You've also learned how to draw and save the map and display it in an interactive web page. You haven't exhausted MapScript's capabilities, but you've created a firm foundation upon which you can build larger, more complicated applications that exercise more of MapScript's talents.

The next chapter will be devoted to creating the same application based on Python rather than Perl. If you're familiar with Python, this parallel development will allow you to compare and contrast the expression of the API in the two languages, and perhaps gain a clearer understanding of both. If you don't know Python, the next chapter might encourage you to learn it.

Code Listings

Code fragments were used during the code analysis so you wouldn't have to flip between code and discussion too often. The code listings, complete and uninterrupted, are presented here.

Listing 6-2. *Perl MapScript version of the third application, perlms_third.pl*

```
001  #!/usr/bin/perl
002  use strict;
003  use mapscript;
004  use CGI ":cgi";
005  # Default values
006  #
007  my $script_name = "/cgi-bin/perlms_third.pl";
008  # path defaults
009  my $map_path = "/home/mapdata/";
010  my $map_file = "third.map";
011  my $img_path = "/var/www/htdocs/tmp/";
012  # Navigation defaults
013  my $zoomsize=2;
014  my $pan="CHECKED";
015  my $zoomout="";
016  my $zoomin="";
017  # Displayed layer defaults
018  my $urbanareas = "CHECKED";
019  my $lakes = "CHECKED";
020  my $states = "CHECKED";
021  my $roads = "CHECKED";
022  my $clickx = 320;
023  my $clicky = 240;
024  my $clkpoint = new mapscript::pointObj();
025  my $old_extent = new mapscript::rectObj();
026  my @extent = (-180, 0, -60, 90);
```



```

027 my $max_extent = new mapscript::rectObj(-180, 0, -60, 90);
028 # Get CGI parms
029 #
030 my $parms = new CGI;
031 # Retrieve mapfile and create a map from it
032 #
033 my $map = new mapscript::mapObj($map_path.$map_file);
034 # We've been invoked by the form, use form variables
035 #
036 if ( $parms->param() ) {
037     # If Refresh button clicked fake the map click
038     #
039     if ( $parms->param('refresh') ) {
040         $clickx = 320;
041         $clicky = 240;
042     } else {
043         # map was clicked, get the real coordinates
044         #
045         $clickx = $parms->param('img.x');
046         $clicky = $parms->param('img.y');
047     }
048     # Set the mouse click location (we need it to zoom)
049     #
050     $clkpoint->setXY($clickx,$clicky);
051     # Selected layers may have changed, set HTML 'checks'
052     #
053     my $layers = join(" ", $parms->param('layer'));
054     my $this_layer = 0;
055     if ($layers =~ /urbanareas/){
056         $urbanareas = "CHECKED";
057         $this_layer = $map->getLayerByName('urbanareas');
058         $this_layer->{status} = 1;
059     } else {
060         $urbanareas = "";
061         $this_layer = $map->getLayerByName('urbanareas');
062         $this_layer->{status} = 0;
063     }
064     if ($layers =~ /lakes/){
065         $lakes = "CHECKED";
066         $this_layer = $map->getLayerByName('lakes');
067         $this_layer->{status} = 1;
068     } else {
069         $lakes = "";
070         $this_layer = $map->getLayerByName('lakes');
071         $this_layer->{status} = 0;
072     }

```

```

073 if ($layers =~ /states/){
074     $states = "CHECKED";
075     $this_layer = $map->getLayerByName('states');
076     $this_layer->{status} = 1;
077 } else {
078     $states = "";
079     $this_layer = $map->getLayerByName('states');
080     $this_layer->{status} = 0;
081 }
082 # invoked by form - retrieve extent
083 #
084 if ( $parms->param('extent') ) {
085     @extent = split(" ", $parms->param('extent'));
086 }
087 # Set the map to the extent retrieved from the form
088 #
089 $map->setExtent($extent[0],$extent[1],$extent[2],$extent[3]);
090 # Save this extent as a rectObj, we need it to zoom.
091 #
092 $old_extent->{minx} = $extent[0];
093 $old_extent->{miny} = $extent[1];
094 $old_extent->{maxx} = $extent[2];
095 $old_extent->{maxy} = $extent[3];
096 # Calculate the zoom factor to pass to zoomPoint method
097 # and setup the variables for web page
098 #
099 #   zoomfactor = +/- N
100 #   if N > 0 zooms in - N < 0 zoom out - N = 0 pan
101 #
102 my $zoom_factor = $parms->param("zoom")*$parms->param("zsize");
103 if ($zoom_factor == 0) {
104     $zoom_factor = 1;
105     $pan = "CHECKED";
106     $zoomout = "";
107     $zoomin = "";
108 } elsif ($zoom_factor < 0) {
109     $pan = "";
110     $zoomout = "CHECKED";
111     $zoomin = "";
112 } else {
113     $pan = "";
114     $zoomout = "";
115     $zoomin = "CHECKED";
116 }
117 $zoomsize = abs( $parms->param('zsize') );
118 # Zoom in (or out) to clkpoint

```

```

119  #
120  $map->zoomPoint($zoom_factor,$clkpoint,$map->{width},
121                $map->{height},$old_extent,$max_extent);
122  }
123  # Set unique image names for map, reference and legend
124  #
125  my $map_id = sprintf("%0.6d",rand(1000000));
126  my $image_name = "third".$map_id.".png";
127  my $image_url="/tmp/".$image_name;
128  my $ref_name = "thirdref".$map_id.".gif";
129  my $ref_url="/tmp/".$ref_name;
130  my $leg_name = "thirdleg".$map_id.".png";
131  my $leg_url="/tmp/".$leg_name;
132  # Draw and save map image
133  #
134  my $image=$map->draw();
135  $map->drawLabelCache($image);
136  $image->save($img_path.$image_name);
137  # Draw and save reference image
138  #
139  my $ref = $map->drawReferenceMap();
140  $ref->save($img_path.$ref_name);
141  # Draw and save legend image
142  #
143  my $leg = $map->drawLegend();
144  $leg->save($img_path.$leg_name);
145  # Get new extent of map (we'll save it in a form variable)
146  #
147  my $new_extent = sprintf("%3.6f",$map->{extent}->{minx})." "
148                    .sprintf("%3.6f",$map->{extent}->{miny})." "
149                    .sprintf("%3.6f",$map->{extent}->{maxx})." "
150                    .sprintf("%3.6f",$map->{extent}->{maxy});
151  # get the scale of the image to display on the web page
152  #
153  my $scale = sprintf("%10d",$map->{scale});
154  # Convert mouse click from image coordinates to map coordinates
155  #
156  my ($mx,$my) = img2map($map->{width},$map->{height},
157                        $clkpoint,$old_extent);
158  my $mx_str = sprintf("%3.6f",$mx);
159  my $my_str = sprintf("%3.6f",$my);
160  # We're done, output the HTML form
161  #
162  print $parms->header();
163  print $parms->start_html(-title=>'Perl Mapscript Third Map');
164  print <<EOF;
165  <html>

```

[illegible]

```

213         <input type="checkbox" name="layer"
214             value="roads" $roads >
215             Roads<BR></font>
216     </td>
217 </tr>
218 </table>
219 </form>
220 </body>
221 </html>
222 EOF
223 #####
224 # Convert coordinates image to map
225 #
226 sub img2map {
227     my ($width, $height, $point, $ext) = @_;
228     my ($x, $y, $dpp_x, $dpp_y) = (0,0,0,0);
229     my $minx = $ext->{minx};
230     my $miny = $ext->{miny};
231     my $maxx = $ext->{maxx};
232     my $maxy = $ext->{maxy};
233     if ($point->{x} && $point->{y}){
234         $x = $point->{x};
235         $y = $point->{y};
236         $dpp_x = ($maxx-$minx)/$width;
237         $dpp_y = ($maxy-$miny)/$height;
238         $x = $minx + $dpp_x*$x;
239         $y = $maxy - $dpp_y*$y;
240     }
241     return ($x, $y);
242 }

```



Using Python MapScript

As noted in the introduction to Chapter 6, “Using Perl MapScript,” the mapping applications described in the first five chapters have all used the MapServer executable as a CGI script. I also described the limitations of this strategy (including lack of flexibility and extensibility) and the benefits of using MapScript (including a richer user interface and access to external functionality like MySQL). The MapScript interface has been ported to several languages, one of which is Python. In this chapter, I’ll describe some of the features of Python MapScript and how to use them.

Python is an interpreted scripting language, often used for the same purposes as Perl, such as system administration and CGI web development. Python provides excellent object-oriented (OO) support and has a clean, elegant syntax that produces very readable code. Python code is portable, and Python interpreters exist for most platforms.

Python MapScript makes full use of object-oriented syntax and constructs, so a familiarity with the concepts is crucial.

Note If you’re not familiar with Python’s object-oriented features, consider picking up a copy of *Beginning Python: From Novice to Professional*, by Magnus Lie Hetland (Apress, 2005).

The material presented here isn’t an exhaustive review of Python MapScript, but the examples described will provide a solid foundation for further learning.

Building and Installing Python MapScript

The Python MapScript module supplied with the MapServer distribution isn’t automatically compiled when MapServer is built. The build process is straightforward—however, I’ll describe the Python installation first, in case your environment doesn’t include a working Python interpreter.

Building Python

There are several platform-specific binary distributions of Python. If you prefer not to (or can’t) build Python from source, choose one of these and consult the documentation that accompanies it for installation instructions. The description provided here is devoted to installing from

the source distribution. Although there are numerous configuration options available that allow you to determine the installation path, whether to include thread support, and other environment-specific details, I'll describe the default installation. A complete description of the options is given in the README file.

Download the source distribution from www.python.org/download, untar it into `/usr/local/src/`, and execute the following command:

```
cd /usr/local/src/Python-2.4.X
```

2.4.X is the version number of the distribution. If you've previously configured the build of this distribution using different options, execute the following command to remove those results:

```
rm -f configure.sh Policy.sh
```

Execute the following commands to configure and build the Python library:

```
./configure
make
make test
```

If no configuration options are specified, default options will be chosen for the build. You can display a list of options by specifying `./configure '-h` or `./configure '-help=recursive` (the latter provides more detail). It's normal for the test script to skip tests that aren't relevant to a particular environment. If `make test` terminates with the following messages:

```
257 tests OK.
33 tests skipped.
. . .
Those skips are all appropriate on environment
```

then the build was successful. (Keep in mind that the number of tests and skips may differ in your environment.)

Execute the following commands:

```
make install
python -V
```

This installs the Python libraries under `/usr/local/`. The last step merely displays the version number in order to confirm that the install placed the libraries in the expected location. You can now proceed to install Python MapScript.

Building Python MapScript

There are two methods for building Python MapScript—one is easy, the other not so easy. Since the guiding philosophy of this book is to keep things simple, the easy method is described here.

Change directory to the root of the Python MapScript source tree (`/usr/src/local/mapserver-4.x.y/mapsript/python/` in the development environment). If you're using a version of Python earlier than 2.2, omit the next step; otherwise execute the following commands:

```
cp modern/mapsript.py ./
cp modern/mapsript_wrap.c ./
```

This overwrites the old files with the more modern versions. Then execute the following to create the module:

```
python setup.py build
```

The Python MapScript distribution contains a test suite in the directory `mapserver-4.x.y/mapsript/python/tests/`. It's wise to run the test before installing the MapScript module. You do this by executing the following commands from the Python build directory:

```
cd tests/cases
python runalltests.py -V
```

If the test script terminates with an OK, you're ready to install the module.

The installation is as easy as the build and test—just execute the following command:

```
python setup.py install
```

The Python MapScript “Hello World” Application

The first MapScript application will be kept as simple as possible, simply replicating the functionality of the “Hello World” MapServer application. This will do two things: it will test the MapScript build just completed, and it will demonstrate the key steps in creating an image and displaying it in a browser. The functionality will be duplicated by means of a simple trick—instead of building a map from scratch, you'll use the specifications in the `hello.map` file.

When MapScript is instructed to create a map object by reading a mapfile, all the map parameters, layers, classes, and attributes are translated into MapScript objects. Default values are chosen for attributes that are left unspecified. If the mapfile produces a map as part of a CGI application, the MapScript version will produce the same map. In addition to this, all the MapScript objects can be modified programmatically. By building an application this way, you can focus your attention on the new MapScript functions without getting mired in mapfile details.

The code for this section is contained in the file `pythonms_hello.py` in the code distribution available from the Apress website. It's shown in Listing 7-1. In the code snippet that follows, Line 002 loads the Python MapScript module and Line 003 loads the `random` module.

```
001 #!/usr/bin/python
002 import mapscript
003 import random
```

Lines 006 and 007 define the path to the mapfile.

```
006 map_path = "/home/mapdata/"
007 map_file = "hello.map"
```

In Lines 010 through 012, one of the methods of the `random` module (`randrange()`) is used to generate a unique image file name every time that the CGI script is invoked. This is accomplished by concatenating the string `pythonms_hello` with a six-digit, zero-filled random integer and the string `.png`. In creating this image name, you could have given it any file extension—however, the mapfile will create a PNG image, and if the extension doesn't conform with the image format, the browser might become confused.


```

010 image_name = "pythonms_hello" \
011             + str(random.randrange(999999)).zfill(6) \
012             + ".png"

```

Line 015 uses the MapScript constructor method `mapObj()` to create a new map object, referenced by the variable `map`, by importing the map specifications from the `hello.map` mapfile found under the directory specified by `map_path`. `map` now possesses all the characteristics of the map specified in the mapfile, but there isn't yet an image to display. Line 018 uses one of the constructor methods associated with `imageObj` to create the image, and returns a reference to it in `img`. Finally, Line 019 uses the `imageObj` method `save()` to write the image to the appropriate place on disk.

```

015 map = mapscript.mapObj(map_path+map_file)
018 img=map.draw()
019 img.save("/var/www/htdocs/tmp/" + image_name)

```

Lines 022 through 033 generate the HTML tags needed to display your map image. Lines 022 through 026 write the header (i.e., the content type and a blank line) and the opening HTML tags (`<html>`, `<header>`, `<title>`, and `<body>`). Lines 027 through 031 define a multiline string (like a *here-document* in Perl) that produces an HTML form. Line 028 opens a `<form>` tag, which identifies the action as this script. Line 029 creates an input field of type `image`, with `src` pointing to the conversion specifier (`%s`) that will contain the name of the image you just created. It's an input field, so you can click on it to execute the script again. Line 030 closes the form tag. Line 031 terminates the multiline string and supplies the image name to the conversion specifier from Line 029. Lines 032 and 033 close the tags that were opened earlier.

```

022 print "Content-type: text/html"
023 print
024 print "<html>"
025 print "<header><title>Python Mapscript Hello World</title></header>"
026 print "<body>"
027 print ""
028 <form name="hello" action="pythonms_hello.py" method="POST">
029 <input type="image" name="img" src="/tmp/%s">
030 </form>
031 "" % image_name
032 print "</body>"
033 print "</html>"

```

All the HTML generated is sent to the browser, in which the “Hello World” image will be displayed with a tiny red dot between the words “Hello” and “World.” That’s all there is to it. Load the URL of this script (http://localhost/cgi-bin/pythonms_hello.py in the development environment) and execute it. This should display the image shown in Figure 7-1, in which you should see the same yellowish rectangle (with the words “Hello World” printed at the center) as you saw when you loaded `hello.html`.

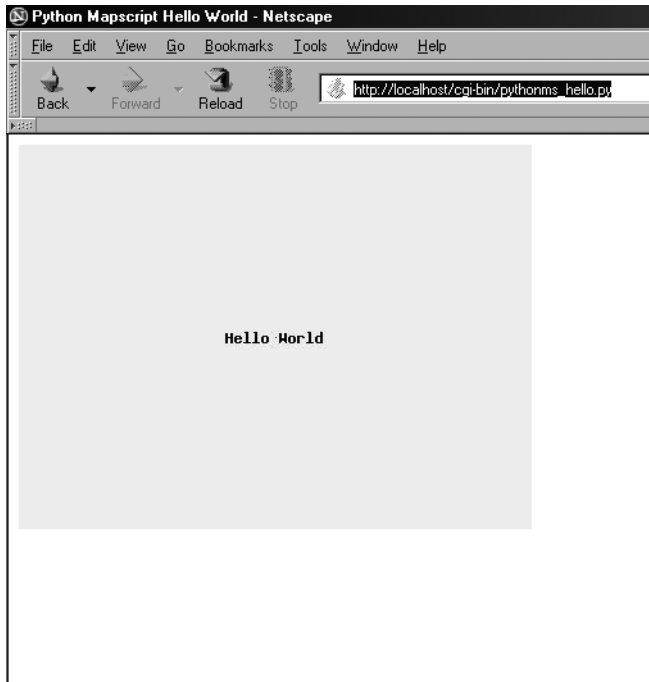


Figure 7-1. *The Python MapScript version of the “Hello World” application*

A Practical Python MapScript Application

The application in the previous section confirmed that MapScript was built properly and demonstrated some fundamental MapScript functions, but it didn’t provide a very useful or interesting map. In this section, you’ll explore MapScript functionality more deeply and produce an interactive map. You’ll employ the same method as last time—that is, the initial definition of map parameters will be taken from a mapfile—but subsequently, you’ll make changes to some of these parameters so that the application behaves the same way as its CGI-based parent. The mapfile used will be `third.map`. The code for this example is found in the file `pythonms_third.py` and is available in the source distribution downloadable from the Apress website. The code is shown in Listing 7-2.

This script begins the same way as the previous one—that is, the modules `mapscript` and `random` are loaded. In addition, the `cgi` module is loaded, the methods of which give convenient access to CGI form variables.

In the next section, Lines 010 through 027 convert the image coordinates of a point (in pixels) to geographic coordinates (these can take several forms, the most common of which are degrees and meters).

Initially, the script is invoked from the Location bar of the browser. This means that a form doesn’t yet exist to pass CGI parameter values to the script. You have to set default values for these parameters so that MapScript knows what to do the first time the script is invoked. This is done in Lines 032 through 059. The relevant code is shown in the following code snippet.

The web page generated by this script will invoke itself—it's identified in Line 026 so you don't lose track of what's executing—`pythonms_third.py`. Lines 028 through 030 identify the path to the mapfile, the mapfile itself, and the path to images created by this script.

```
026 script_name = "/cgi-bin/pythonms_third.py"
028 map_path = "/home/mapdata/"
029 map_file = "third.map"
030 img_path = "/var/www/htdocs/tmp/"
```

The navigation defaults in Lines 040 through 043 set the initial zoomsize and the values of the select variables `pan`, `zoomin`, and `zoomout`. Recall that an input variable of type `radio` can have several values associated with it. Only one of these can be selected at a time. If the state of a value is `CHECKED`, then the value that's `CHECKED` is returned. You initialize the navigation radio buttons so that the map will be in `Pan` mode the first time the user sees the web page containing the map. The other values are set to empty strings.

```
032 zoomsize=2
033 pan="CHECKED"
034 zoomout=""
035 zoomin=""
```

Lines 037 through 040 set the `CHECKED` state for several layers. HTML will be generated that allows the user to select which layers should be displayed. In this case, the input variable will be of type `CHECKBOX`, which is similar to a radio button, except that it allows either none, some, or all the layers to be selected.

```
036 # Displayed layer defaults
037 urbanareas = "CHECKED"
038 lakes = "CHECKED"
039 states = "CHECKED"
040 roads = "CHECKED"
```

When the user clicks on some point in the map image, the pixel coordinates of that mouse click are returned to the script. However, should the user use the submit button to refresh the image, a *virtual* click point needs to be created so that MapScript has some point of reference when it zooms in or out when refreshed. Lines 042 and 043 place this point at the center of the image. (Recall from the mapfile `third.map` that the image is 640 pixels wide by 480 pixels high.)

```
042 clickx = 320
043 clicky = 240
```

Next, Line 044 defines a MapScript `pointObj()` object. A `pointObj()` object contains a pair of coordinates. You'll use this object to refer to the click point (real or virtual) and assign values to its coordinates later.

```
044 clkpoint = mapscript.pointObj()
```

In Line 046, a rectangle object, `rectObj()`, is created. A rectangle object consists of two coordinate pairs: the coordinates of the lower-left and upper-right corners of the rectangle. `rectObj()` is MapScript's way of referencing a map extent. `old_extent` refers to the extent of the

map that has already been displayed in the browser. (If this is the first invocation, it will refer to the default extent.)

```
046 old_extent = mapscript.rectObj()
```

Line 047 defines the default extent as an array. The values of the array elements are the extent coordinates specified in the mapfile. On first invocation, this is the extent that will be saved on the web page as a hidden variable. Subsequent invocations will assign current values to the coordinates. Line 048 defines the maximum extent of the map—the MapScript `zoomPoint()` method employed in the following code won't zoom out farther than this. This extent should also equal the extent specified in the mapfile, or else strange behavior will occur.

```
047 extent = (-180.0, 0.0, -60.0, 90.0)
048 max_extent = mapscript.rectObj(-180.0, 0.0, -60.0, 90.0)
```

Line 051 creates a CGI object referenced by `parms`. As mentioned previously, the CGI method `FieldStorage()` is used to retrieve form values.

```
051 parms = cgi.FieldStorage()
```

Line 054 creates a new `mapObj` map based on the contents of the mapfile specified previously (i.e., `third.map`). The extent of this map is the extent specified in the mapfile, and the layers rendered are those for which the `STATUS` is on or default.

```
054 map = mapscript.mapObj(map_path+map_file)
```

Lines 057 and 058 determine whether the script has been invoked by a form or not. All the code up to this point is executed every time the script runs, and default values have been assigned to most variables. If the script has been invoked by the form, then the CGI method `parms.getfirst()` will return the value of its argument string. The block of code following the `if` statement will then be executed.

```
057 if (parms.getfirst('img.x') and parms.getfirst('img.y')) \
058     or parms.getfirst('refresh'):
```

If the script hasn't been invoked by a form, the `if` conditional will evaluate to `False` and execution will drop through to Line 142 without executing any conditional code. Let's assume that this is the first invocation so that execution falls through.

Line 142 creates a unique identifier for the various images associated with this map by formatting a random number as a six-digit string. Lines 143 through 148 define file names and URLs for the map image, the reference map image, and the legend image.

```
142 map_id = str(random.randrange(999999)).zfill(6)
143 image_name = "pythird" + map_id + ".png"
144 image_url="/tmp/" + image_name
145 ref_name = "pythirdref" + map_id + ".gif"
146 ref_url="/tmp/" + ref_name
147 leg_name = "pythirdleg" + map_id + ".png"
148 leg_url="/tmp/" + leg_name
```

Note Reference map image types can only be in GD-based formats: GIF, PNG, and JPEG. Furthermore, the image type of the output image must be the same as the input image type specified in the REFERENCE object. Reference maps also ignore the image type specified in the IMAGEFORMAT object.

Line 151 uses the `draw()` method to create the map image. Finally, the map image is saved by the `imageObj` method `save()`. Lines 155 through 160 perform similar steps for creating and saving the reference map and legend images.

```
151 image=map.draw()
152 image.save(img_path + image_name)
155 ref = map.drawReferenceMap()
156 ref.save(img_path + ref_name)
159 leg = map.drawLegend()
160 leg.save(img_path + leg_name)
```

The next step is superfluous the first time the script is executed. Lines 164 through 166 retrieve the extent of the map just saved to disk as a string with spaces separating the coordinates. The first time through, this extent is the same as the default extent defined previously. On subsequent invocations, however, the map will have different extents after the user zooms and pans. A `mapObj` possesses an extent, which consists of four coordinates. Here, you want to access each of the coordinates individually, so you use the chain of references (`map.extent.minx`, for example).

```
164 new_extent = str(map.extent.minx)+" "+str(map.extent.miny) \
165             + " " + str(map.extent.maxx) \
166             + " " + str(map.extent.maxy)
```

Line 169 retrieves the map scale from the map object. Lines 172 through 175 invoke the function `img2map()` to convert the mouse-click point from image coordinates to map coordinates (which in the present case are measured in decimal degrees).

```
169 scale = map.scale
172 clkgeo = img2map(map.width,map.height, \
173                 clkpoint.x,clkpoint.y,old_extent)
174 x_geo = clkgeo[0]
175 y_geo = clkgeo[1]
```

You've created and saved the three images that are required (map, reference map, and legend), and you've calculated the scale and the new extent, so you're now ready to generate the web page. Lines 176 through 180 move several variables to an array to allow the use of meaningful labels in the multiline string that's used to format the web page.

```
176 Mapvars= {'image_url':image_url,'ref_url':ref_url, 'scale':scale, \
177           'x_geo':x_geo, 'y_geo':y_geo, 'new_extent':new_extent, \
178           'leg_url':leg_url, 'pan':pan, 'zoomin':zoomin, \
179           'zoomout':zoomout, 'zoomsize':zoomsize, 'lakes':lakes, \
180           'states':states, 'roads':roads, 'urbanareas':urbanareas}
```

Lines 183 through 187 print the preamble and opening tags for the web page (<html>, <head>, and <body>).

```
183 print "Content-type: text/html"
184 print
185 print "<html>"
186 print "<header><title>Python Mapscript Third Map</title></header>"
187 print "<body bgcolor=\"#E6E6E6\">"
```

Since the web page had already been formatted as a template for the CGI-based MapServer application, it was easy to import that file directly into a multiline string. In order to populate this string with appropriate values, format codes are inserted where a value should be. The values can be form variables or any piece of information you'd like to place in front of the viewer. When this string is processed for output to the browser, MapScript scans it and searches for format codes. When it finds one, it replaces it with a value from the array specified on Line 243.

The web page is now sent back to the browser via the Apache web server, and the user sees a page that is almost identical to the page displayed by the CGI-based MapServer application (as shown in Figure 7-2).

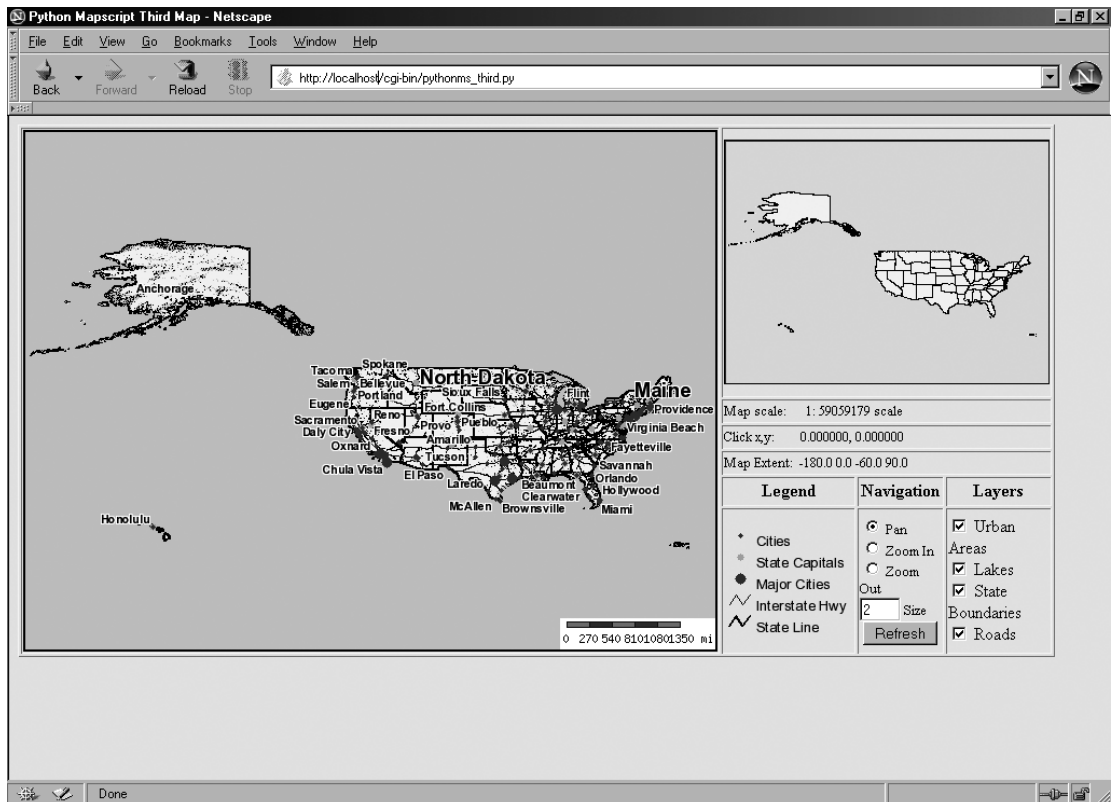


Figure 7-2. The Python MapScript version of the third application, *pythonms_third.py*

Now, let's assume that the user changes the zoom state from Pan to Zoom In and clicks somewhere on the map image. When this happens, the Apache server receives the request from the browser and executes the script. The first part of the script (up to Line 057) is executed just as before, but now, when execution reaches the if statement, the method `parms.getfirst()` returns a true value, since the form variables `img.x`, `img.y`, or `refresh` will have been returned from the browser. Therefore, execution of the if block proceeds to Line 059.

```
057 if (parms.getfirst('img.x') and parms.getfirst('img.y')) \
058     or parms.getfirst('refresh'):
059     if parms.getfirst('refresh'):# refresh, fake the coordinates
060         clickx = 320
061         clicky = 240
062     else:                        # map click, use real coordinates
063         clickx = int( parms.getfirst('img.x') )
064         clicky = int( parms.getfirst('img.y') )
```

If the user had clicked Refresh, then `parms.getfirst('refresh')` would return a true value and the code in Lines 060 and 061 would assign fake image coordinates (at the center of the image) to the variables `clickx` and `clicky`.

But in this case, you're assuming that the coordinates of the click point are returned because the user clicks on the map. Since the name of the input field containing the map image is `img`, the coordinates are returned as the values of form variables `img.x` and `img.y`. Lines 063 and 064 then save these values in `clickx` and `clicky`. The `pointObj`, `clkpoint`, was created in Line 044. Its coordinate values are now set to `clickx` and `clicky` in Lines 067 and 068.

```
067     clkpoint.x = clickx
068     clkpoint.y = clicky
```

Line 071 retrieves a list of layers that the user has chosen to display by clicking the appropriate check boxes. These layer names are concatenated into the space-delimited string, `layers`, in Line 072.

```
071     layerlist = parms.getlist('layer')
072     layers = " ".join(layerlist)
```

In Line 073, the Python string-comparison method `find()` is used to search for the string 'urbanareas' in `layers`. If it's found, then the variable `urbanareas` is set to `CHECKED`. Remember, if you want this layer to be checked when you generate the web page again, you must set the value of `urbanareas`. Next, the `mapObj` method `getLayerByName()` is used to retrieve a pointer (`this_layer`) to the layer named `urbanareas`. The pointer is then used to access the status of the layer and set it to on by assigning the value 1 to `layer status`. On the other hand, if the string 'urbanareas' isn't found, this means that the user has unchecked the `urbanareas` check box. In this case, the variable `urbanareas` is set to the empty string, and the layer status is set to off by assigning it the value 0. This is repeated for the other layers.

```
073     if layers.find('urbanareas') > -1:
074         urbanareas = "CHECKED"
075         this_layer = map.getLayerByName('urbanareas')
076         this_layer.status = 1
077     else:
```

```

078         urbanareas = ""
079         this_layer = map.getLayerByName('urbanareas')
080         this_layer.status = 0

```

In Line 099, the form variable 'extent' is retrieved and its four components split into array extent. The elements of extent are then used by the mapObj method setExtent() to set the extent of the map. Recall that when this script is executed, the extent of the map is initially set to the default value specified in the mapfile. Of course, on the previous invocation, it likely had some other value. This value would have been saved as a space-delimited string in the hidden variable extent. It's not until this point in the script that the value of this variable is retrieved, parsed, and used to set the extent to its previous value in Lines 103 and 104.

```

099         if parms.getfirst('extent'):
100             extent = parms.getfirst('extent').split(' ')
101             map.setExtent(float(extent[0]),float(extent[1]), \
102                           float(extent[2]),float(extent[3]))

```

A rectObj() object containing the current extent is required by the zoomPoint() method used in the following code—therefore, the elements of the extent array are used to set the values of the components of the rectObj old_extent in Lines 107 through 110.

```

107         old_extent.minx = float(extent[0])
108         old_extent.miny = float(extent[1])
109         old_extent.maxx = float(extent[2])
110         old_extent.maxy = float(extent[3])

```

Lines 117 and 118 calculate the zoom factor to pass to the zoomPoint() method. zoom_factor is the product of the form variables zoom and zsize. Recall that zoom is set to 0 if pan equals "CHECKED", -1 if zoomout equals "CHECKED", and 1 if zoomin equals "CHECKED". Lines 119 through 132 then set the values of the navigation variables that are to be saved in the form. There are a couple of things to note. Line 121 sets zoom_factor to 1 if zoom_factor equals 0 since the zoomPoint() method can't accept a zoom factor of 0. Line 119 sets zoomsize to the absolute value of the form variable zsize, just in case a user should enter a negative value. (A negative zsize would reverse the meaning of the Zoom In and Zoom Out radio buttons.)

```

117         zoom_factor = int(parms.getfirst('zoom')) \
118                        * int(parms.getfirst('zsize'))
119         zoomsize = str( abs( int( parms.getfirst('zsize')) ) )
120         if zoom_factor == 0:
121             zoom_factor = 1
122             pan = "CHECKED"
123             zoomout = ""
124             zoomin = ""
125         elif zoom_factor < 0:
126             pan = ""
127             zoomout = "CHECKED"
128             zoomin = ""
129         else:
130             pan = ""
131             zoomout = ""
132             zoomin = "CHECKED"

```


Finally, Lines 135 and 136 employ the `zoomPoint()` method to center the map on the click point, and then zoom in or out according to the value of `zoom_factor`.

```
135         map.zoomPoint(zoom_factor,clkpoint,map.width,      \
136                        map.height,old_extent,max_extent)
```

You've now created the map, zoomed or panned, and changed its extent. At this point, execution drops out of the block conditioned on the presence of form variables, and prepare to both draw the images and generate the HTML that's to be forwarded to the browser. You can skip this step since you've been through this part of the code before.

The only part of this code you haven't looked at is the function `img2map()` in Lines 008 through 022. When you click on the map, the coordinates are returned in image coordinates. The `zoomPoint()` method requires the coordinates of the click point to be supplied in image coordinates too. But when viewing the map, you probably want to know the position of the click point in terms of map coordinates. There's no MapScript method to perform this calculation, so you're required to write your own. Lines 009 through 012 set the initial values of some variables.

```
008 def img2map (width, height, x, y, ext):
009     x = 0
010     y = 0
011     dpp_x = 0
012     dpp_y = 0
```

Since the width (and height) of the map is known in both pixels (width and height) and map coordinates (from the function parameter passed to `ext`), and you know the click point coordinates in pixels, you can convert pixels to map coordinates using proportions. Lines 013 through 016 extract the maximum and minimum map coordinates from `ext`. Lines 017 and 018 determine the number of map units per pixel (e.g., degrees per pixel or meters per pixel).

```
013     minx = ext.minx
014     miny = ext.miny
015     maxx = ext.maxx
016     maxy = ext.maxy
017     dpp_x = (maxx-minx)/width # degrees per pixel
018     dpp_y = (maxy-miny)/height
```

Note Map coordinates aren't restricted to decimal degrees. If you project your spatial data (which I'll discuss further in the Appendix), then map coordinates will be actual distances like miles, kilometers, or feet—not angular measures.

Now, if a point is x pixels from the left edge of the image, you obtain its longitude by multiplying x by the number of degrees per pixel, and adding this to the longitude of the west (or left) side of the extent, as in Line 019. You do something similar for the height-to-latitude conversion, but keep in mind that pixel row count increases downward, so you must calculate the degrees of latitude per pixel, multiply by the number of pixels from the top of the image, and

subtract that number from the maximum extent (as in Line 020). Finally, the map coordinates are returned to the calling routine in Line 022.

```
019     x = minx + dpp_x*x           # degrees from left
020     y = maxy - dpp_y*y           # degrees from top because
022     return (x, y)
```

each layer in the group. In this application, you used the `getLayerByName()` method to retrieve a reference to a layer. Using that reference, you could then set the status of the layer. MapScript, however, doesn't possess a `*getLayersByGroupName()` method. In order to use the group name, you would have to retrieve layers with the `getLayer()` method (which uses the layer number as a reference) by searching through all layers and selecting those with the appropriate GROUP name. In order to keep the application simple, this wasn't done in the present case. If you require it in your own applications, however, it's a straightforward application of techniques already described.

Summary

In this chapter, you've examined some basic methods of Python MapScript and created an application that duplicates the functionality of a MapServer CGI application. You've seen how to create a map object from a mapfile and manipulate some of its internal attributes and objects by means of MapScript methods. You've also learned how to draw and save the map, and display it in an interactive web page. You haven't exhausted MapScript's capabilities, but you've created a firm foundation upon which you can build larger, more complicated applications that exercise more of MapScript's talents.

The next chapter will be devoted to creating the same application based on PHP. If you're familiar with PHP, this parallel development will allow you to compare and contrast the expression of the API in the three languages, and perhaps gain a clearer understanding of them all. If you don't know PHP, the next chapter might encourage you to learn it.

Code Listings

Listing 7-1. *The Python MapScript version of the “Hello World” application, `pythonms_hello.py`*

```
001 #!/usr/bin/python
002 import mapscript
003 import random
004 # path defaults
005 #
006 map_path = "/home/mapdata/"
007 map_file = "hello.map"
008 # Create a unique image name every time through
009 #
010 image_name = "pythonms_hello" \
011             + str(random.randrange(999999)).zfill(6) \
012             + ".png"
013 # Create a new instance of a map object
014 #
015 map = mapscript.mapObj(map_path+map_file)
016 # Create an image of the map and save it to disk
017 #
018 img=map.draw()
019 img.save("/var/www/htdocs/tmp/" + image_name)
```

```

020 # Output the HTML form and map image
021 #
022 print "Content-type: text/html"
023 print
024 print "<html>"
025 print "<header><title>Python Mapscript Hello World</title></header>"
026 print "<body>"
027 print ""
028 <form name="hello" action="pythonms_hello.py" method="POST">
029 <input type="image" name="img" src="/tmp/%s">
030 </form>
031 "" % image_name
032 print "</body>"
033 print "</html>"

```

Listing 7-2. *The Python MapScript version of the third application, pythonms_third.py*

```

001 #!/usr/bin/python
002 import mapscript
003 import cgi
004 import random
005 #####
006 # Convert image coordinates to map coordinates
007 #
008 def img2map (width, height, x, y, ext):
009     x = 0
010     y = 0
011     dpp_x = 0
012     dpp_y = 0
013     minx = ext.minx
014     miny = ext.miny
015     maxx = ext.maxx
016     maxy = ext.maxy
017     dpp_x = (maxx-minx)/width # degrees per pixel
018     dpp_y = (maxy-miny)/height
019     x = minx + dpp_x*x         # degrees from left
020     y = maxy - dpp_y*y         # degrees from top because
021                               # pixels count down from top
022     return (x, y)
023 #####
024 # Default values
025 #
026 script_name = "/cgi-bin/pythonms_third.py"
027 # path defaults
028 map_path = "/home/mapdata/"
029 map_file = "third.map"
030 img_path = "/var/www/htdocs/tmp/"

```

```

031 # Navigation defaults
032 zoomsize=2
033 pan="CHECKED"
034 zoomout=""
035 zoomin=""
036 # Displayed layer defaults
037 urbanareas = "CHECKED"
038 lakes = "CHECKED"
039 states = "CHECKED"
040 roads = "CHECKED"
041 # map click defaults
042 clickx = 320
043 clicky = 240
044 clkpoint = mapscript.pointObj()
045 # extent defaults
046 old_extent = mapscript.rectObj()
047 extent = (-180.0, 0.0, -60.0, 90.0)
048 max_extent = mapscript.rectObj(-180.0, 0.0, -60.0, 90.0)
049 # Get CGI parms
050 #
051 parms = cgi.FieldStorage()
052 # Retrieve mapfile and create a map from it
053 #
054 map = mapscript.mapObj(map_path+map_file)
055 # We've been invoked by the form, use form variables
056 #
057 if (parms.getfirst('img.x') and parms.getfirst('img.y')) \
058     or parms.getfirst('refresh'):
059     if parms.getfirst('refresh'):# refresh, fake the coordinates
060         clickx = 320
061         clicky = 240
062     else:                        # map click, use real coordinates
063         clickx = int( parms.getfirst('img.x') )
064         clicky = int( parms.getfirst('img.y') )
065     # Set mouse click location in pointObj (we need it to zoom)
066     #
067     clkpoint.x = clickx
068     clkpoint.y = clicky
069     # Selected layers may have changed, set HTML 'checks'
070     #
071     layerlist = parms.getlist('layer')
072     layers = " ".join(layerlist)
073     if layers.find('urbanareas') > -1:
074         urbanareas = "CHECKED"
075         this_layer = map.getLayerByName('urbanareas')
076         this_layer.status = 1

```

```

077     else:
078         urbanareas = ""
079         this_layer = map.getLayerByName('urbanareas')
080         this_layer.status = 0
081     if layers.find('lakes') > -1:
082         lakes = "CHECKED"
083         this_layer = map.getLayerByName('lakes')
084         this_layer.status = 1
085     else:
086         lakes = ""
087         this_layer = map.getLayerByName('lakes')
088         this_layer.status = 0
089     if layers.find('states') > -1:
090         states = "CHECKED"
091         this_layer = map.getLayerByName('states')
092         this_layer.status = 1
093     else:
094         states = ""
095         this_layer = map.getLayerByName('states')
096         this_layer.status = 0
097     # retrieve extent of displayed map
098     #
099     if parms.getfirst('extent'):
100         extent = parms.getfirst('extent').split(' ')
101         # Set the new map to the extent retrieved from the form
102         #
103         map.setExtent(float(extent[0]),float(extent[1]), \
104             float(extent[2]),float(extent[3]))
105         # Save this extent as a rectObj (we need it to zoom)
106         #
107         old_extent.minx = float(extent[0])
108         old_extent.miny = float(extent[1])
109         old_extent.maxx = float(extent[2])
110         old_extent.maxy = float(extent[3])
111         # Calculate the zoom factor to pass to zoomPoint method
112         # and setup the variables for web page
113         #
114         # zoomfactor = +/- N
115         # if N > 0 zooms in - N < 0 zoom out - N = 0 pan
116         #
117         zoom_factor = int(parms.getfirst('zoom')) \
118             * int(parms.getfirst('zsize'))
119         zoomsize = str( abs( int( parms.getfirst('zsize')) ) )
120         if zoom_factor == 0:
121             zoom_factor = 1
122             pan = "CHECKED"
123             zoomout = ""
124             zoomin = ""

```

```

125     elif zoom_factor < 0:
126         pan = ""
127         zoomout = "CHECKED"
128         zoomin = ""
129     else:
130         pan = ""
131         zoomout = ""
132         zoomin = "CHECKED"
133     # Zoom in (or out) to clkpoint
134     #
135     map.zoomPoint(zoom_factor,clkpoint,map.width,      \
136                  map.height,old_extent,max_extent)
137 # We've dropped thru because the script was invoked directly
138 # or we've finished panning, zooming and setting layers on or off
139 #
140 # Set unique image names for map, reference and legend
141 #
142 map_id = str(random.randrange(999999)).zfill(6)
143 image_name = "pythird" + map_id + ".png"
144 image_url="/tmp/" + image_name
145 ref_name = "pythirdref" + map_id + ".gif"
146 ref_url="/tmp/" + ref_name
147 leg_name = "pythirdleg" + map_id + ".png"
148 leg_url="/tmp/" + leg_name
149 # Draw and save map image
150 #
151 image=map.draw()
152 image.save(img_path + image_name)
153 # Draw and save reference image
154 #
155 ref = map.drawReferenceMap()
156 ref.save(img_path + ref_name)
157 # Draw and save legend image
158 #
159 leg = map.drawLegend()
160 leg.save(img_path + leg_name)
161 # Get extent of map after any zooming or panning
162 # (we'll save it in a form variable)
163 #
164 new_extent = str(map.extent.minx)+" "+str(map.extent.miny) \
165              + " " + str(map.extent.maxx) \
166              + " " + str(map.extent.maxy)
167 # get the scale of the image to display on the web page
168 #
169 scale = map.scale
170 # Convert mouse click from image coordinates to map coordinates
171 #

```

[illegible]


```

219     Zoom Out<br>
220     <input type="text" name="zsize" value="%(zoomsize)s" size=2>
221     Size<br>
222     <center>
223         <input type="submit" name="refresh" value="Refresh">
224     </center>
225     </td>
226     <td align="top">
227         <input type="checkbox" name="layer"
228             value="urbanareas" %(urbanareas)s >
229             Urban Areas<BR>
230         <input type="checkbox" name="layer"
231             value="lakes" %(lakes)s >
232             Lakes<BR>
233         <input type="checkbox" name="layer"
234             value="states" %(states)s >
235             State Boundaries<BR>
236         <input type="checkbox" name="layer"
237             value="roads" %(roads)s >
238             Roads<BR></font>
239     </td>
240 </tr>
241 </table>
242 </form>
243 """ % Mapvars
244 print "</body>"
245 print "</html>"

```



Using PHP/MapScript

As previously noted, MapServer's role as a CGI script, while powerful, is limited to a simple web user interface, and it can't be extended with external libraries that provide additional functionality. MapScript is an interface that allows access to MapServer's underlying functionality from a variety of programming environments. In earlier chapters, this interface was described for Perl and Python MapScript. Both of these derive from the same code base and are therefore very similar. In fact, there's an entire family of such MapScripts (which include, in addition to the two mentioned previously, Java, Tcl, and Ruby MapScripts) that share similar functionality. PHP/MapScript is distinguished not only orthographically from the rest, but also by its code, which is maintained independently. Therefore, its functionality may not correspond exactly with the other versions of MapScript.

This chapter will provide an introduction to PHP/MapScript that's more a primer than a comprehensive reference. But, while only a fraction of MapScript's capabilities are described, the fundamentals presented will enable you to build your own applications. PHP/MapScript's object-oriented interface is defined in terms of classes and methods, and some understanding of these techniques is required to use it effectively.

Note If you're unfamiliar with PHP or its object-oriented features, consider picking up a copy of *Beginning PHP 5 and MySQL: From Novice to Professional*, by W. Jason Gilmore (Apress, 2004).

Building and Installing PHP/MapScript

The source code for the PHP/MapScript module is supplied with the MapServer distribution, but the module isn't created along with MapServer unless you specifically request it when you configure the MapServer build. Also, since PHP 5 is sufficiently recent that you may not have it installed, I'll briefly describe the PHP build process before proceeding to PHP/MapScript.

In some environments, PHP 5 and MapScript don't work well together. In order to avoid problems later, a couple of issues need to be resolved. The first concerns how PHP itself interacts with the Apache server. The default PHP build creates an executable that's intended to function in a CGI environment—that is, it's loaded every time Apache is called upon to execute a PHP script. On the other hand, if PHP is compiled as a DSO (dynamic shared object), it becomes part of Apache and no longer needs to be reloaded for every request. The DSO option is faster,

but since MapScript is supported only as a CGI, not a DSO, this can sometimes lead to problems. In the build described, you'll create PHP as a CGI. The Apache configuration changes described will allow you to run PHP and PHP/MapScript as a CGI even if your current environment supports PHP as a DSO.

The other issue is related to an incompatibility between PHP's built-in regular expression library and the system library. Apache and MapScript use the system library, but the default PHP build uses the bundled library, which can lead to problems. However, there are two possible solutions: you can recompile Apache and MapScript using PHP's library, or you can go with the simpler alternative and specify that the build use the system regex library. To keep things simple, the build described here will do the latter.

Building PHP

There are numerous configuration options available when building PHP, and your installation will probably have requirements beyond those needed to use the application code described in this book. However, for simplicity's sake, you'll perform a build that specifies only the functionality you need.

Download the PHP distribution (available at www.php.net) and untar it into `/usr/src/`. Then, change to the PHP source directory and run `configure`, `make`, and `make install` by executing the following commands:

```
cd /usr/src/php-5.0.2
./configure --with-regex=system --with-mysql=/usr/include
make
make install
```

The first configuration option specifies the regular expression library to use—in this case, the system library. The second option identifies the location of the header files that are required to provide MySQL support. When the build has completed, execute the following command from the PHP source directory to place the php binary in Apache's script directory:

```
cp sapi/cgi/php /var/www/cgi-bin/
```

Next, check that the Apache configuration file `httpd.conf` (in the development environment, this is found in `/etc/apache/`) contains the following lines:

```
LoadModule php5_module      libexec/libphp5.so
AddType    application/x-httpd-php .php .html .phtml
```

If it does, then PHP is a loadable module and Apache will parse all documents with extensions `php`, `html`, or `phtml`, and execute any embedded PHP code. If it doesn't, then Apache hasn't been configured to load PHP as a DSO. In either case, you want Apache to load the CGI version of PHP when it handles PHP/MapScript requests. In order to force Apache to load the CGI version of PHP, you must define a handler for documents with a specific extension and an action to perform when such a document is loaded. Add the following lines to the Document types section of `httpd.conf`:

```
AddHandler php-script .php
Action php-script /cgi-bin/php
```

Delete the reference to `.php` from the `AddType` line (if it exists), so it looks like this:

```
AddType    application/x-httpd-php .html .phtml
```

By removing `.php` from the `AddType` directive, files with this extension will use the CGI version of PHP even if the environment contains a DSO version. If you don't remove the reference, then Apache will continue to use the DSO version of PHP to handle documents of type `php`.

The PHP distribution contains a default configuration, `php.ini-dist`, that's suitable only for development (because of security concerns). Copy this configuration file to its default location.

```
cp php.ini-dist /usr/local/lib/php.ini
```

Open `php.ini` in a text editor and look for the directive `extension_dir`. The directory assigned is the location where the PHP interpreter will look for loadable modules (such as the MapScript module). In the development environment, this line is

```
extension_dir = "/usr/lib/php/extensions"
```

The directory specified in your environment might be different. Note the location, however, because you'll need it in the next section when you install MapScript. Finally, restart the Apache server

```
apachectl restart
```

and proceed to the PHP/MapScript build.

Building PHP/MapScript

If you had been reading ahead, this step wouldn't be required, but overloading the initial MapServer build with yet another configuration option would have complicated the process. It may even have stalled it if Apache and the PHP environment required changes. Now, however, with a functioning MapServer CGI application and some experience with the build process, you can proceed to a fairly simple recompile that will produce PHP/MapScript.

Recall that when you configured the build in Chapter 1, you used the following command:

```
./configure --with-proj --with-gdal --with-ogr
```

This provided support for the Proj.4, GDAL, and OGR libraries. In order to build the PHP/MapScript module `php_mapscript.so`, you just have to indicate the location of the PHP install directory. You do this in the development environment with the configure option `--with-php=/usr/local/include/php`. As noted in Chapter 1, the default library locations used by `make` may not be appropriate for your environment. In the present case, configure requires that you specify the location of the PHP install directory explicitly. (Also keep in mind that your location might be different).

You rebuild MapServer and build MapScript by typing the following:

```
make clean
./configure --with-proj --with-gdal --with-ogr --with-php=/usr/local/include/php
make
```

This will create the module `php_mapscript.so` in `./mapscript/php3/`. Copy this to the PHP extensions directory that you noted in the previous section (in the development environment, this is `/usr/lib/php/extensions/`), and ensure that all directories along the path to the module

are readable *and* executable by the Apache user. PHP needs to know that it must load this module when a script is executed. In order for this to happen, you must insert a directive in the `php.ini` file. You can place it after the `extension_dir` directive. It should read

```
extension = "php_mapscript.so"
```

You're now ready to proceed to the next section and create a PHP MapScript "Hello World" application.

The PHP/MapScript "Hello World" Application

The first MapScript application will be kept as simple as possible, merely replicating the functionality of the "Hello World" MapServer application. This will do two things: it will test the MapScript build just completed and it will demonstrate the key steps in creating an image and displaying it in a browser. The functionality will be duplicated by means of a simple trick—instead of building a map from scratch, you'll use all the specifications in the `hello.map` file.

When MapScript is instructed to create a map object by reading a mapfile, all the map parameters, layers, classes, and attributes are translated into MapScript objects. Default values are chosen for attributes that are left unspecified, and if the mapfile (when part of a CGI application) produces a map, the MapScript version will produce the same map. In addition to this, of course, all the MapScript objects can be modified programmatically. By building an application this way, you can focus your attention on the new MapScript functions without getting mired in mapfile details.

The code for this section is contained in the file `phpms_hello.php` in the code distribution available from the Apress website. It's shown in Listing 8-1.

As shown in the code snippet that follows, Line 001 opens the PHP script. A unique name is required for images each time the script is invoked, and this is accomplished in Line 003 by concatenating the string `phpms_hello`; a six-digit, zero-filled, random integer; and the string `.png`. In creating this image name, you could have given it any file extension—however, the mapfile will create a PNG image, and if the extension doesn't conform to the image format, the browser might become confused.

```
001  <?php
003  $image_name = sprintf("phpms-hello%0.6d",rand(0,999999)).".png";
```

Line 005 uses the MapScript constructor method `newMapObj()` to create a map object (referenced by the variable `$map`) by importing map specifications from the `hello.map` mapfile found under `/home/mapdata/`. `$map` now possesses all the characteristics of the map specified in the mapfile, but there isn't yet an image to display. Line 007 uses the `imageObj` method `draw()` to create the image, and returns a reference to it in `$image`. Line 008 uses the `imageObj` method `saveImage()` to write the image to the appropriate place on disk. Finally, Line 009 closes the block containing the PHP script.

```
005  $map = ms_newMapObj("/home/mapdata/hello.map");
006  // Create an image of the map and save it to disk
007  $image=$map->draw();
008  $image->saveImage("/var/www/htdocs/tmp/".$image_name);
009  ?>
```

Lines 010 through 018 generate the HTML tags needed to display the map image. Lines 010 through 012 write the opening HTML tags (<html>, <head>, <title>, and <body>). Lines 013 through 016 produce an HTML form. Line 013 opens a <form> tag, identifying the action as this script. Lines 014 and 015 create an input field of type image, with src pointing to the map image created previously. The URL to this image is inserted into the tag by the inline php echo command. Line 016 closes the form tag. Lines 017 and 018 close the tags that were opened earlier.

```
010 <html>
011 <head><title>PHP MapScript Hello World</title></head>
012 <body>
013   <form action="phpms_hello.php" method="POST">
014     <input type="image" name="img"
015       src="/tmp/<?php echo $image_name; ?>">
016   </form>
017 </body>
018 </html>
```

All the HTML generated is sent to the browser where the “Hello World” image is displayed with a tiny red dot between the words “Hello” and “World.” That’s all there is to it. Load the URL of this script (http://localhost/phpms_hello.php in the development environment) and execute it. This should display the image shown in Figure 8-1, in which you should see the same yellowish rectangle (with the words “Hello World” printed at the center) as you saw when you loaded `hello.html`.

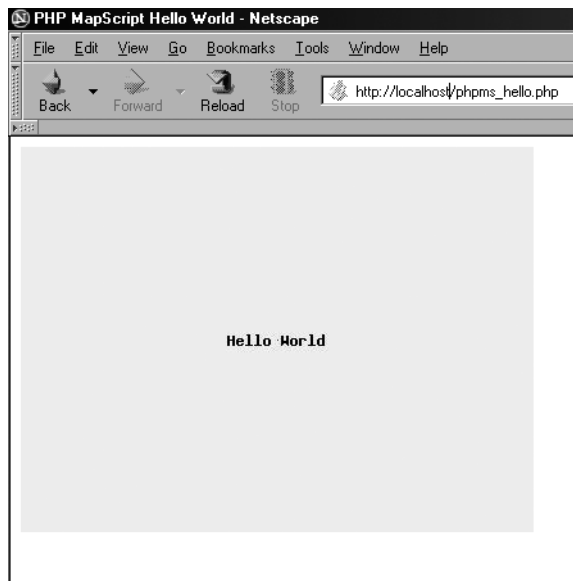


Figure 8-1. *The PHP/MapScript version of the “Hello World” application*

At the first presentation of the original CGI-based “Hello World” HTML file (`hello.html`), a broken image icon was displayed because Apache loaded the page as static HTML. Apache didn’t load MapServer at all because no form had been defined—therefore, no action was

specified. It was only when the user clicked the submit key that Apache invoked MapServer, which built the map image, replaced the substitution strings, and sent the HTML page (this time with a valid image) back to Apache for forwarding to the browser. However, this isn't how the MapScript application works.

Since the PHP script is contained within the web page itself, the first time Apache loads the HTML file (`phpms_hello.php`), it parses the file and searches for PHP commands to execute. (It does this because of the `AddType` directive that you inserted into `httpd.conf`.) The PHP MapScript commands create the map image and save it. Then, the PHP `echo` commands insert appropriate values into the HTML portion of the file. The page is complete before it gets sent to the browser, so no initialization file is required and no broken image icon is displayed.

A Practical PHP/MapScript Application

The application in the previous section confirmed that PHP MapScript was built properly, and demonstrated some fundamental MapScript functions, but it didn't provide a very useful or interesting map. In this section, you'll explore MapScript functionality in greater depth and produce an interactive map. You'll employ the same method as last time—that is, the initial definition of map parameters will be taken from a mapfile. However, this time, changes will be made to some of these parameters interactively, so that the application will behave the same way as its CGI-based counterpart. The mapfile used will be `third.map`. The code for this example can be found in the file `phpms_third.php`, and is available in the source distribution downloadable from the Apress website. The code is shown in Listing 8-2.

The script opens at Line 001. Lines 004 through 020 define the function `img2map()`. When you click on the map, the coordinates are returned in image coordinates. The `zoomPoint()` method requires the coordinates of the click point to be supplied in image coordinates as well. But when viewing the map, you'd probably like to know the position of the click point in terms of map or world coordinates (for example, latitude and longitude). There's no MapScript method to perform this calculation, so you're required to write your own.

The method is very simple. The width (and height) of the map is known in both pixels (`$width` and `$height`) and map coordinates (from the function parameter `$ext`). You know the click point coordinates in pixels, so you can use proportions to convert pixels to map coordinates. Lines 005 through 008 extract the maximum and minimum map coordinates from `$ext`. Lines 012 and 013 determine the number of map units per pixel (e.g., degrees per pixel or meters per pixel).

Note Map coordinates aren't restricted to decimal degrees. If you project your spatial data (more about that later), map coordinates will be actual distances like miles, kilometers, or feet—not angular measures.

Now, if a point is `x` pixels from the left edge of the image, you can obtain its longitude by multiplying `x` by the number of degrees per pixel and adding this to the longitude of the west (or left) side of the extent, as in Line 014. You do something similar for the height-to-latitude conversion, but keep in mind that pixel row count increases downward, so you must calculate the degrees of latitude per pixel, multiply by the number of pixels from the top of the image, and subtract that number from the maximum extent (as in Line 015). Lines 017 and 018 save the two coordinate values in an array in order to return them in Line 019.

```

001 <?php
002 //-----
003 // Convert from image to map coordinates

004 function img2map($width,$height,$point,$ext) {
005     $minx = $ext->minx;
006     $miny = $ext->miny;
007     $maxx = $ext->maxx;
008     $maxy = $ext->maxy;
009     if ($point->x && $point->y){
010         $x = $point->x;
011         $y = $point->y;
012         $dpp_x = ($maxx-$minx)/$width;
013         $dpp_y = ($maxy-$miny)/$height;
014         $x = $minx + $dpp_x*$x;
015         $y = $maxy - $dpp_y*$y;
016     }
017     $pt[0] = $x;
018     $pt[1] = $y;
019     return $pt;
020 }

```

Initially, this script (`phpms_third.phpml`) is invoked from the Location bar of the browser. This means that a form doesn't yet exist to pass CGI parameter values to the script. Default values have to be set for these parameters so that MapScript knows what to do the first time the script is invoked. This is done in Lines 023 through 046.

The HTML form generated by this script will invoke itself. It's identified in Line 023 (`phpms_third.php`) so that the name can be inserted into a form tag as the action. Next, Lines 025 through 027 identify the path to the mapfile, the mapfile itself, and the path to images created by this script.

```

023 $script_name = "phpms_third.php";
025 $map_path = "/home/mapdata/";
026 $map_file = "third.map";
027 $img_path = "/var/www/htdocs/tmp/";

```

The navigation defaults in Lines 029 through 032 set the initial zoomsize and the values of the select variables `$pan`, `$zoomin`, and `$zoomout`. Recall that an input variable of type radio can have several values associated with it. Only one of these can be selected at a time. If the state of a value is CHECKED, then the value that's CHECKED is returned. The navigation radio buttons are initialized so that the first time the user sees the web page containing the map, the map will be in Pan mode. The other values are set to empty strings.

```

029 $zoomsize=2;
030 $pan="CHECKED";
031 $zoomout="";
032 $zoomin="";

```


Lines 034 through 037 set the CHECKED state for several layers. HTML will be generated that will allow the user to select which layers should be displayed. In this case, the input variable will be of type CHECKBOX, which is similar to a radio button, except it allows either none, some, or all of the layers to be selected.

```
034 $urbanareas = "CHECKED";
035 $lakes = "CHECKED";
036 $states = "CHECKED";
037 $roads = "CHECKED";
```

When the user clicks on some point in the map image, the pixel coordinates of that mouse click are returned to the script. However, should the user click the submit button to refresh the image, a *virtual* click point needs to be created so that MapScript has some point of reference when it zooms in or out when refreshed. Lines 039 and 040 place this point at the center of the image. (Recall from the mapfile `third.map` that the image is 640 pixels wide by 480 pixels high.)

```
039 $clickx = 320;
040 $clicky = 240;
```

Next, Line 041 defines a MapScript `PointObj()`, and returns a reference in `$clkpoint`. You'll use this object to refer to the click point (real or virtual) and assign values to its coordinates later.

```
041 $clkpoint = ms_newPointObj();
```

A rectangle object, `rectObj()` is created in Line 042. A rectangle object consists of two coordinate pairs: the coordinates of the lower-left and upper-right corners of the rectangle. A `RectObj()` is MapScript's way of referencing a map extent. `$old_extent` refers to the extent of the map that has already been displayed in the browser (in the case of the first invocation, it refers to the default extent).

```
042 $old_extent = ms_newRectObj();
```

Line 044 defines the default extent as an array. The values of the array elements are the extent coordinates specified in the mapfile. On first invocation, this is the extent that will be saved on the web page as a hidden variable. Subsequent invocations will assign current values to the coordinates. Lines 045 and 046 create another `RectObj()` and set it to the default extent. The MapScript `zoomPoint()` method employed in the following code won't zoom out farther than this. This extent should also equal the extent specified in the mapfile, or else strange behavior occurs.

```
044 $extent = array(-180, 0, -60, 90);
045 $max_extent = ms_newRectObj();
046 $max_extent->setextent(-180, 0, -60, 90);
```

Line 048 creates a new `mapObj` based on the contents of the mapfile specified previously (i.e., `third.map`). The extent of this map is the extent defined in the mapfile, and the layers rendered are those for which the STATUS is on or default.

```
048 $map = ms_newMapObj($map_path.$map_file);
```

Lines 051 and 052 determine whether the script has been invoked by a form. All the code up to this point is executed every time the script runs, and default values have been assigned to most variables. If the script is invoked by the form, then the form variables retrieved by `$_POST[]` will be defined, and the block of code following the `if` statement will be executed.

```
051 if (( $_POST['img_x'] and $_POST['img_y'] )
052     or $_POST['refresh']) {
```

If the script isn't invoked by the form, then a value of `false` will be returned, and execution will drop through to Line 134 without executing any conditional code. Assume that this is the first invocation so that execution falls through.

Line 134 creates a unique identifier for the various images associated with this map by formatting a random number as a six-digit string. Lines 135 through 140 define file names and URLs for the map image, the reference map image, and the legend image.

```
134 $map_id = sprintf("%0.6d",rand(0,999999));
135 $image_name = "third".$map_id.".png";
136 $image_url="/tmp/".$image_name;
137 $ref_name = "thirdref".$map_id.".gif";
138 $ref_url="/tmp/".$ref_name;
139 $leg_name = "thirdleg".$map_id.".png";
140 $leg_url="/tmp/".$leg_name;
```

Line 142 uses the `draw()` method to create the map image. Next, the map image is saved using the `imageObj` method `saveImage()`. Lines 145 through 149 perform similar steps to create and save reference map and legend images.

```
142 $image=$map->draw();
143 $image->saveImage($img_path.$image_name);
145 $ref = $map->drawReferenceMap();
146 $ref->saveImage($img_path.$ref_name);
148 $leg = $map->drawLegend();
149 $leg->saveImage($img_path.$leg_name);
```

The next step is superfluous the first time the script is executed. Lines 152 through 155 retrieve the extent of the map just saved to disk and convert it to a space-delimited string. The first time through, this extent is the same as the default extent defined previously. Subsequent invocations, however (after zooming and panning), will produce different extents. This value will be saved in a hidden form variable and retrieved in the next invocation in order to specify the current extent of the map.

A `mapObj` possesses an extent. In order to access each of the four coordinates of the extent individually, a chain of references is used (`map->extent->minx`, for example).

```
152 $new_extent = sprintf("%3.6f",$map->extent->minx)." "
153             .sprintf("%3.6f",$map->extent->miny)." "
154             .sprintf("%3.6f",$map->extent->maxx)." "
155             .sprintf("%3.6f",$map->extent->maxy);
```

Line 157 retrieves the map scale from the map object and formats it for output. Lines 159 through 161 invoke the function `img2map()` to convert the mouse-click point from image coordinates to map coordinates (which in the present case are measured in decimal degrees), and format the returned values so they can be displayed in the web page. The script is closed in Line 163.

```
157 $scale = sprintf("%10d",$map->scale);
159 list($x,$y) = img2map($map->width,$map->height,$clkpoint,$old_extent);
160 $x_str = sprintf("%3.6f",$x);
161 $y_str = sprintf("%3.6f",$y);
163 ?>
```

The three required images (map, reference map, and legend) have been created and saved, and the scale and the new extent have been calculated, so you're now ready to generate the web page. Lines 164 through 166 print the preamble and opening tags for the web page (`<html>`, `<head>`, `<title>`, and `<body>`).

```
164 <html>
165 <head><title>MapScript Third Map</title></head>
166 <body bgcolor="#E6E6E6">
```

Note Since the web page had already been formatted as a template for the CGI-based MapServer application, it was imported directly into this script, and the substitution strings were replaced by PHP commands that insert appropriate values.

A form is opened in Line 167 with the action specified by the variable `$script_name`, which was set to `phpm3_third.php` in Line 023.

```
167 <form method=post action="<?php echo $script_name;?>">
```

A table is opened to format the output in Line 168. Following this, the values of several script variables are inserted into the page.

```
168 <table width="100%" border="1">
```

Lines 170 and 171 specify the element containing the map image, and the reference image is inserted in Lines 173 and 174.

```
170 <input name="img" type="image" src="<?php echo $image_url;?>"
171 width=640 height=480 border=2></td>
173 <img SRC="<?php echo $ref_url;?>"
174 width=300 height=225 border=1></td></tr>
```

The map scale, the click point coordinates, and the map extent are inserted in Lines 176 through 184.

```

176      Map scale:&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~?<br>
179      Click x,y:&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~?<br>
180      <?php echo $x_str;?>, <?php echo $y_str;?></font></td></tr><br>
181 <tr><td align="left" colspan="3"><font size="-1"><br>
182     <input type="hidden" name="extent"<br>
183       value="<?php echo $new_extent;?>">Map Extent:&nbsp;&nbsp;&nbsp;&nbsp;&~?<br>
184     <?php echo $new_extent;?></font></td></tr>
```

Line 188 inserts the legend image. In Lines 189 through 197, the navigation variables `$pan`, `$zoomin`, `$zoomout`, and `$zoomsize` are displayed. The submit button is defined in Line 198.

```

188 <tr><td rowspan="2"></td>
189 <td align="left"><font size="-1">
190 <input type="radio" name="zoom" VALUE=0
191 <?php echo $pan;?> Pan<br>
192 <input type="radio" name="zoom" VALUE=1
193 <?php echo $zoomin;?> Zoom In<br>
194 <input type="radio" name="zoom"
195 VALUE=-1 <?php echo $zoomout;?> Zoom Out<br>
196 <input type="text" name="ssize"
197 VALUE="<?php echo $zoomsize;?>" SIZE=2> Size<br>
198 <input type="SUBMIT" name="refresh" VALUE="Refresh"></td>

```

The layer selection controls are specified in Lines 199 through 207. The input variable `layer` will generally contain several values (i.e., the names of the selected layers). In order that these values are returned in a form that PHP understands, the name is specified as `layer[]`, with the square brackets indicating an array of values.

```

199         <td align="top">
200             <input type="checkbox" name="layer[]" value="urbanareas"
201                 <?php echo $urbanareas;?> >Urban Areas<br>
202             <input type="checkbox" name="layer[]" value="lakes"
203                 <?php echo $lakes;?> >Lakes<br>
204             <input type="checkbox" name="layer[]" value="states"
205                 <?php echo $states;?> >State Boundaries<br>
206             <input type="checkbox" name="layer[]" value="roads"
207                 <?php echo $roads;?> >Roads<br></font></td></tr>

```

The <table>, <form>, and other open tags are closed in Lines 208 through 211. The web page is now sent back to the browser via the Apache web server, and the user sees a page that's almost identical to the page displayed by the CGI-based MapServer application (as shown in Figure 8-2).

```
208     </table>
209 </form>
210 </body>
211 </html>
```

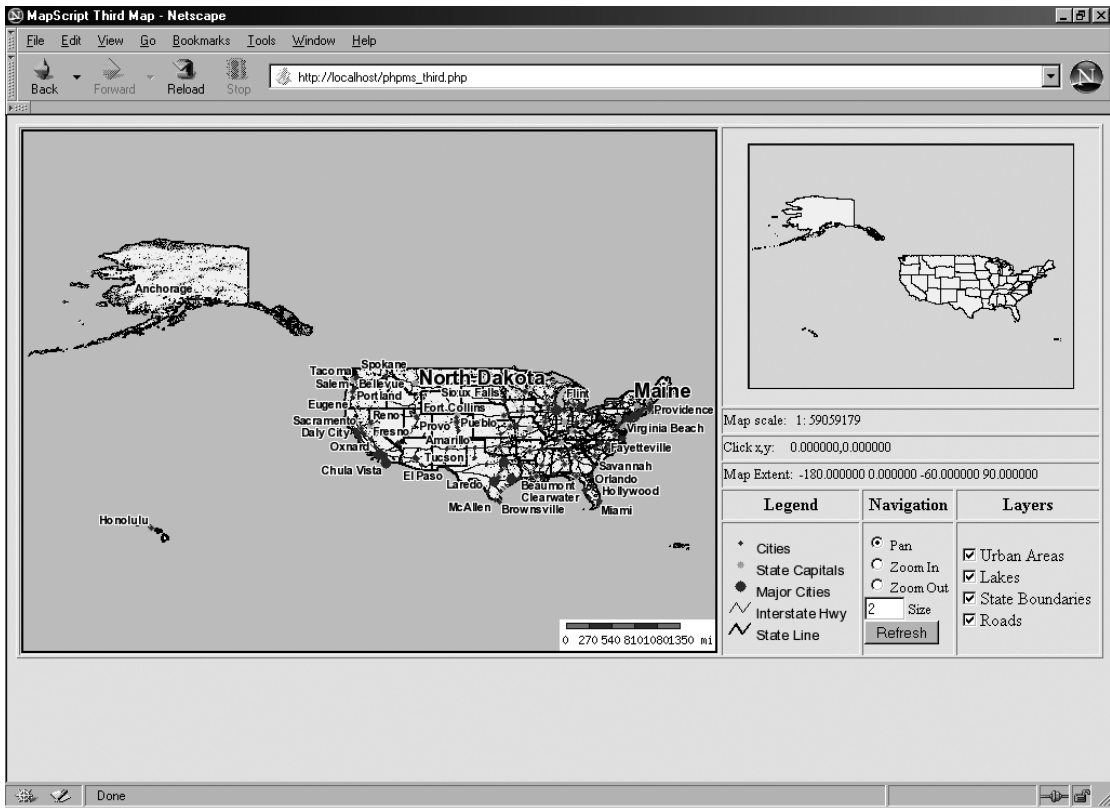


Figure 8-2. The PHP/MapScript version of the third application, *phpms_third.php*

Now, let's assume that the user looks at the map and changes the zoom state from Pan to Zoom In and clicks somewhere on the map image. The Apache server then receives the request from the browser and executes the script. The first part of the script (up to Line 051) is executed just as before—but now, when execution reaches the `if` statement, the values `$_POST['img_x']` and `$_POST['img_y']` will be defined, since the form variables `img_x` and `img_y` will have been returned from the browser. Therefore, execution proceeds to Line 053.

```
053     if ( $_POST['refresh'] ) { // Refresh, fake the coordinates
054         $clickx = 320;
055         $clicky = 240;
056     } else { // map click, use real coordinates
057         $clickx = $_POST['img_x'];
058         $clicky = $_POST['img_y'];
059     }
```

If the user had clicked Refresh, then `$_POST['refresh']` would return a true value, and the code in Lines 054 and 055 would assign fake image coordinates (at the center of the image) to the variables `$clickx` and `$clicky`.

But assuming that the user clicked on the map, the image coordinates of the click point are returned. Since the name of the input field containing the map image is `img`, these coordinates are returned as the values of form variables `img_x` and `img_y`. Lines 057 and 058 then save these values in `$clickx` and `$clicky`. Previously, an instance of `PointObj`, `$clkpoint`, was created—now the `PointObj` method `setXY` is used to set its coordinate values to `$clickx` and `$clicky` in Line 061.

```
061    $clkpoint->setXY($clickx,$clicky);
```

Lines 063 through 067 retrieve a list of layers that the user chose to display (by clicking the appropriate check boxes), and concatenate the list into the space-delimited list `$layers`. If no layers have been selected, the value of the variable `$layers` is set to an empty string, since an error is produced if you try to join an empty list. (Note the syntax in Line 200, which is used to return an array of values to a PHP script.)

```
063    if ( $_POST['layer'] ) {                                // any layers selected?
064        $layers = join(" ", $_POST['layer']); // yes
065    } else {
066        $layers = "";                                       // no
067    }
```

In Line 069, the PHP Perl regular expression function `preg_match()` is used to search for the string 'urbanareas' in `$layers`. If it's found, then the variable `$urbanareas` is set to `CHECKED`. Remember, if you want this layer to be checked when you generate the web page again, you must set the value of `$urbanareas`. Next, the `mapObj` method `getLayerByName()` is used to retrieve a pointer (`$this_layer`) to the layer named `urbanareas`. The pointer is then used to access the status of the layer and set it to on by assigning the value `MS_ON` to layer status. On the other hand, if the string 'urbanareas' isn't found, it means that the user has unchecked the `urbanareas` check box. Therefore, the variable `$urbanareas` is set to the empty string, and the layer status is set to off by assigning it the value `MS_OFF`. This is repeated for the other layers.

```
069    if (preg_match("/urbanareas/", $layers)){
070        $urbanareas = "CHECKED";
071        $this_layer = $map->getLayerByName('urbanareas');
072        $this_layer->set('status', MS_ON);
073    } else {
074        $urbanareas = "";
075        $this_layer = $map->getLayerByName('urbanareas');
076        $this_layer->set('status', MS_OFF);
077    }
```

In Lines 097 through 101, the form variable `extent` is retrieved, and its four components are split into array `$extent`. The elements of `$extent` are then used by the `mapObj` method `setExtent()` to set the extent of the map. Recall that whenever this script is executed, the extent of the map is set initially to the default value specified in the `mapfile`. Of course, on the previous invocation, it likely had some other value. This value would have been saved as a space-delimited string in the hidden variable `$extent`. It isn't until this point in the script that the value of this variable is retrieved, parsed, and used to set the map extent to what it was previously.

```

097     if ( $_POST['extent'] ) {
098         $extent = split(" ", $_POST['extent']);
099     }
101     $map->setExtent($extent[0],$extent[1],$extent[2],$extent[3]);

```

A `rectObj()` containing the current extent is required by the `zoomPoint()` method used in Lines 128 and 129, so the elements of the `$extent` array are used to set the values of the components of the `rectObj` `$old_extent` in Lines 103 and 104.

```

103     $old_extent->setextent(
104         $extent[0],$extent[1],$extent[2],$extent[3]);

```

Line 110 calculates the zoom factor that's passed to `zoomPoint()`. `$zoom_factor` is the product of the form variables `zoom` and `zsize`. Recall that `zoom` is set to 0 if `$pan` equals `CHECKED`, -1 if `$zoomout` equals `CHECKED`, and 1 if `$zoomin` equals `CHECKED`. Lines 112 through 126 then set the values of the navigation variables that are to be saved in the form. There are a couple of things to note. Line 113 sets `$zoom_factor` to 1 if `$zoom_factor` equals 0 since `zoomPoint()` can't accept a zoom factor of 0. Line 126 sets `$zoomsize` to the absolute value of form variable `zsize`, just in case a user should enter a negative value. (If a negative `$zoomsize` were allowed, the meanings of `Zoom In` and `Zoom Out` would be reversed.)

```

110     $zoom_factor = $_POST['zoom']*$_POST['zsize'];
112     if ($zoom_factor == 0) {
113         $zoom_factor = 1;
114         $pan = "CHECKED";
115         $zoomout = "";
116         $zoomin = "";
117     } elseif ($zoom_factor < 0) {
118         $pan = "";
119         $zoomout = "CHECKED";
120         $zoomin = "";
121     } else {
122         $pan = "";
123         $zoomout = "";
124         $zoomin = "CHECKED";
125     }
126     $zoomsize = abs( $_POST['zsize'] );

```

Finally, Line 128 employs the `zoomPoint()` method to center the map on the click point and then zoom in or out according to the value of `$zoom_factor`.

```

128     $map->zoomPoint($zoom_factor,$clkpoint,$map->width,
129         $map->height,$old_extent,$max_extent);

```

The map has now been created, and its extent has been changed by whatever zoom and pan factors the user entered. At this point, execution of the block conditioned on the presence of form variables ends. The next step is to draw the images and generate the HTML that will be forwarded to the browser. Since you've been through that part of the code before, you're now ready to see the script in action.

Enter the URL `http://localhost/phpms_third.shtml` into the address bar of your browser to display the resulting web page. It should resemble Figure 8-2.

This application will mimic (for the most part) the operation of its CGI-based cousin. However, there's one significant difference, which you can demonstrate by deselecting the roads layer and clicking Refresh. When the map is displayed again, the roads are still there and the check box is now checked. See Figure 8-3—you'll notice that the map's appearance is very different when all the selectable layers are turned off (except for the roads, which refuse to go away). What happened?

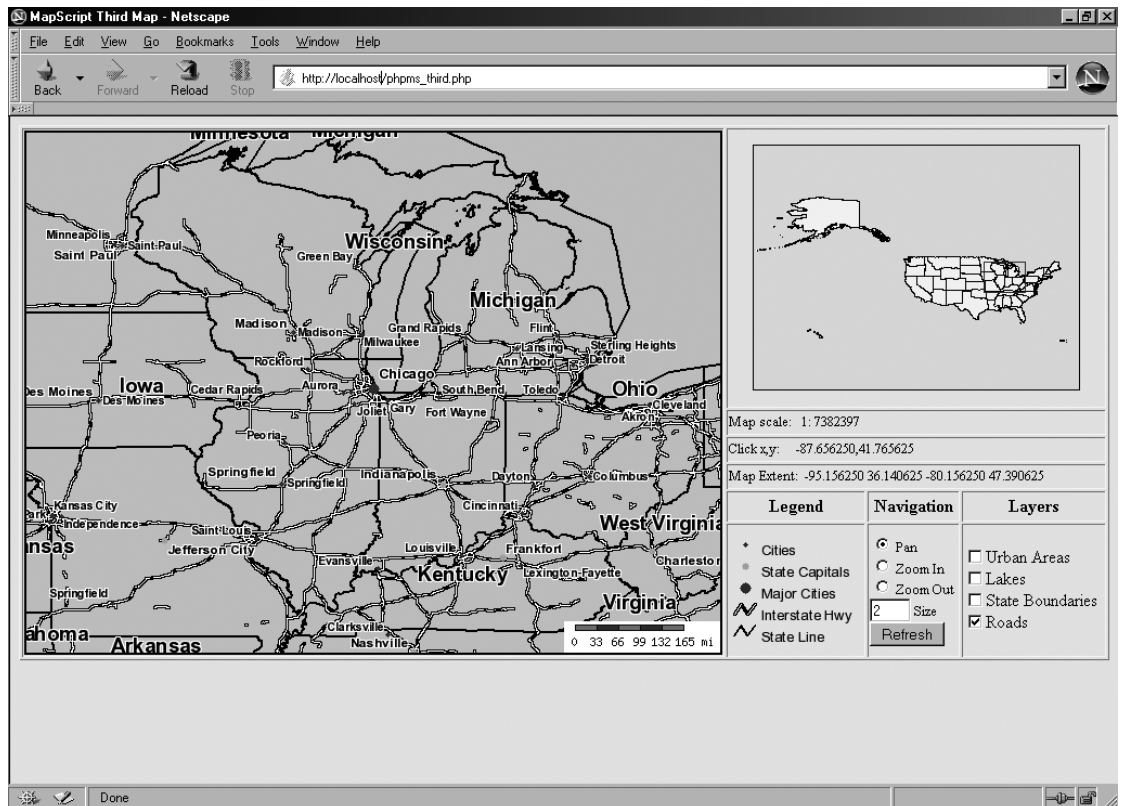


Figure 8-3. MapScript lacks a `*getLayersByGroupName()` method, which makes setting layer status for a group cumbersome.

You'll notice that the mapfile `third.map` contains several road layers that contain the keyword-value pair `GROUP roads`. By specifying a GROUP name for several layers in a mapfile, the CGI-based application can use the group name instead of the layer name to set the STATUS of each layer in the group. In this application, you used the `getLayerByName()` method to retrieve a reference to a layer. Using that reference, you could then set the status of the layer. MapScript, however, doesn't possess a `*getLayersByGroupName()` method. In order to use the group name, you would have to retrieve layers with the `getLayer()` method (which uses the layer number as reference) by searching through all layers and selecting those with the appropriate GROUP name.

In order to keep the application simple, this wasn't done in the present case. If you require it in your own applications, however, it's a straightforward application of techniques already described.

Summary

In this chapter, you've examined some basic methods of PHP/MapScript and created an application that duplicates the functionality of a MapServer CGI application. You've seen how to create a map object from a mapfile and manipulate some of its internal attributes and objects by means of MapScript methods. You've also learned how to draw and save the map image, and display it in an interactive web page. You haven't exhausted MapScript's capabilities, but you *have* created a firm foundation upon which you can build larger, more complicated applications that exercise more of MapScript's talents.

In Chapter 9, you'll build upon the skills acquired in this chapter to create a PHP application using both MySQL and MapScript. This application will be complete and demonstrate functionality that you'll be able to incorporate easily into your own applications to make them spatially aware.

Code Listings

The code for this chapter is presented here, complete and uninterrupted.

Listing 8-1. *The PHP/MapScript version of the "Hello World" application, `phpms_hello.php`*

```

001 <?php
002     // Create a unique image name every time through
003     $image_name = sprintf("phpms-hello%0.6d",rand(0,999999)).".jpg";
004     // Create a new instance of a map object
005     $map = ms_newMapObj("/home/mapdata/hello.map");
006     // Create an image of the map and save it to disk
007     $image=$map->draw();
008     $image->saveImage("/var/www/htdocs/tmp/".$image_name);
009 ?>
010 <html>
011 <head><title>PHP MapScript Hello World</title></head>
012 <body>
013     <form action="phpms_hello.php" method="POST">
014         <input type="image" name="img"
015             src="/tmp/<?php echo $image_name; ?>">
016     </form>
017 </body>
018 </html>

```

Listing 8-2. *The PHP/MapScript version of the third application, `phpms_third.php`*

```

001 <?php
002 //-----
003 // Convert from image to map coordinates

004 function img2map($width,$height,$point,$ext) {

005     $minx = $ext->minx;
006     $miny = $ext->miny;
007     $maxx = $ext->maxx;
008     $maxy = $ext->maxy;

009     if ($point->x && $point->y){
010         $x = $point->x;
011         $y = $point->y;

012         $dpp_x = ($maxx-$minx)/$width;
013         $dpp_y = ($maxy-$miny)/$height;

014         $x = $minx + $dpp_x*$x;
015         $y = $maxy - $dpp_y*$y;
016     }
017     $pt[0] = $x;
018     $pt[1] = $y;
019     return $pt;
020 }
021 //-----

022 // Default values

023 $script_name = "phpms_third.php";

024 // path defaults

025 $map_path = "/home/mapdata/";
026 $map_file = "third.map";
027 $img_path = "/var/www/htdocs/tmp/";

028 // Navigation defaults

029 $zoomsize=2;
030 $pan="CHECKED";
031 $zoomout="";
032 $zoomin="";

033 // Displayed layer defaults

```

```

034 $urbanareas = "CHECKED";
035 $lakes = "CHECKED";
036 $states = "CHECKED";
037 $roads = "CHECKED";

038 // Default click point

039 $clickx = 320;
040 $clicky = 240;
041 $clkpoint = ms_newPointObj();
042 $old_extent = ms_newRectObj();

043 // Default extent

044 $extent = array(-180, 0, -60, 90);
045 $max_extent = ms_newRectObj();
046 $max_extent->setextent(-180, 0, -60, 90);

047 // Retrieve mapfile and create a map from it

048 $map = ms_newMapObj($map_path.$map_file);

049 // If we've been invoked by the form, use form variables
050 // else drop through and create first map

051 if (( $_POST['img_x'] and $_POST['img_y'] )
052     or $_POST['refresh']) {

053     if ( $_POST['refresh'] ) { // Refresh, fake the coordinates
054         $clickx = 320;
055         $clicky = 240;

056     } else { // map click, use real coordinates

057         $clickx = $_POST['img_x'];
058         $clicky = $_POST['img_y'];
059     }

060     // Set the mouse click location (we need it to zoom)

061     $clkpoint->setXY($clickx,$clicky);

062     // Selected layers changed? set checkbox "CHECKED" status

063     if ( $_POST['layer'] ) { // any layers selected?
064         $layers = join(" ", $_POST['layer']); // yes

```

```
065     } else {
066         $layers = "";
067     }
068     $this_layer = 0;

069     if (preg_match("/urbanareas/", $layers)){
070         $urbanareas = "CHECKED";
071         $this_layer = $map->getLayerByName('urbanareas');
072         $this_layer->set('status', MS_ON);
073     } else {
074         $urbanareas = "";
075         $this_layer = $map->getLayerByName('urbanareas');
076         $this_layer->set('status', MS_OFF);
077     }

078     if (preg_match("/lakes/", $layers)){
079         $lakes = "CHECKED";
080         $this_layer = $map->getLayerByName('lakes');
081         $this_layer->set('status', MS_ON);
082     } else {
083         $lakes = "";
084         $this_layer = $map->getLayerByName('lakes');
085         $this_layer->set('status', MS_OFF);
086     }

087     if (preg_match("/states/", $layers)){
088         $states = "CHECKED";
089         $this_layer = $map->getLayerByName('states');
090         $this_layer->set('status', MS_ON);
091     } else {
092         $states = "";
093         $this_layer = $map->getLayerByName('states');
094         $this_layer->set('status', MS_OFF);
095     }

096     // retrieve extent of displayed map

097     if ( $_POST['extent'] ) {
098         $extent = split(" ", $_POST['extent']);
099     }

100     // Set the map to the extent retrieved from the form

101     $map->setExtent($extent[0],$extent[1],$extent[2],$extent[3]);

102     // Save this extent as a rectObj, we need it to zoom.
```

```

103     $old_extent->setextent(
104         $extent[0],$extent[1],$extent[2],$extent[3]);

105     // Calculate the zoom factor to pass to zoomPoint method
106     //
107     //   zoomfactor = +/- N
108     //   if N > 0 zoom in - N < 0 zoom out - N = 0 pan
109     //

110     $zoom_factor = $_POST['zoom']*$_POST['zsize'];

111     // Set the zoom direction checkbox status

112     if ($zoom_factor == 0) {
113         $zoom_factor = 1;
114         $pan = "CHECKED";
115         $zoomout = "";
116         $zoomin = "";
117     } elseif ($zoom_factor < 0) {
118         $pan = "";
119         $zoomout = "CHECKED";
120         $zoomin = "";
121     } else {
122         $pan = "";
123         $zoomout = "";
124         $zoomin = "CHECKED";
125     }
126     $zoomsize = abs( $_POST['zsize'] );

127     // Zoom in (or out) to clkpoint

128     $map->zoomPoint($zoom_factor,$clkpoint,$map->width,
129         $map->height,$old_extent,$max_extent);

130 }

131 // We've dropped thru because the script was invoked directly
132 // or we've finished panning, zooming and setting layers on or off

133 // Set unique image names for map, reference and legend

134 $map_id = sprintf("%0.6d",rand(0,999999));

135 $image_name = "third".$map_id.".png";
136 $image_url="/tmp/".$image_name;

```

```
137 $ref_name = "thirdref".$map_id.".gif";
138 $ref_url="/tmp/".$ref_name;

139 $leg_name = "thirdleg".$map_id.".png";
140 $leg_url="/tmp/".$leg_name;

141 // Draw and save map image

142 $image=$map->draw();
143 $image->saveImage($img_path.$image_name);

144 // Draw and save reference image

145 $ref = $map->drawReferenceMap();
146 $ref->saveImage($img_path.$ref_name);

147 // Draw and save legend image

148 $leg = $map->drawLegend();
149 $leg->saveImage($img_path.$leg_name);

150 // Save the extent after panning and zooming in
151 // a form variable as a space delimited string

152 $new_extent = sprintf("%.3.6f",$map->extent->minx)." "
153                     .sprintf("%.3.6f",$map->extent->miny)." "
154                     .sprintf("%.3.6f",$map->extent->maxx)." "
155                     .sprintf("%.3.6f",$map->extent->maxy);

156 // Format the scale of the image for display

157 $scale = sprintf("%10d",$map->scale);

158 // Convert click coordinates to map coordinates & format for display

159 list($x,$y) = img2map($map->width,$map->height,$clkpoint,$old_extent);
160 $x_str = sprintf("%.3.6f",$x);
161 $y_str = sprintf("%.3.6f",$y);

162 // We're done, output the HTML form

163 ?>

164 <html>
165 <head><title>MapScript Third Map</title></head>
166 <body bgcolor="#E6E6E6">
```



```
204         <input type="checkbox" name="layer[]" value="states"
205         <?php echo $states;?> >State Boundaries<br>
206         <input type="checkbox" name="layer[]" value="roads"
207         <?php echo $roads;?> >Roads<br></font></td></tr>

208     </table>
209 </form>
210 </body>
211 </html>
```




Extending the Capabilities of MapScript with MySQL

Previously, you used MapServer's CGI mode to create interactive maps, and you also duplicated that same map functionality using the MapScript API in Perl, Python, and PHP. This manner of presentation built upon your understanding of MapServer to introduce you to some of the features of MapScript in several languages you likely use on a regular basis. Although this provided a useful introduction, the present chapter will explore new territory to show how the API can significantly extend MapServer's reach and allow you to produce powerful, spatially aware applications.

There are a number of ways to give MapServer more power. For example, a spatially aware database management system such as PostGIS (based on PostgreSQL) could be used. Providing feature selection (via a SQL `WHERE` clause), this kind of technology supports more complex query applications than shapefile access allows. And, although shapefile speed can exceed that of more sophisticated storage and retrieval technologies, this is accomplished at the cost of the increased complexity required to tile the shapefiles and create spatial indexes. At some point, application complexity and performance issues will require that you examine this option. However, the added complexity of converting data sets and accessing the data from MapServer places this outside the scope of an introductory book.

Alternatively, we could have looked at a MapServer implementation in mobile hardware. With restricted data sets and inexpensive GPS hardware, running MapServer in a tablet PC (or even a PDA) is certainly possible. However, while this is certainly an interesting project, it seems too narrowly attractive to serve as an effective pedagogical exercise.

Instead, you'll create an application that bumps up the functionality of raw MapServer significantly, and provides practice with using some of the techniques already presented while presenting some important new ones.

If you've read each of the chapters describing the MapScript API implementations in Perl, Python, and PHP, you'll have noticed the similarities and differences between each implementation. Since the differences are small, and this application will be longer than the previous MapScript applications, only the PHP version will be described. If you have a clear understanding of the techniques so far presented, the conversion to Perl or Python will be straightforward.

Describing Application Requirements

One of the limitations of MapServer noted earlier is its inability to use items in joined DBF files to classify features. This is an example of the larger issue of MapServer's general lack of database capabilities. It's possible, of course, to employ a spatially aware database engine such as PostGIS, and manipulate the contents of the mapfile via URLs to obtain some added functionality (an interesting topic, but beyond the scope of this book), but this comes at the cost of increased complexity and still doesn't provide full database capability.

The alternative described in this chapter will require only a small increment in complexity over the previous MapScript application, and will provide the ability to display dynamic data on the map and retrieve attributes associated with these dynamic features.

The application will provide a simple locating service for a mythical restaurant chain called Slurp and Burp Restaurants. They specialize in coffee, several varieties of flat food, math lectures, and free WiFi access, but not all products or services are available at all locations.

Before we get started, allow me to give you some background on Slurp and Burp's fictional existence. A market survey indicated that a significant fraction of the Slurp and Burp clientele were itinerant early adopters of cellular wireless technologies. These customers wished to optimize their break time by avoiding restaurants (which Slurp and Burp calls stores) that don't provide the required products or services. In order to serve the needs of these customers, Slurp and Burp's Marketing VP decided that the company needed a website that would provide this information in a graphical manner.

This specification was passed to the IT Director, and after some consultation with store managers, a list of requirements was compiled. The company's rapid growth and continual reassessment of features ruled out the use of static maps because of the frequent need for updates. Generating dynamic maps containing store information required the use of a database, so in order to leverage IT experience with SQL, MySQL was chosen as the database engine. After extensive product research and detailed examination of several proprietary mapping engines, MapServer was selected as the map rendering tool. MapServer played to the strengths of the IT department because it's open source and possesses a PHP API. Additionally, it can interoperate with MySQL and supports the required dynamic capabilities.

The operational requirements were developed by a committee that included technical staff and several store managers. None had a background in mapping or GIS, but perusal of the MapServer Application Gallery (<http://mapserver.gis.umn.edu/gallery.html>) demonstrated the possibilities of the engine and allowed staff, naïve in the ways of mapping, to develop a realistic list of requirements.

The final design document specified the following requirements:

1. The initial display must present Slurp and Burp's market area with store locations indicated by markers of some sort.
2. When a user mouses over a marker, a box will display, containing some information about that store's location and hours of operation.
3. When a user clicks on the map, he or she will be provided with a list of stores located within a user-specified distance of the click point.

4. The user must be able to navigate by panning across the map and zooming in and out.
5. The information available for display will include store address, phone number, hours of operation, geographic coordinates, and services provided at each store.

In order to support these features, the application that follows will consist of an external MySQL database that contains tables describing various attributes of Slurp and Burp's stores (noted in Point 5). A PHP script will access MySQL to retrieve dynamic information, and will use the PHP MapScript API to render a map of an urban area in which several Slurp and Burp stores are located. The map will display street, neighborhood, and water features rendered from the contents of several shapefiles. Each store location will be marked with the Slurp and Burp logo (a cup of coffee), which MapScript will render directly from the geographical coordinates stored in the database.

Store details will be displayed in tool tips (small message boxes) that pop up when the mouse hovers over the symbol marking each store location. The tool tips will be implemented as a combination of third-party JavaScript code and HTML tags.

The user will also be able to enter a number representing a search radius, and then click on the map to perform a spatial query that returns a list of the stores within the specified distance of the mouse click.

In addition to this, the richer user interface will require that browser-specific interaction issues be addressed. The application will provide similar functionality in two browser environments: the Microsoft IE (Internet Explorer) environment and the Mozilla environment. JavaScript will be used to tailor the interface to the browser so that the same PHP code will work in both, without requiring any knowledge of which browser is requesting service.

Addressing Some Design Issues

In the previous section, the Slurp and Burp corporate context was used to describe one common way that technical staff are introduced to MapServer—through a request from a non-technical user for an application that falls pretty much outside the skill set of someone who doesn't work in the usual GIS environment. This context was adopted because it makes the fewest assumptions about the technical capabilities of the technical staff. It allows you to address (if briefly) most of the questions surrounding the selection of any technology: can it do what you need it to do, can it interface with your other systems, and what does it cost? Finally, it's a way to describe the operational requirements without addressing design issues in any depth. These issues are reviewed in the following section.

Mozilla vs. IE

The most significant of the design issues results from the differing capabilities of Mozilla-like browsers and IE-like browsers. Much of the functionality of this application depends on the sensing of the mouse pointer location over client-side imagemaps for the display of tool tips, and the sensing of mouse clicks on the image to provide pan and zoom capabilities. While both browser types have handlers for each of these events, they don't function the same way.

Note Mozilla-like refers not only to Mozilla itself but also the various Netscape and Firefox releases. IE-like refers to Microsoft's Internet Explorer and all other browser types.

As shown in the code snippet, Mozilla and Netscape allow an image in an `<input>` tag to be used as a client-side imagemap. The imagemap is used to track the position of the mouse pointer over the image and display a tool tip when the pointer is over an imagemap hot spot. At the same time, because the image is an input variable, it's also clickable so that the user can navigate the map simply by pointing and clicking.

```
<input name="img" type="image" src="/some/image" width=640 height=480 usemap="#map">
```

Microsoft IE and KDE Konqueror don't allow an image in an `<input>` tag to act as an HTML imagemap. In the code snippet that follows, the image in the `<input>` tag is clickable, allowing point-and-click navigation just like Mozilla, but since mouseover events aren't sensed, tool tips can't be popped up.

```
<input name="img" type="image" src="/some/image" width=640 height=480>
```

If the image is displayed in an `` tag, imagemap capabilities are available, but point-and-click navigation *isn't*, since the image is no longer an input variable.

```

```

Special handling is needed if the application is to function correctly in both types of browsers. This special handling is treated in greater detail in the code analysis that follows, but to anticipate this discussion, I'll note briefly that JavaScript code embedded in the HTML generated by the PHP script can determine which block of HTML tags (those for Mozilla or those for IE) is actually rendered when the page is loaded. By displaying the image in an `` tag and providing a different means of navigation for IE-like browsers, you can keep the functionality in the two environments fairly similar—but not identical.

Despite its importance to the operation of the application, extensive knowledge of JavaScript, while useful, isn't a heavy requirement. The pop-up tool tip, implemented with a third-party JavaScript library, is more or less a black box. The code used to tailor the navigation interface uses a conditional statement and a single JavaScript function, and is trivial.

Creating the MySQL Database

MySQL is a powerful, popular open source database engine. It understands SQL (Structured Query Language), which is more or less standard (although different implementations provide different flavors of SQL). The target version of the MySQL database engine used for the development of this book is 4.1, but any more recent version will be satisfactory. A detailed description of MySQL, including installation and use, is beyond the scope of this book—it's assumed that your environment already contains a functioning MySQL engine at least as recent as the target version.

Note MySQL is a topic worthy of its own book and many have been published. W. Jason Gilmore's book *Beginning PHP 5 and MySQL: From Novice to Professional* (Apress, 2004), provides an excellent introduction.

A potentially more serious issue is the PHP build configuration. The target version of PHP is 5.0.2—since this is a fairly recent release, it's likely that you'll need to install it. It's important that you include MySQL support when you configure the PHP build. In the last chapter, you added the option `--with-mysql=/usr/include` to build a MySQL-enabled version of PHP. If in doubt, enter the following tags into a file called `phpinfo.php`:

```
<?
    phpinfo();
?>
```

and load it into a browser. If your Apache server is configured to parse and execute files with the extension `.php`, then a page similar to Figure 9-1 will be displayed. This will confirm that your installed PHP libraries understand both MapScript and MySQL.

MapScript

MapServer Version	MapServer version 4.4.1 OUTPUT=GIF OUTPUT=PNG OUTPUT=JPEG OUTPUT=WBMP SUPPORTS=PROJ SUPPORTS=FREETYPE SUPPORTS=WMS_SERVER INPUT=EPPL7 INPUT=OGR INPUT=GDAL INPUT=SHAPEFILE
PHP MapScript Version	(\$Revision: 1.220.2.2 \$ \$Date: 2004/12/19 22:17:59 \$)

mysql

MySQL Support	enabled
Active Persistent Links	0
Active Links	0
Client API version	3.23.56
MYSQL_MODULE_TYPE	external
MYSQL_SOCKET	/var/run/mysql/mysql.sock
MYSQL_INCLUDE	-I/usr/include/mysql
MYSQL_LIBS	-L/usr/lib -lmysqlclient

Directive	Local Value	Master Value
mysql.allow_persistent	On	On
mysql.connect_timeout	60	60
mysql.default_host	no value	no value
mysql.default_password	no value	no value

Figure 9-1. The `phpinfo.php` page shows that both the MapScript and MySQL modules have been loaded.

Since you couldn't have gotten this far without PHP MapScript, the most likely error is a PHP installation that isn't MySQL aware. If this is the case, rebuild PHP with the MySQL configure option `--with-mysql=/usr/include` (using the appropriate location of the MySQL include files in your environment). If the build fails because you don't have MySQL installed, download the latest version from www.mysql.com and install it.

An understanding of the basics of SQL is required for retrieving information from tables, but the MySQL interface in PHP isn't difficult to use, and this application doesn't make extensive use of MySQL's capabilities. The application connects to the MySQL server on localhost, selects the restaurant database, and executes a straightforward SQL `select`. Initial creation of the restaurant database is done by loading a file containing the MySQL instructions that create the database and populate its tables. These instructions are located in the file `mapserver_create_restaurant`, in the code distribution available at the Apress website.

To create the restaurant database, run the MySQL client `mysql` under the administrator account—(typically the user ID is `root`) by executing the following command:

```
$ mysql -u root -p
```

Enter the password when requested. Assuming you've copied `mapserver_create_restaurant` to `/tmp/`, run the following command at the prompt, in order to create and populate the database:

```
mysql> source /tmp/mapserver_create_restaurant;
```

A series of messages will scroll past. At the prompt, run the following command:

```
mysql> show databases;
```

which displays the databases that MySQL knows about, as shown in the results that follow. In this case, you're only interested in the one just created, `restaurant`.

```
+-----+
| Database |
+-----+
| mysql    |
| restaurant |
| test     |
+-----+
4 rows in set (0.00 sec)
```

Now, select the restaurant database.

```
mysql> use restaurant;
```

Then, type the following command:

```
mysql> show tables;
```

which produces a list of the tables in the restaurant database.

```

+-----+
| Tables_in_restaurant |
+-----+
| menu                  |
| product               |
| store                 |
+-----+
3 rows in set (0.00 sec)

```

This confirms that the tables were created. Finally, check the creation of the store table by typing the following:

```
mysql> SELECT * FROM store;
```

which should display the following lines:

```

+-----+-----+-----+-----+-----+-----+-----+
| id | address                | latitude | longitude | phone | opentime | closetime |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | 201 Portage Ave        | 49.8955 | -97.1385 | 728-1234 | 08:00:00 | 20:00:00 |
| 2 | 1200 Grant Ave         | 49.8578 | -97.1692 | 958-6789 | 07:00:00 | 21:00:00 |
| 3 | 1436 Pembina Highway   | 49.8091 | -97.1565 | 726-6205 | 00:00:00 | 00:00:00 |
| 4 | 800 St Marys Rd        | 49.8442 | -97.1127 | 510-8976 | 08:00:00 | 00:00:00 |
| 5 | 10234 King Edward St   | 49.8983 | -97.2071 | 233-9248 | 06:00:00 | 12:00:00 |
+-----+-----+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)

```

You can check that each of the other tables was properly created the same way.

Creating the Application User Account

In the development environment, the user `mysql` is used by the application to access the database, but you can choose whatever user ID you like as long as you give it the appropriate privileges. For this application, `mysql` only needs read access to the database. Creating it requires MySQL root privileges. The password of user `mysql` in the application code is set to `password`, which isn't a good idea either, so change it to something more appropriate. Login as root and execute the following command:

```
$ mysql -u root -p
```

Enter the password when requested. At the prompt, run the following commands to install user `mysql` with password `password`:

```
mysql> GRANT select ON restaurant.* TO mysql@local IDENTIFIED BY 'password';
mysql> flush privileges;
```

The next section describes how to install the JavaScript library that supports the pop-up tool tips.

Installing the JavaScript Tool Tip Code

The tool tips used in this application are implemented using overlib, a third-party JavaScript library created by Erik Bosrup. The library is released under the *Artistic license*, and is available at www.bosrup.com/web/overlib. The site provides extensive documentation and examples. The overlib distribution contains the library itself, `overlib.js`, as well as a number of plug-ins that perform other functions. Installation of overlib is just a matter of unzipping the archive into some location in the Apache DocumentRoot. In the development environment, this is `/var/www/htdocs/`.

Patching PHP MapScript

Generally, every feature in a layer possesses a shape index (which is just a sequential number identifying the shape). The shape index is the usual means by which a feature is referenced. If a feature matches a spatial query, its shape index can be retrieved and used to reference that feature in subsequent processing.

This application renders some shapes dynamically. Instead of retrieving them from a shapefile, the features are drawn based on coordinates passed directly from the application to MapScript. You want to be able to use the store ID number as the shape index so that spatial query matches will return a pointer to the appropriate row in the store table.

Unfortunately, PHP MapScript (version 4.4.1) doesn't allow the shape index of a feature to be set—it's read-only. Attempting to retrieve the shape index for a matching feature from a dynamically generated layer always returns the value `-1`.

However, there's a solution: a patch that allows the shape index to be set from the script. (This patch will be incorporated into the next release, MapServer 4.6). This patch is necessary for performing the spatial search component of the application. If this isn't a critical function for you, then you can probably forgo installing the patch, but if you need the functionality and feel confident running a slightly unconventional version of MapScript, then install it. The patch is included with this chapter's application in the file `mapscript.c-patchedversion`.

The installation is straightforward. Put the new source file someplace safe, like `/tmp/`, and then change to the PHP MapScript directory and rename the current version of the main MapScript source before copying the new version. You can, if you like, rebuild the entire MapServer distribution with the new code installed for PHP MapScript, but that isn't necessary (and probably not even wise—you should always keep changes contained and to a minimum). Instead, just rebuild MapScript directly. The following procedure takes just a few minutes.

```
$ cd /usr/local/src/mapserver-4.4.1/mapscript/php3
$ cp mapscript.c mapscript.c-original-4.4.1
$ cp /tmp/php_mapscript.c-patchedversion ./php_mapscript.c
$ make
```

This re-creates `php_mapscript.so`, which you can then copy to the location specified in `php.ini`, in which PHP looks for loadable modules. (In the development environment, this is `/usr/lib/php/extensions/`.) PHP MapScript will now be able to set the value of `shapeindex`, which allows the shape index to be used as a pointer to the store table in the MySQL database.

Building the Application

This application creates a map displaying an urban area with streets, rivers, and neighborhoods. These features are all drawn from a spatial data set consisting of three shapefiles. From these three shapefiles, four layers are generated: a neighborhood layer, a hydrographic layer (in this case, a river), and two street layers. At scales below 1:120,000 (i.e., 1:120,001 and above), only major streets are shown; at larger scales, smaller thoroughfares are also rendered. Layer selection is available (as in previous applications), and the usual navigation features (pan and zoom) are available. However, depending on the browser used to view the map, pan and zoom are implemented differently.

On top of this urban map, large coffee cups are rendered, with each cup representing the location of a Slurp and Burp store. When the mouse pointer hovers over a cup, a small box is popped up that displays information about that store. The map can be used in Browse mode, in which normal navigation rules apply. But mode selection also allows the map to be used in Query mode. In this mode, a numerical value that represents a search radius can be entered. When the user clicks on the map image, all stores within the search radius around the click point are returned and used to build a table that displays information for the matching stores. This table is displayed below the map.

For the sake of completeness, the mapfile and PHP script for this application are shown in Listings 9-1 and 9-2. While you could type all 1,200 lines (after all, I originally typed them myself), I advise you to download this code from the Apress website.

From a learning standpoint, it's also advisable to install the application before analyzing the code—1,200 lines is a lot to digest without seeing it in operation. Since the MySQL database has been created, and the JavaScript tool tip library has already been installed, just copy the files `fifth.map` and `phpms_fifth.php` to the Apache DocumentRoot (`/var/www/htdocs/` in the development environment).

In the next section, I'll give a brief description of the application in action before proceeding to analysis of the code.

The Application in Action

Type the application URL `http://localhost/phpms_fifth.php` into the browser address bar and press Enter. You should see an image similar to Figure 9-2 if your browser is Mozilla-like, or Figure 9-3 if your browser is IE-like.

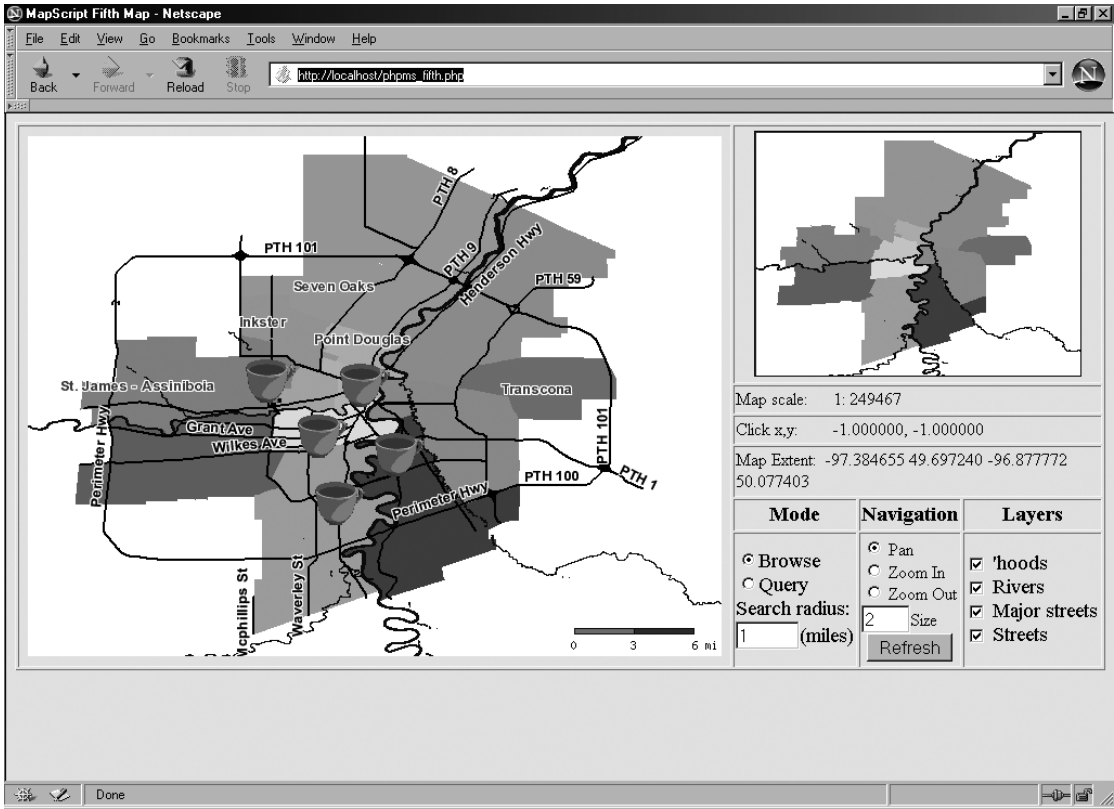


Figure 9-2. The initial display of the Slurp & Burp application in Netscape Navigator

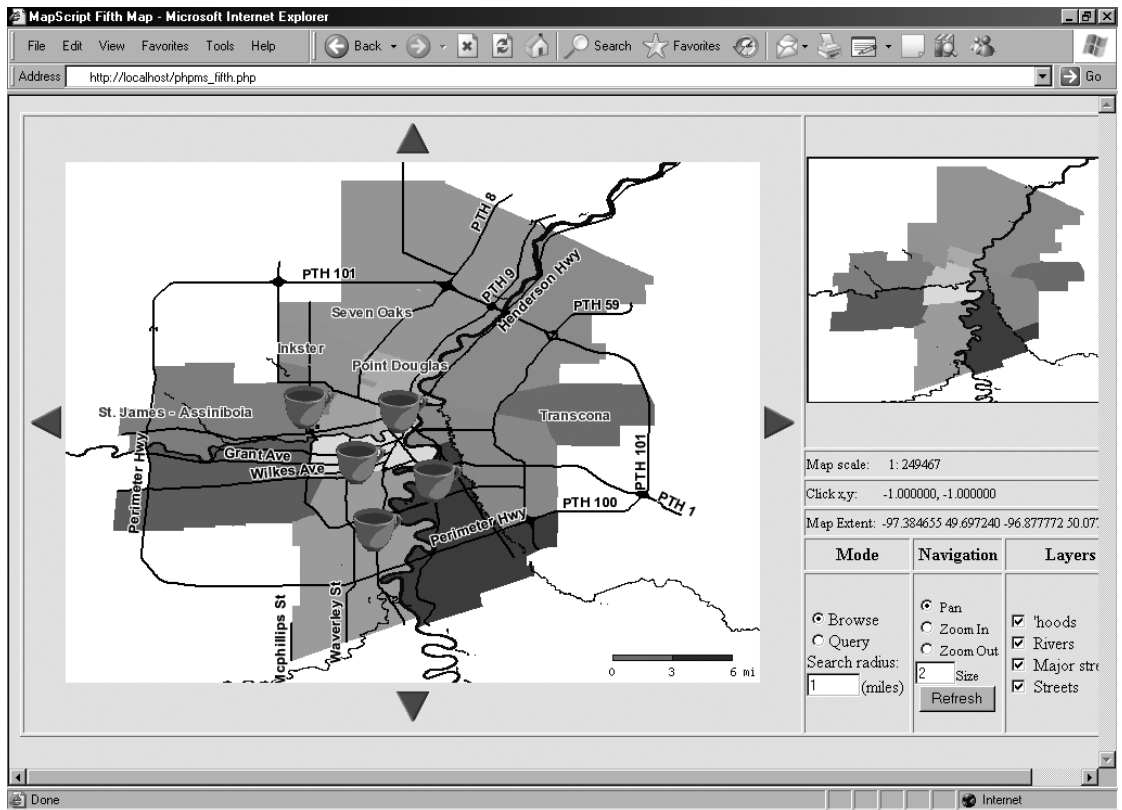


Figure 9-3. The initial display of the Slurp & Burp application in IE (note the presence of navigation arrows)

The Mozilla interface is the same as previous applications—just point and click to zoom and pan. Because of the IE imagemap issues discussed earlier, the IE interface is somewhat different. Notice the arrows at each side of the map. These are now the only means of panning the image. Clicking one of these arrows will pan the map in the direction of that arrow. In order to zoom, set the zoom direction and size, and click Refresh. If Zoom In is selected, this will zoom in to the center of the map image. If Zoom Out is selected, it will zoom out from the center. To zoom in toward a point near the upper-left corner of the image, you must first pan to bring this point to the approximate center of the map and then zoom in toward it.

Hover the mouse pointer over one of the coffee cups that represent store locations, and the tool tip will pop up to display some information about that store, as shown in Figures 9-4 and 9-5.

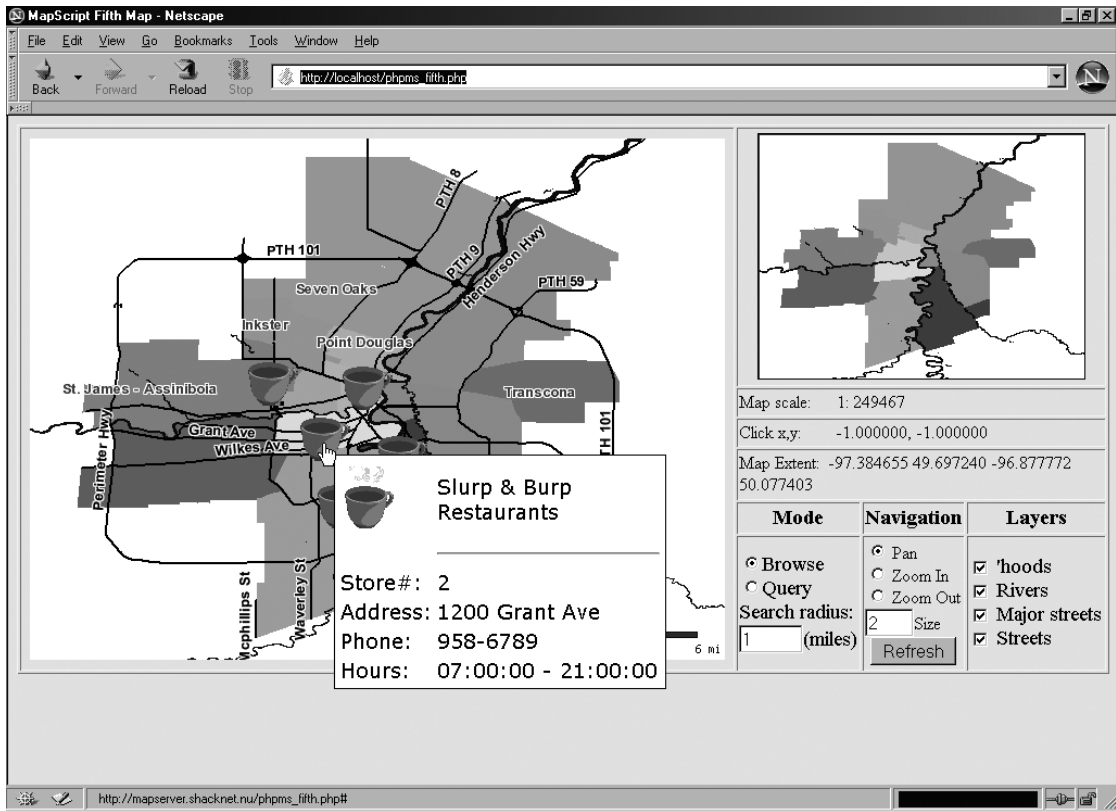


Figure 9-4. A pop-up tool tip displayed in Netscape Navigator

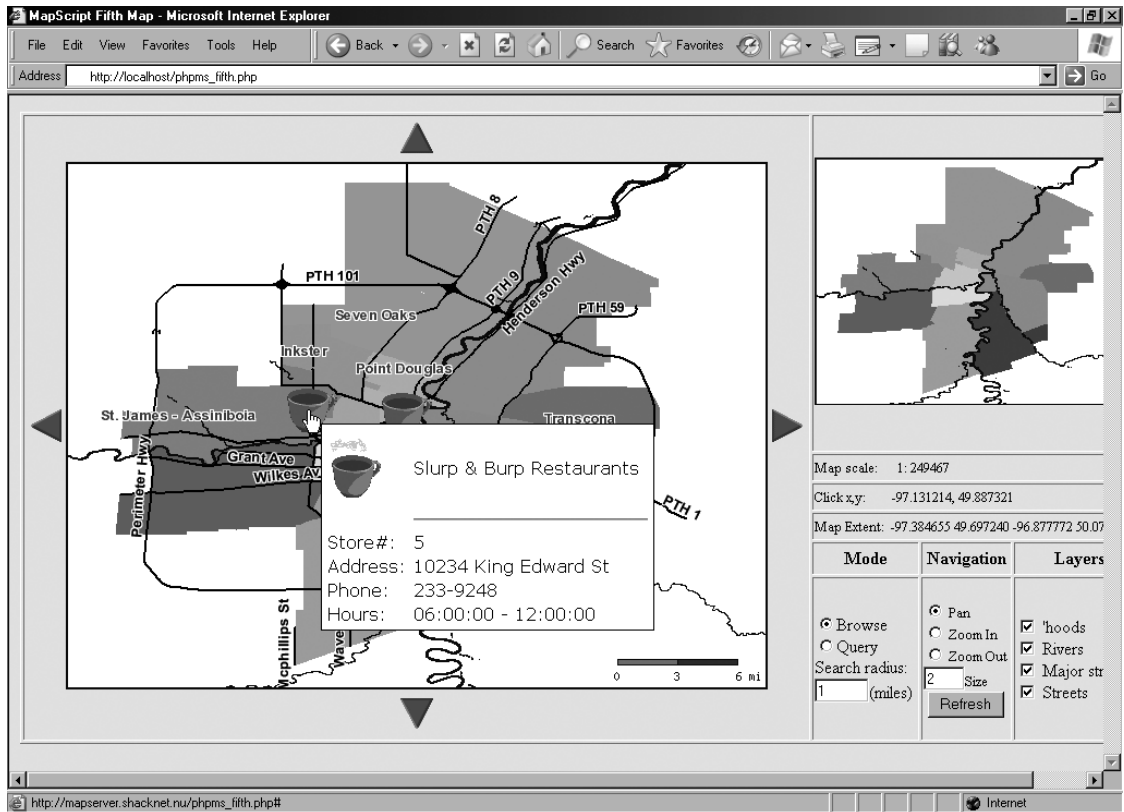


Figure 9-5. A pop-up tool tip displayed in IE

In the IE environment, try panning and zooming to get a feel for how the controls operate. Then change the mode from Browse to Query. Set the search radius to some large value (like 5 miles) and click somewhere on the map image. (IE users will need to click Refresh before being able to click on the map.) Since the search radius is large, the search will retrieve several stores and present a table of results at the bottom of the page. Note that the query will only match stores that are displayed on the map, so setting a large search radius and zooming in so that only a single store is visible will return just that store. If no stores are visible, then no matches will be found. Figures 9-6 and 9-7 show the results of the search.

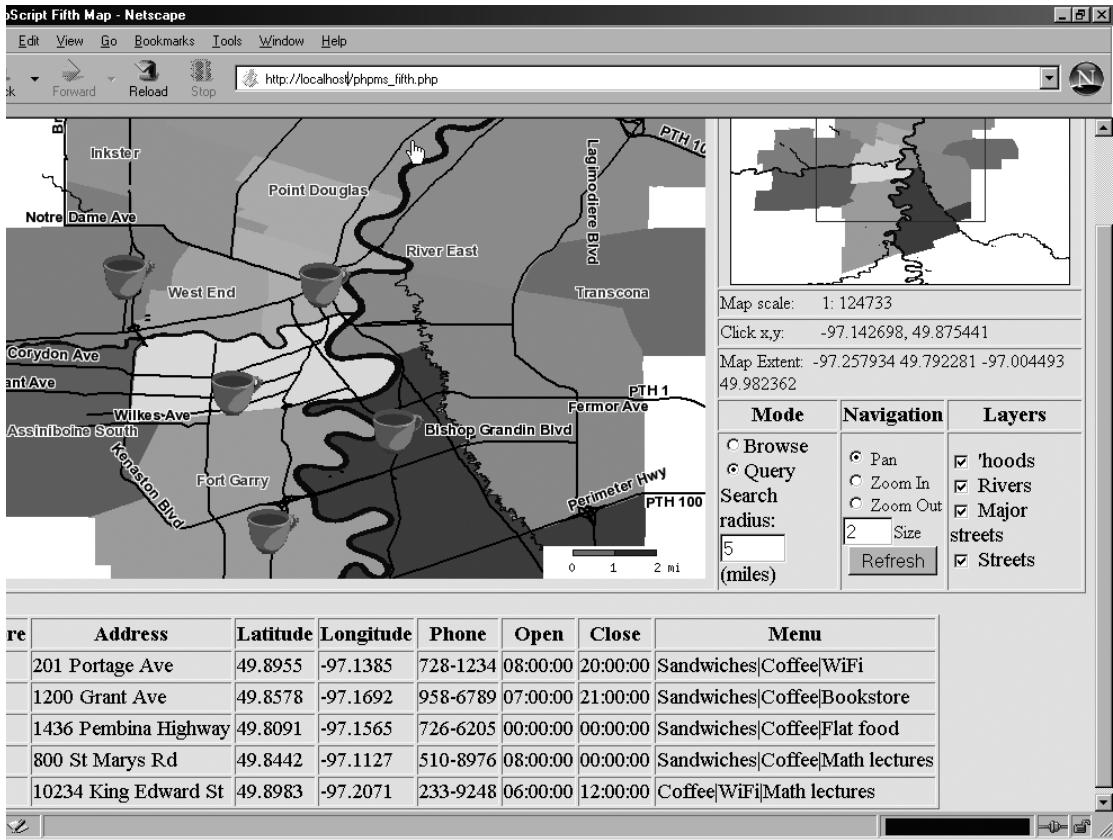


Figure 9-6. Multiple query results displayed in Netscape Navigator

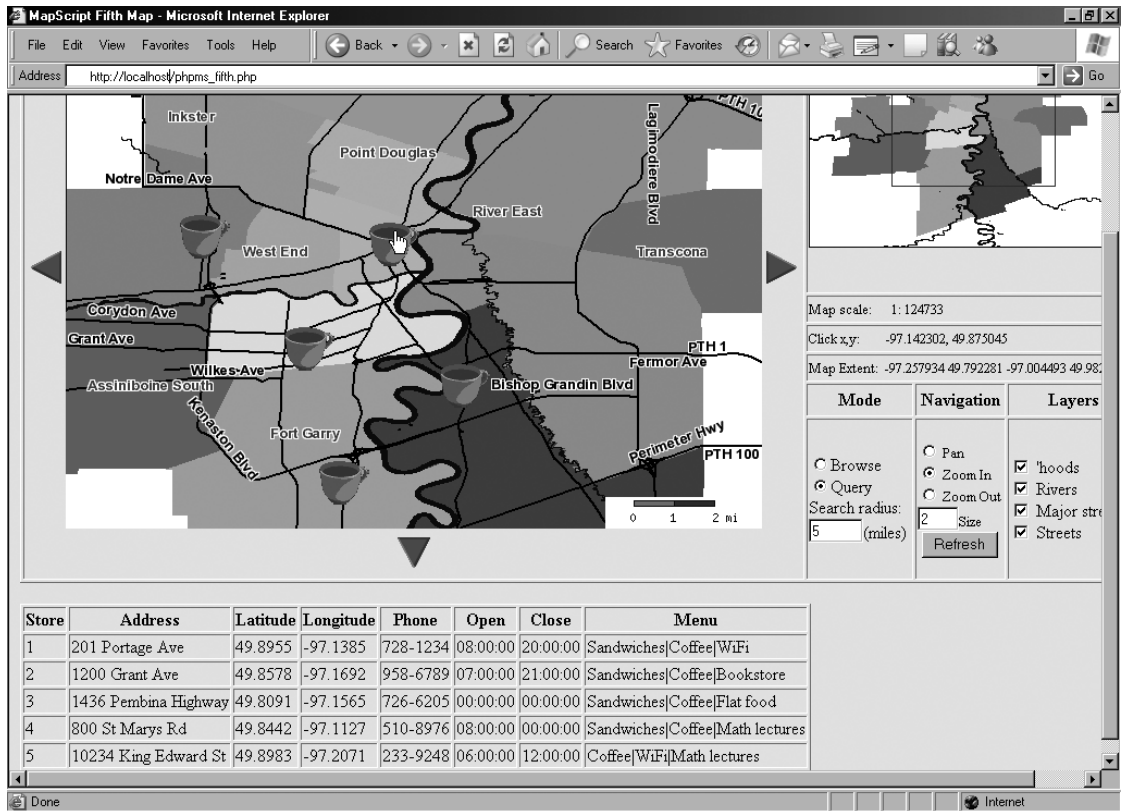


Figure 9-7. Multiple query results displayed in IE

Finally, if you look closely at Figure 9-8, you'll notice that the tool tip is displayed while the mode is set to Query. Because of IE's inability to track mouse cursor position on an input image, this isn't possible with IE (as you can see in Figure 9-9, in which the cursor is sitting on a cup but no tool tip is displayed).

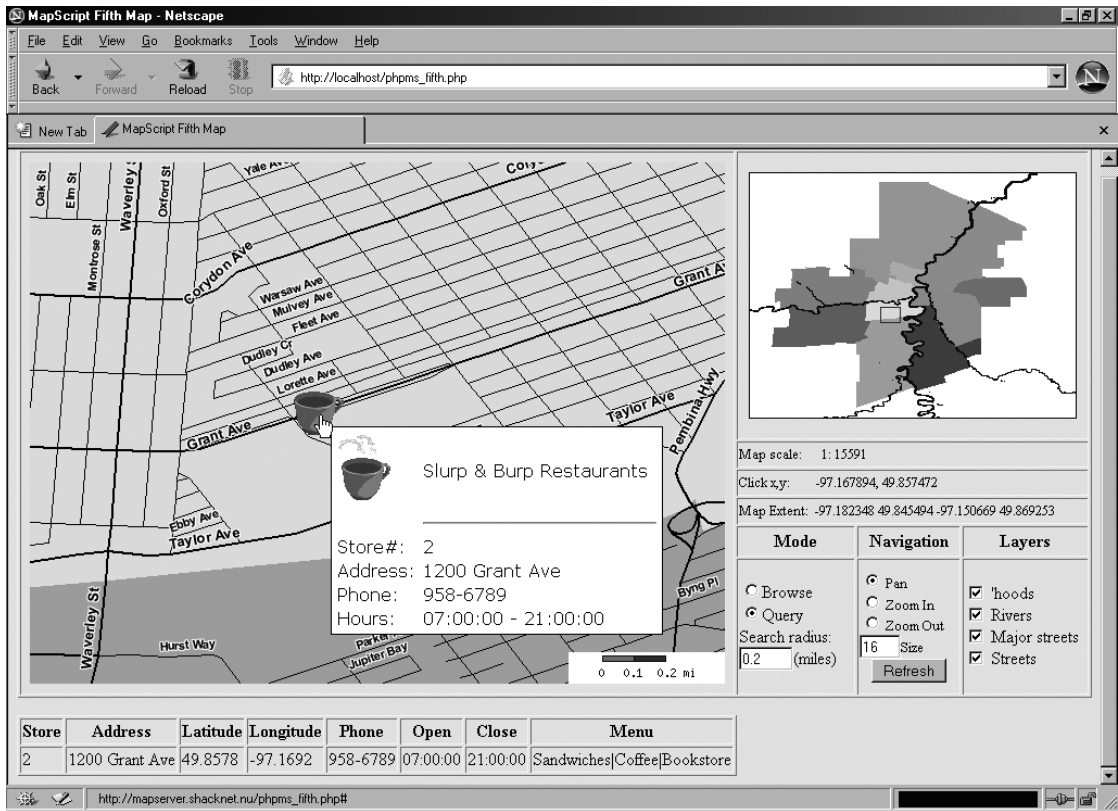


Figure 9-8. In Query mode, Netscape Navigator uses an input image to track both the coordinates of the mouse click and the coordinates of the mouse pointer as it hovers over the image—thus, it's possible to display a tool tip in this mode.

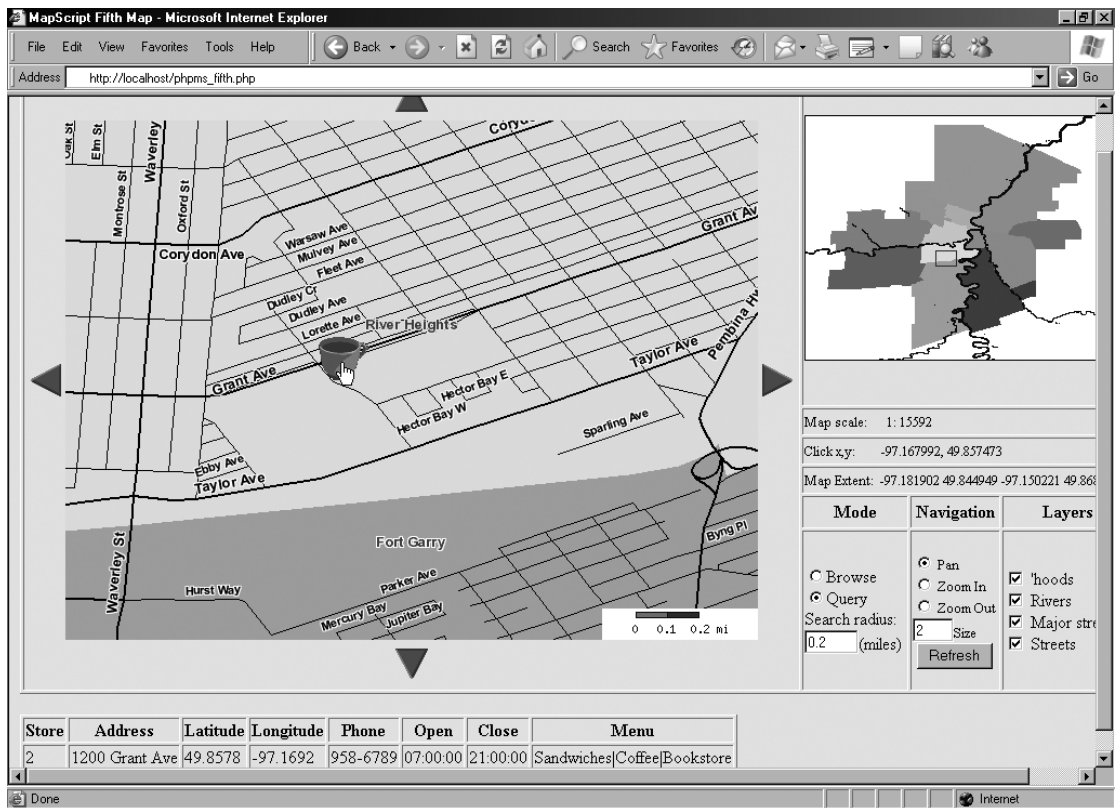


Figure 9-9. IE must use an input image to track mouse clicks in Query mode. Because it can't track hover coordinates at the same time, it's unable to display tool tips in this mode.

Experiment with the interface in several browsers to understand how the interfaces differ operationally, and how this might affect the functionality of your own application.

Mozilla-like browsers use the map image embedded in an `<input>` tag as both an input field for capturing mouse-click coordinates and an `imagemap` to capture hover coordinates—which means that tool tips are displayed regardless of mode. IE can capture click coordinates from an image embedded in an `<input>` tag *or* it can capture hover coordinates from an image embedded in an `` tag—but it can't use a single image to perform both functions.

Since you're now familiar with the operation of the application, the next section will proceed to a detailed analysis of the mapfile and PHP code.

Creating the Mapfile

As in previous chapters, you begin with a mapfile. There are no surprises in this mapfile—all the keywords have been used in previous applications. It's shown in its entirety in Listing 9-1. The first 26 lines contain the usual objects: map parameters like name, size, extent, etc. Lines 030 through 035, however, define a new type of symbol: `TYPE pixmap`. Symbols of this type employ an image (specified by the keyword `IMAGE`) to render a feature.

```

030 SYMBOL
031     NAME "Cup"
032
033     TYPE pixmap
034     IMAGE "/var/www/htdocs/cup.gif"
035 END

```

A reference map and scale bar are defined next in Lines 039 through 067, but I'll skip the description because you've seen both of these objects several times already.

Lines 071 through 261 specify a layer named hoods, which represents the neighborhoods of this urban area. The only thing notable about this layer is the 13 classes it contains (only the first is shown in the code snippet that follows). In previous applications, layers contained far fewer classes (usually only one or two). Having many classes in a layer, however, isn't an unusual occurrence; sometimes many distinctions are warranted in a single set of features. In the present case, each class represents a different neighborhood, and fewer classes wouldn't do justice to the facts on the ground. (13 is the minimum number of neighborhoods required to cover this urban area. In reality, there are many more.)

```

071 LAYER
072     NAME "hoods"
073     DATA "nrn_geo"
074     STATUS on
075     TYPE polygon
076     LABELCACHE on
077     LABELITEM "NRNNAME"
078     CLASSITEM "NRN"
079     CLASS
080         EXPRESSION /^01/
081         STYLE
082             COLOR 173 152 4
083         END
084         LABEL
085             TYPE truetype
086             FONT "arialbd"
087             SIZE 10
088             COLOR 140 66 28
089             POSITION cc
090             OUTLINECOLOR 255 255 255
091         END
092     END

```

Lines 265 through 277 describe a hydrographic layer—that is, a layer devoted to water features. Here, only the rivers are rendered. Notably, this is a polygon layer, which means that rivers will be rendered at their true extents—not merely as their centerlines. At small scales, rendering a river as a line makes sense, but at large scales, your map should show features more realistically.

```
265 LAYER
266     NAME "rivers"
267     DATA "waterp_geo"
268     STATUS on
269     TYPE polygon
270     CLASSITEM "HYDRO_NAME"
271     CLASS
272         EXPRESSION /RIVER*/
273         STYLE
274             COLOR 0 0 255
275     END
276 END
277 END
```

Lines 281 through 307 define a road layer—majorstreets. Features are selected based on the value of their STATUS attribute. If [STATUS] = 1, then the road is a major street.

```
281 LAYER
282     NAME "majorstreets"
283     DATA "roads_type"
284     STATUS on
285     TYPE line
286     LABELCACHE on
287     LABELITEM "NAME"
288     CLASS
289         EXPRESSION ([STATUS] = 1)
290         STYLE
291             SYMBOL "BigLine"
292             SIZE 2
293             COLOR 0 0 0
294         END
295         LABEL
296             TYPE truetype
297             FONT "arialbd"
298             SIZE 10
299             COLOR 0 0 0
300             OUTLINECOLOR 255 255 255
301             MINDISTANCE 300
302             POSITION auto
303             ANGLE auto
304             MINFEATURESIZE 25
305         END
306     END
307 END
```

Lines 311 through 338 describe another road layer (only a fragment is shown in the following code snippet), which renders non-major streets. This layer contains a lot of detail that would clutter the map and increase response time, so it's only rendered at scales larger than 1:120,000.

```

311 LAYER
312     NAME "streets"
313     DATA "roads_type"
314     STATUS on
315     TYPE line
316     LABELCACHE on
317     LABELITEM "NAME"
318     MAXSCALE 120000
319     CLASS
320         EXPRESSION ([STATUS] != 1)
321         STYLE
322         SYMBOL "BigLine"

```

The next layer, defined in Lines 342 through 360 (see the following code snippet), really defines the bread and butter of this application: the points-of-interest layer. This is a point layer with a single default class. Each point is marked by a Cup symbol circle with a width of 40 pixels. TOLERANCEUNITS is set to miles, so any spatial query made on this layer will have to match within some specified number of miles. However, there's an issue with the `queryByPoint()` method, in that it appears to ignore the TOLERANCEUNITS setting and uses native units, which in this case are decimal degrees. Specifying the keyword here has no effect now, but when this situation is rectified, it will be essential. This matter is addressed further when I discuss what changes the PHP script requires to handle this bug.

```

342 LAYER
343     NAME "poi"
344     STATUS default
345     TYPE point
346     LABELCACHE on
347     TOLERANCEUNITS miles
348     CLASS
349         SYMBOL "Cup"
350         SIZE 40
351         STYLE
352             COLOR 255 0 0
353         END
354         TEMPLATE "dummy.html"
355         TEXT ""
356         LABEL
357             BUFFER 20
358         END
359     END
360 END

```

One notable feature of this layer is its lack of a spatial data set. This layer will be used to render the coffee cups representing store locations. Since these are dynamic features, they don't exist as features in a spatial data set—each is simply a pair of coordinates stored in the MySQL table store. The layer itself could be created and populated directly by the PHP script. However, this would complicate the code, so instead, an empty layer is defined in the mapfile and populated with points relegated to the script.

Finally, in Line 354, a template named `dummy.html` is referenced. Recall that any layer that's to be queried must contain a class-level or layer-level query template. This is also the case when using MapScript. The keyword and value must be present—even if you're creating the map without using a mapfile. The file itself doesn't have to exist, however. Every queriable layer must possess a non-null template attribute, but the value associated with that attribute doesn't have to exist as a real object somewhere on a file system.

The PHP Script

In previous chapters, code changes have been incremental. This has provided a smooth transition from the trivial "Hello World" MapServer application at the beginning of the book, through the increasingly complex examples, to the point you've reached now: prepared to build a sophisticated, spatially aware, DBMS-using PHP application. It shouldn't surprise you that this step, too, is incremental. There are a few new MapScript methods and concepts to understand, but most of the application has already been built in preceding chapters.

The analysis of the PHP script will proceed, as before, in the sequence in which the script is executed, starting from first invocation through its invocation from the form. Most of the new code has been incorporated as a set of functions, and these will be described in detail as they're encountered. Those parts that are familiar will only be touched upon briefly.

First Invocation

When first invoked, execution begins at Line 258. The next few lines set the values of numerous variables that you've seen before—script name, path defaults, and navigation defaults. Four layers are defined in the mapfile. You set the values of the layer selection variables so that when the map is first presented to the user, all layers will be selected.

```
258 $script_name = "phpms_fifth.php";
262 $map_path = "/home/mapdata/";
263 $map_file = "fifth.map";
264 $img_path = "/var/www/htdocs/tmp/";
266 $zoomsize = 2;
267 $pan = "CHECKED";
268 $zoomout = "";
269 $zoomin = "";
271 $hoods = "CHECKED";
272 $rivers = "CHECKED";
273 $streets = "CHECKED";
274 $majorstreets = "CHECKED";
```

There are two modes of operation for this program: Browse mode, in which the usual pan and zoom controls are active and mouse clicks navigate the map; and Query mode, in which

navigation controls aren't active and mouse clicks initiate spatial queries on the points-of-interest layer poi. Lines 276 and 277 set the values of the mode selection variables so that the application starts in Browse mode. The search radius (in miles) is also specified. The search radius is used in the same way as the value of the keyword TOLERANCE in the mapfile. Remember that you set TOLERANCEUNITS to miles in the mapfile, although the current version of MapScript ignores this when performing a point query.

```
276 $browse = "CHECKED";
277 $nquery = "";
278 $radius = 1;
```

Next, default mouse-click points and extents are defined. Finally, in Line 290, the mapfile specified previously (fifth.map) is opened and a map is created.

```
281 $clickx = 320;
282 $clicky = 240;
283 $clkpoint = ms_newPointObj();
284 $old_extent = ms_newRectObj();
286 $extent = array(-97.384655, 49.697475, -96.877772, 50.077168);
287 $max_extent = ms_newRectObj();
288 $max_extent->setextent(-97.384655, 49.697475, -96.877772, 50.077168);
290 $map = ms_newMapObj($map_path.$map_file);
```

Lines 294 through 296 check to see how the script was invoked. If none of the form variables are defined, the script was loaded from the address bar of the browser, not from a button or a click on the map. Note that there are four form variables you haven't seen before: left_x, right_x, up_x and down_x, which indicate a mouse click on one of the four input images that represent the navigation arrows for up, down, left, and right.

```
294 if ( ( $ _POST['img_x'] and $ _POST['img_y'] ) or
295     $ _POST['refresh'] or $ _POST['left_x'] or
296     $ _POST['right_x'] or $ _POST['up_x'] or $ _POST['down_x'] ) {
```

These variables allow a single script to handle navigation tasks despite differences in browser capabilities. We'll return to a discussion of the buttons and their uses later.

Since this is the first invocation of the script, none of the form variables are defined, so control drops through to Line 414.

Retrieving Dynamic Information

At this point, some variables have been given default values, and the map has been drawn from the mapfile. Recall that when the mapfile layer poi was defined, no data source was specified. This layer must be populated with the coffee cups representing store locations. The information required to do this (i.e., the geographic coordinates of each store) is stored in the MySQL database that you've already created. Lines 414 through 416 retrieve this information, add points to the poi layer, and then create the pop-up tool tips.

```
414 $qresult = GetStoreTable();
```

In Line 414, the function GetStoreTable() is invoked to retrieve every row in the store table and return it as a row in the array \$qresult. The function is defined in Lines 161 through 177.

In the following code snippet, the PHP function `mysql_connect()` is used to connect to the database server on localhost. It passes the user ID (`mysql`), and the user's password (`password`). Prefixing the function name with an `@` sign suppresses the return of an error message if the connection can't be made. Instead, a tailored message is displayed by the `die` statement.

```
161 function GetStoreTable() {  
162     @mysql_connect("localhost", "mysql", "password")  
163     or die("Could not connect to MySQL server!");
```

Caution The topic of password security is far outside the scope of this book, but one point deserves mention. Don't use `password` as the password—it's used here for demonstration purposes only.

Once the connection to the server has been made, a database must be selected. In this case, it's the restaurant database created previously. The function `mysql_select_db()` takes two parameters. The first is the name of the database. The second is a resource ID that's only required when accessing more than one database at a time. Since you're accessing only the restaurant database, you can omit it. Here again, default error messages are replaced with tailored ones.

```
165     @mysql_select_db("restaurant")  
166     or die("Could not select database");
```

Lines 167 and 168 format the query string and execute the query. The query string selects all rows from the table `store`. Then, the function `mysql_query()` takes the query string as its only argument and performs the query, returning a reference to the query result set.

```
167     $query = "SELECT * FROM store";  
168     $result = mysql_query($query);
```

Note It's unwise to execute an unrestricted SQL `select` statement against a large table, but it's acceptable here since this table contains only five stores. In a production environment, you should limit the `select` statement to reduce the system load.

Lines 170 through 176 loop through the elements of the result set (using the function `mysql_fetch_array()` to retrieve each row of the result set) and save it as a row in the array `$qresult`. It takes two parameters—the reference to the result set `$result`, and a constant that specifies how the returned results are formatted. The array `$qresult` is then returned.

```
170     $i = 0;  
171     while ( $row = mysql_fetch_array($result,MYSQL_NUM) ) {  
172         $qresult[$i] = $row;  
173         $i++;  
174     }  
175     return $qresult;
```

Note There are three possible values for the second parameter of `mysql_fetch_array()`. They are `MYSQL_NUM` (which returns results as a numerically indexed array, with the fields in the same sequence as they're found in the table), `MYSQL_ASSOC` (which returns an associative array using field names for keys and field contents for values), and `MYSQL_BOTH` (which returns both array types).

All the information required to render store locations on the map and create the pop-up tool tips has been retrieved. The next section describes the process of adding points to a layer so that they can be drawn.

Adding Features to a Layer

Line 415 invokes the function `AddPoints()`, and passes it references to the map created on Line 290 and the array containing the store information retrieved from the MySQL database.

```
415 AddPoints( $map, $qresult );
```

Although fairly short in terms of code, the function `AddPoints()` isn't as straightforward as it appears. In order to add features to a layer, a reference to that layer must be created.

Line 180 uses the map object method `getLayerByName()` to retrieve a reference to the layer named `poi`, which is found in the map referenced by `$map`. Then, the array containing the query results is scanned. In Lines 188 through 190, a new point object (`$poi[]`), line object (`$ln[]`), and shape object (`$shp[]`) are created for every store. The latitude and longitude of each store is then used in Line 191 to set the coordinates of the point just created in Line 190. `$row[3]` is the longitude and `$row[2]` is the latitude of the store.

```
180 function AddPoints ( $map, $qresult ) {
185     $this_layer = $map->getLayerByName('poi');
186     $i = 0;
187     foreach($qresult as $row) {
188         $poi[$i] = ms_newPointObj();
189         $ln[$i] = ms_newLineObj();
190         $shp[$i] = ms_newShapeObj(MS_SHAPE_POINT);
191         $poi[$i]->setXY($row[3],$row[2]);
```

It might seem obvious that any feature added to a layer must be a shape object—however, a point isn't a shape object, it's a point object. In addition to this, you can't add a point object to a shape object directly—you must first add the point to a line object, then add the line to a shape object, and finally, add the shape object to a layer. Lines 192 and 193 accomplish this.

```
192         $ln[$i]->add($poi[$i]);
193         $shp[$i]->add($ln[$i]);
```

In MapScript version 4.4.1, it's not possible to set the shape index of a dynamically created feature. The shape index is the sequential identifier of each shape in a layer. While sequential, it's not necessarily consecutive—gaps are allowed. Since the shape index is the only link between the MySQL table containing store information and the features on the map, this is a critical issue. This has been noted previously, and if you need the spatial query functionality, you'll

have to install the patch. (This patch has made it into MapServer version 4.6, so the patch is only required for earlier versions.) Line 194 uses the newly available capability to set the shape index to the value of the store ID (`$row[0]`), which is an integer value.

```

194         $shp[$i]->set(index, $row[0]);
195         $this_layer->addFeature( $shp[$i] );
196         $i++;
197     }
198     return;
199 }
```

When points representing all the stores have been added to the layer, `AddPoints()` returns, and the map created previously now knows about the points added to the poi layer.

Creating the Imagemap

Line 416 invokes the function `CreateTImagemap()` with the query results array `$qresult` and the map reference `$map`. `CreateTImagemap()` returns a string, `$imagemap`, that contains the HTML tags needed to implement a client-side imagemap. It will be inserted into the data stream sent back to the browser when the CGI script file (`phpms_fifth.php`) is parsed and the embedded PHP instructions are executed.

```

416 $imagemap = CreateTImagemap($qresult,$map);
```

`CreateTImagemap()` is defined in Lines 120 through 136. As shown in the following code snippet, the first task is the creation of a point object to hold the geographical coordinates of a store. Then, in Line 124, the first tag in the imagemap is defined.

```

120 function CreateTImagemap($qresult,$map) {
123     $hotSpot = ms_newPointObj();
124     $imagemap = "<map name=\"stores\">";
```

The query result array is scanned and coordinates are set for the hot spot in Line 128. In Line 129, the function `MarkSpot()` is invoked with the sequence number, map width and height, map extent, and query result passed as parameters. Each invocation of `MarkSpot()` returns a string containing the HTML tags for a single imagemap `<area>` tag. Each `$newarea` is appended to `$imagemap` until the result set is exhausted and the complete imagemap block is returned in Line 135.

```

126     for ( $i = 0; $i < count($qresult); $i++) {
127         $row = $qresult[$i];
128         $hotSpot->setXY($row[3],$row[2]);
129         $newarea = MarkSpot($i,$map->width,$map->height,
130                             $hotSpot,$map->extent,$row);
131         $imagemap = $imagemap."\n".$newarea;
132     }
134     $imagemap = $imagemap."</map>\n";
135     return $imagemap;
136 }
```

The code for function `MarkSpot()`, defined in Lines 085 through 117, is shown in the block that follows. A hot spot consists of a square region surrounding the location of a store. The square is 30 pixels on each side. The store coordinates are in decimal degrees and have to be converted to image coordinates in pixels. In Line 093, the function `map2img()` inverts the transformation performed by `img2map()` that was described in the previous chapter—it's passed a point object with coordinates in decimal degrees, and it returns image coordinates in pixels.

```
085 function MarkSpot($seq,$width,$height,$point,$ext,$row) {
091     $size = 15;
093     list($x, $y) = map2img($width,$height,$point,$ext);
```

With the variables `$x` and `$y` now in image coordinates, a square extent about this point is determined. The extent is 30 pixels by 30 pixels. Some tinkering is done in Lines 095 through 098 to ensure that no extent extends beyond the image.

```
095     $xm = $x - $size; if ($xm < 0 ) {$xm = 0;}
096     $ym = $y - $size; if ($ym < 0 ) {$ym = 0;}
097     $xp = $x + $size; if ($xp > $width ) {$xp = $width;}
098     $yp = $y + $size; if ($yp > $height ) {$yp = $height;}
```

The coordinates of this extent are now used in the definition of the `imagemap <area>` tags. Lines 100 through 114 create the tag and set its name to the sequence number passed from the caller. The JavaScript event handler `onMouseOver` invokes `overlib` in Line 101. Then Lines 102 through 110 define a table for formatting the tool tip. Lines 110 and 111 specify some `overlib` parameters to set the color, size, and alignment of the tool tip. Line 112 invokes the `overlib` function `nd()` to close the tool tip. Line 113 determines the behavior if the hot spot is clicked—in this case, nothing happens. Line 114 adds coordinates and then closes the `<area>` tag, which is returned in Line 116.

```
100     $area = "<area name=\"\$seq\" ";
101     $area = $area."onmouseover=\"return overlib(";
102     $area = $area."<table width=300>";
103     $area = $area."<tr><td><img src=steamingcup.gif></td>";
104     $area = $area."<td>Slurp & Burp Restaurants</td></tr>";
105     $area = $area."<tr><td></td><td><HR></td></tr>";
106     $area = $area."<tr><td>Store#:</td><td>$row[0]</td></tr>";
107     $area = $area."<tr><td>Address:</td><td>$row[1]</td></tr>";
108     $area = $area."<tr><td>Phone:</td><td>$row[4]</td></tr>";
109     $area = $area."<tr><td>Hours:</td><td>$row[5] - $row[6]</td></tr>";
110     $area = $area."</table>','FGCOLOR,'#FFFFFF',BGCOLOR,'#000000','";
111     $area = $area."WIDTH,300,HAUTO,VAUTO);\" ";
112     $area = $area."onmouseout=\"return nd();\" ";

113     $area = $area."onclick=\"return false;\" ";
114     $area = $area."coords=\"\$xm,$ym,$xp,$yp\" href=\"#\>\n";
115     return $area;
117 }
```

Each store is now represented by a coffee cup displayed on the map, and as an <area> tag in a client-side imagemap block that contains store information to be displayed in the pop-up tool tip.

This section has described one method of producing pop-up tool tips, in which the creation of a client-side imagemap to sense mouseover events allows the use of a third-party JavaScript library to display and hide the information associated with each <area> tag.

Since this is still the first invocation, execution has dropped down past the pan, zoom, and query code to retrieve dynamic information from a MySQL database, drawn coffee cups on a map, and created the framework for displaying pop-up tool tips. The next task will be very familiar.

Creating and Saving Map Images

Lines 426 through 430 generate unique names for map and reference images. The images are rendered and saved in Lines 432 through 436.

```
426 $map_id = sprintf("%0.6d",rand(0,999999));
427 $image_name = "fifth".$map_id.".png";
428 $image_url="/tmp/".$image_name;
429 $ref_name = "fifthref".$map_id.".gif";
430 $ref_url="/tmp/".$ref_name;

432 $image=$map->draw();
433 $image->saveImage($img_path.$image_name);
435 $ref = $map->drawReferenceMap();
436 $ref->saveImage($img_path.$ref_name);
```

Some variables (map extent, map scale, and click coordinates) are formatted so they can be displayed on the page or saved as hidden variables in order to maintain state. Finally, the last task before sending the HTML form back to the browser is a call to the function `HandleIE()`, in Line 448. This returns a text string containing some HTML tags with embedded JavaScript code. The purpose of this function is the topic of the next section.

```
438 $new_extent = sprintf("%3.6f",$map->extent->minx)." "
439                 .sprintf("%3.6f",$map->extent->miny)." "
440                 .sprintf("%3.6f",$map->extent->maxx)." "
441                 .sprintf("%3.6f",$map->extent->maxy);
443 $scale = sprintf("%10d",$map->scale);
445 list($mx,$my) = img2map($map->width,$map->height,$clkpoint,$old_extent);
446 $mx_str = sprintf("%3.6f",$mx);
447 $my_str = sprintf("%3.6f",$my);

448 $NavigateIE = HandleIE($_POST['mode'],$image_url);
450 ?>
```

Using JavaScript to Fake Out IE

Unlike Mozilla-like browsers, in which an image can be used simultaneously as a clickable input form variable and a client-side imagemap, IE requires that an image perform one task or

the other. This means that an image used as an imagemap must be enclosed in an `` tag, and an image used as an input form variable must be enclosed in an `<input>` tag.

In the case of an `` tag, the mouse pointer position is available to JavaScript event handlers (such as `onMouseOver()`). If a user clicks on such as image, however, the form won't be submitted and the coordinates of the mouse click won't be returned to the CGI script. This application needs the JavaScript event handler to display and hide the pop-up tool tip, but it also needs the coordinates of the mouse click in order to perform a spatial query around that point. Since the two browser types provide such different capabilities with regard to this issue, some additional code is required to provide similar functionality regardless of browser type.

The technique used is similar to faking mouse-click coordinates when Refresh is clicked. Before the form is displayed in the browser, some JavaScript code embedded in the HTML is executed to determine which browser is being used. Some of this code is generated by the PHP script and inserted when the HTML is parsed for PHP commands, and some is embedded directly in the HTML.

The function `HandleIE()` is defined in Lines 021 through 052. Two parameters are passed: the URL of the map image, and the current mode. On first invocation, the mode is set to Browse. In Browse mode, you want tool tips to be able to pop up, so you must enclose the IE imagemap image in an `` tag. If you used this less elegant user interface for all browsers, there would be no need for this code—however, you still want to retain the greater functionality of the Mozilla browser, so you enclose the Mozilla imagemap in an `<input>` tag.

The following code snippet returns a string that contains JavaScript code that determines which browser has loaded the document, and thus uses the appropriate tag. If the browser is IE, then an `` tag is written into the document. If the browser isn't IE, then an `<input>` tag is written into the document. When the browser renders the document, the user will see the image in an `` tag or an `<input>` tag, depending on browser. Note that you only need to make this distinction in the case of Browse mode. In Query mode, both browsers require the image to be enclosed in an `<input>` tag. IE loses its pop-up tool tips when in Query mode, but Mozilla continues to show them.

```
021 function HandleIE($mode, $image_url) {
022     // browser is IE *and* mode is browse display map image in
023     // <img> tag otherwise display map image in <input> tag
024     if ($mode == "browse" ) {
025         $jscript = <<<ENDOFSCRIPT
026         <script>
027         if (navigator.appName!="Netscape")
028         {
029             document.write(
030                 ''
032             )
033         } else {
034             document.write(
035                 '<input name="img" type="image" src="$image_url" '
036                 +'width=640 height=480 usemap="#stores">'

```

```

037     )
038   }
039   </script>
040 ENDOFSCRIPT;
041   } else {
042     $jscript = <<<ENDOFSCRIPT
043     <script>
044       document.write(
045         '<input name="img" type="image" src="$image_url" '
046         +'width=640 height=480 usemap="#stores">'
047       )
048     </script>
049 ENDOFSCRIPT;
050   }
051   return $jscript;

052 } // end HandleIE

```

Now that the map image is enclosed in browser-appropriate tags, some of the navigation functionality lost by IE has to be replaced.

In the HTML section of the script, Lines 471 through 487 create a table with three rows and three columns. The top and bottom row each have a single element that spans the width of the table. The middle row has three elements.

```

471     <table border="0">
472 <!-- Display up arrow if browser is IE-like -->
473     <tr><td align="center" colspan="3"><?php echo Arrow("up"); ?>
474         </td></tr>
475 <!-- Display left arrow if browser is IE-like -->
476     <tr><td valign="center"><?php echo Arrow("left"); ?>
477         </td>
478 <!-- Displays image as <img> or <input> depending on mode and browser -->
479         <td><?php echo $NavigateIE; ?>
480         </td>
481 <!-- Display right arrow if browser is IE-like -->
482         <td valign="center"><?php echo Arrow("right"); ?>
483         </td></tr>
484 <!-- Display down arrow if browser is IE-like -->
485     <tr><td align="center" colspan="3"><?php echo Arrow("down"); ?>
486         </td></tr>
487     </table>

```

The middle element (Line 479) is the string with the appropriately tagged map image returned by the function `HandleIE()`. The contents of the other elements are shown in the code snippet that follows. The function `Arrow()`, defined in Lines 004 through 018, takes a single parameter: either up, down, left, or right. It returns a string containing JavaScript code that will display an input image of the appropriate arrow if the browser is IE; or an empty string if not.

Since these arrows are input fields, the CGI script can determine if any of them were clicked in the previous invocation. This will be used later to pan the image in the case of an IE browser.

```

004 function Arrow( $which ) {
005     // return javascript code to display navigation buttons
006     // if the browser is IE-like or nothing if it's not
007     $arrow = <<<ENDOFSCRIPT
008     <script>
009     if (navigator.appName!="Netscape")
010     {
011         document.write(
012             '<input name="$which" type="image" src="$which.png">'
013         )
014     }
015     </script>
016 ENDOFSCRIPT;
017     return $arrow;
018 }

```

Note that the purpose of this table is to tailor the navigation interface to the browser. IE users are presented with the tools to pan and zoom, while Mozilla users can continue to use point-and-click methods to navigate, without having to see the arrows. The same script handles both cases.

At this point, the main CGI script is completed. The HTML form following the script is parsed, any PHP instructions are executed, and the results are inserted into the page. The page is then sent back to the browser, and the browser executes the various JavaScript functions and displays the page with the appropriate navigation controls.

The user can zoom and pan, and turn layers on an off just as before—but now, whenever the mouse pointer hovers over a coffee cup, a small rectangle pops up and displays information about that store.

Suppose now that the user changes the mode from Browse to Query. What to do next depends on the browser, and that is the topic of the next section.

Performing Spatial Queries

For Mozilla-like browsers, if the user changes the mode to Query and clicks on the map image, it will cause a spatial query to be performed around the point clicked. If the browser is IE, however, the user can't click on the map since it's still just an image, not an input variable. Clicking Refresh will cause a query to be performed, but the click point will be the center of the map image—probably not what's desired. Subsequent clicks on the image *do* work as expected however. (It's no doubt possible to use JavaScript to re-render the images on the map without going back to the server and changing the tag to an <input> tag, but this hasn't been done.)

Assume that this is the third invocation so that there's no question of how the click point was generated—that is, assume the user clicked somewhere on the map. Assume also that the search radius was left at 1 mile.

Now, instead of dropping through at Line 294, the code inside the if block is executed. Input fields are evaluated to determine how the CGI script was invoked: with the Refresh button, with one of the arrow buttons, or by a click on the map. Since this mechanism has been used before, only the new navigation code will be described.

Lines 301 through 316 test whether one of the arrow keys has been clicked by checking for the existence of one of the associated form variables. For example, if the variable `$_POST['left_x']` exists, then the script knows that the user wants to pan to the left. A fake click point is then specified halfway between the center of the map image and the left edge. Control then drops out of the if block. A similar process is performed if one of the other arrow keys is clicked.

```

301      // left arrow clicked - pan left 1/4 image width
302      } elseif ( $_POST['left_x'] ) {
303          $clickx = 160;
304          $clicky = 240;
305      // right arrow clicked - pan right 1/4 image width
306      } elseif ( $_POST['right_x'] ) {
307          $clickx = 480;
308          $clicky = 240;
309      // up arrow clicked - pan up 1/4 image height
310      } elseif ( $_POST['up_x'] ) {
311          $clickx = 320;
312          $clicky = 120;
313      // down arrow clicked - pan down 1/4 image height
314      } elseif ( $_POST['down_x'] ) {
315          $clickx = 320;
316          $clicky = 360;

```

The script now checks the status and sets the variables `$nquery` and `$browse` to CHECKED or null, depending on the value returned from the browser. Recall that the CHECKED variables maintain state from one session to another. In order to avoid negative search radii, the absolute value of the quantity retrieved from the form is used.

```

325      if ( $_POST['mode'] == "nquery" ) {
326          $nquery = "CHECKED";
327          $browse = "";
328      } else {
329          $nquery = "";
330          $browse = "CHECKED";
331      }
332      $radius = abs( $_POST['radius'] );

```

Lines 334 through 378 deal with setting layers on and off and setting the extent of the map to the extent retrieved from the browser. I'll assume you're familiar with these topics and move on to something somewhat new.

Lines 381 through 383 convert the coordinates of the click point from image coordinates to geographic coordinates, and use those values to create a point object that will be used to perform a spatial query.


```

381     list($qx,$qy) = img2map($map->width,$map->height,$clkpoint,$old_extent);
382     $qpoint = ms_newPointObj();
383     $qpoint->setXY($qx,$qy);

```

The rest of the code up to Line 410 is concerned with navigation, another old topic that can be skipped. Control drops down to Line 414, where a few things happen: the MySQL database is queried to retrieve store information, points are added to the map to represent store locations, and pop-up tool tips are created.

Lines 419 through 424 perform the spatial query. Note that the map is now in the same state as it was when it was last displayed to the user. No panning or zooming was done, since `$_POST['mode'] == "nquery"` and the `zoomPoint()` method are only invoked if the mode is Browse.

```

419  if ( $_POST['mode'] == "nquery" ) {
421      $nearby = NearbyStores($qpoint,$map,$radius);
423      $result_table = BuildResultTable($nearby,$qresult);
424  }

```

The function `NearbyStores()`, defined in Lines 202 through 224, is invoked with the query point, map, and search radius passed as parameters. It returns a list of store IDs that were found within the search radius of the query point.

In order to perform a spatial search, a search layer has to be selected—Line 204 (shown in the following code snippet) uses the `getLayerByName()` method to return a reference to the poi layer.

```

202  function NearbyStores($point,$map,$radius) {
204      $qlayer = $map->getLayerByName('poi');

```

The next step presents another MapScript quirk. The `queryByPoint()` method ignores whatever `TOLERANCEUNITS` might be set, and instead uses native map coordinates. The workaround requires that you scale your search radius by the number of statute miles in 1 degree of latitude. Since this is approximately 69.04 miles per degree, you can still allow the user to specify query parameters in useful units like miles, while handling the quirk with this scaling process. The same process could be used for other units, but the scale factor would change.

```

205      $qlayer->set("tolerance",$radius);

```

In Line 212, the actual spatial query is performed. The `queryByPoint()` method takes three parameters: the query point (`$point`), a constant value that determines whether a query will return all results (`MS_MULTIPLE`) or just the first (`MS_SINGLE`), and the search radius (in this case, scaled by 69.04 statute miles per degree).

```

212      @$qlayer->queryByPoint($point, MS_MULTIPLE, $radius/69.04);

```

The query creates a `queryCacheObj` for the layer. It's accessed indirectly by the `getNumResults()` method in Line 213. If there *are* results, then the loop in Lines 215 through 220 retrieves each result one by one. It does so by first creating a reference to a result via the `getResult()` method in Line 217. This reference is then used to retrieve the shape index of the matching feature. This shape index is the integer-valued store number that was set previously. The store number is saved in an array. Finally, the array of store numbers is returned—or if there are no results, a null is returned.

```

213     $numResults = $qlayer->getNumResults();
215     if ($numResults != 0) {
216         for ($i = 0; $i < $numResults; $i++) {
217             $query_result = $qlayer->getResult($i);
218             $StoreList[$i] = $query_result->shapeindex;
219         }
220     } else {
221         $StoreList = "";      // no results
222     }
223     return $StoreList;
224 }

```

Displaying Spatial Query Results

Now that a list of nearby stores has been retrieved, an HTML table needs to be constructed to hold and display the results on the page. This task is performed by the function `BuildResultTable()` defined in Lines 227 through 255. Two parameters are passed: the list of nearby stores, and the query result array that was created when the stores table was first queried to retrieve store information and display it on the map and in the tool tips.

Lines 229 through 237 define the start of a table and a row of column headings. If there are stores nearby, then the table is filled with details from each nearby store. If there are no stores nearby, a single line explains this to the user.

```

227 function BuildResultTable($nearby,$qresult) {
229     $result_table = "<table border=1>\n<tr>";
230     $result_table = $result_table."<th>Store</th>";
231     $result_table = $result_table."<th>Address</th>";
232     $result_table = $result_table."<th>Latitude</th>";
233     $result_table = $result_table."<th>Longitude</th>";
234     $result_table = $result_table."<th>Phone</th>";
235     $result_table = $result_table."<th>Open</th>";
236     $result_table = $result_table."<th>Close</th>";
237     $result_table = $result_table."<th>Menu</th></tr>\n";

```

The loop in Lines 239 through 248 steps through the array `$nearby`, using the `$store` value to retrieve store details from the query result array. It retrieves the menu of each store with a call to the function `GetStoreMenu()`. Then, each row of the table is populated by the loop in Lines 244 through 246. Finally, the result table is returned to the caller.

```

239     if ( $nearby ) {
240         foreach ($nearby as $store) {
241             $row = $qresult[$store - 1];
242             $menu = GetStoreMenu($store);
243             $result_table = $result_table."<tr>";
244             for ($j = 0; $j < 7; $j++) {
245                 $result_table = $result_table."<td>$row[$j]</td>";
246             }
247             $result_table = $result_table."<td>$menu</td></tr>\n";
248         }

```

```

250     } else {
251         $result_table = $result_table."<tr><td colspan=8>No results</td></tr>";
252     }
253     $result_table = $result_table."</table>";
254     return $result_table;
255 }

```

Retrieving a store menu requires a somewhat more complex query string than was used to retrieve store information. In the first place, when items were retrieved from the store table in order to populate the poi layer, no selection criteria needed to be specified since all stores were to be selected. Second, only a single table needed to be accessed, whereas retrieving a menu requires that all the tables in the database be queried. This task is performed by the function `GetStoreMenu()` defined in Lines 139 through 158.

Line 141 connects to the MySQL server on localhost, and Line 143 selects the restaurant database.

```

139 function GetStoreMenu($store_id) {
141     @mysql_connect("localhost", "mysql", "password")
142         or die("Could not connect to MySQL server!");
143     @mysql_select_db("restaurant")
144         or die("Could not select database");

```

Lines 145 through 148 build the query string in stages. Line 145 selects the description field in the product table and specifies that the three tables (store, menu, and product) will all be referenced by this query.

Line 146 restricts the query to rows for which the value of `id` in the store table equals the value of `store_id` in the menu table. Line 147 similarly limits the query to rows for which the `product_id` in the menu table equals the `id` in the product table. Finally, Line 148 requires that the query return results only for the particular store ID passed to it.

```

145     $query = "SELECT product.description FROM store, menu, product ";
146     $query = $query."WHERE store.id=menu.store_id ";
147     $query = $query."AND menu.product_id=product.id ";
148     $query = $query."AND store.id=$store_id";

```

Line 149 performs the query, returning a pointer to the result set in `$result`. Lines 151 through 155 retrieve these results in sequence and store each product description in an element of the array `$item`. When all results have been retrieved, the elements of `$item` are joined into a string representing the menu, and returned.

```

149     $result = mysql_query($query);
151     $i = 0;
152     while ( $row = mysql_fetch_array($result,MYSQL_NUM) ) {
153         $item[$i] = $row[0];
154         $i++;
155     }
157     return join("|",$item);
158 }

```

At this point, the table of query results has been created and the map with store location symbols has been created. Imagemap code has been generated that holds the pop-up information. All that's left for this invocation to do is save the map image to disk, format some variables for display on the page, and shoot it back to the user.

That's it—you've created a PHP MapScript application that uses MySQL to store dynamic information, displays the dynamic information on a map, and allows spatial queries of that map. Now, download the code (if you haven't already), create the restaurant database, and fire up the browser to take a look at what it can do.

Summary

This chapter has consolidated your understanding of elementary MapServer and MapScript, and provided an introduction to the more sophisticated concept of MapScript spatial queries. Of course, you've barely scratched the surface—MapScript queries aren't limited to the simple point queries described in this chapter—all the query modes described in Chapter 5 are available to MapScript. Nevertheless, the foundation you've built is a solid one—the more complex query modes are accessed in much the same way as the NQUERY mode you've used here. The next step will be incremental rather than a great leap.

You've also seen how to extend MapScript's capabilities by incorporating MySQL functionality into a spatially aware application. PHP, Perl, and Python all possess numerous other special-purpose libraries that provide additional functionality—and these can be used in conjunction with MapScript in the same manner as MySQL. MapScript has supplied you with a powerful new tool—how you use it will be up to you.

Code Listings

Listing 9-1. *The mapfile for the restaurant application, fifth.map*

```

001 # This is our fifth map file
002 NAME "fifth"
003 UNITS dd
004 EXTENT -97.384655 49.697475 -96.877772 50.077168
005 SIZE 640 480
006 IMAGECOLOR 255 255 255
007 IMAGETYPE PNG
008 SHAPEPATH "/home/mapdata"
009 FONTSET "/var/www/htdocs/fontset.txt"

010 #####
011 # Symbol for drawing fat lines
012 #
013 SYMBOL
014     NAME "BigLine"
015     TYPE ELLIPSE
016     POINTS 1 1 END
017 END

```

```

018 #####
019 # Symbol for drawing spots
020 #
021 SYMBOL
022     NAME "Circle"
023     FILLED true
024     TYPE ellipse
025     POINTS 1 1 END
026 END

027 #####
028 # Symbol for drawing cup symbols
029 #
030 SYMBOL
031     NAME "Cup"
032
033     TYPE pixmap
034     IMAGE "/var/www/htdocs/cup.gif"
035 END

036 #####
037 # Reference map
038 #
039 REFERENCE
040     IMAGE "/var/www/htdocs/fifth_wpgref.gif"
041     SIZE 300 225
042     EXTENT -97.384655 49.697475 -96.877772 50.077168
043     STATUS ON
044     COLOR -1 -1 -1
045     OUTLINECOLOR 255 0 0
046 END

047 #####
048 # Scalebar
049 #
050 SCALEBAR
051     LABEL
052         COLOR 0 0 0
053         ANTIALIAS true
054         SIZE small
055     END
056     POSITION lr
057     INTERVALS 2
058     STATUS embed
059     SIZE 144 5
060     STYLE 0
061     UNITS miles

```

```

062     BACKGROUND_COLOR 255 0 0
063     IMAGE_COLOR 255 255 255
064     COLOR 128 128 128
065     OUTLINE_COLOR 0 0 255
066     TRANSPARENT off
067 END

068 #####
069 # Neighborhoods layer - hoods
070 #
071 LAYER
072     NAME "hoods"
073     DATA "nrn_geo"
074     STATUS on
075     TYPE polygon
076     LABELCACHE on
077     LABELITEM "NRNNAME"
078     CLASSITEM "NRN"
079     CLASS
080         EXPRESSION /^01/
081         STYLE
082             COLOR 173 152 4
083         END
084         LABEL
085             TYPE truetype
086             FONT "arialbd"
087             SIZE 10
088             COLOR 140 66 28
089             POSITION cc
090             OUTLINE_COLOR 255 255 255
091         END
092     END
093     CLASS
094         EXPRESSION /002/
095         STYLE
096             COLOR 192 92 74
097         END
098         LABEL
099             TYPE truetype
100             FONT "arialbd"
101             SIZE 10
102             COLOR 140 66 28
103             POSITION cc
104             OUTLINE_COLOR 255 255 255
105         END
106     END
107 CLASS

```

```

108      EXPRESSION /^03/
109      STYLE
110      COLOR 179 178 107
111      END
112      LABEL
113      TYPE truetype
114      FONT "arialbd"
115      SIZE 10
116      COLOR 140 66 28
117      POSITION cc
118      OUTLINECOLOR 255 255 255
119      END
120  END
121  CLASS
122      EXPRESSION /^04/
123      STYLE
124      COLOR 140 66 28
125      END
126      LABEL
127      TYPE truetype
128      FONT "arialbd"
129      SIZE 10
130      COLOR 140 66 28
131      POSITION cc
132      OUTLINECOLOR 255 255 255
133      END
134  END
135  CLASS
136      EXPRESSION /^05/
137      STYLE
138      COLOR 198 148 90
139      END
140      LABEL
141      TYPE truetype
142      FONT "arialbd"
143      SIZE 10
144      COLOR 140 66 28
145      POSITION cc
146      OUTLINECOLOR 255 255 255
147      END
148  END
149  CLASS
150      EXPRESSION /006/
151      STYLE
152      COLOR 98 142 96
153      END
154      LABEL

```

```
155         TYPE truetype
156         FONT "arialbd"
157         SIZE 10
158         COLOR 140 66 28
159         POSITION cc
160         OUTLINECOLOR 255 255 255
161     END
162 END
163 CLASS
164     EXPRESSION /^07/
165     STYLE
166         COLOR 206 156 24
167     END
168     LABEL
169         TYPE truetype
170         FONT "arialbd"
171         SIZE 10
172         COLOR 140 66 28
173         POSITION cc
174         OUTLINECOLOR 255 255 255
175     END
176 END
177 CLASS
178     EXPRESSION /^08/
179     STYLE
180         COLOR 169 173 99
181     END
182     LABEL
183         TYPE truetype
184         FONT "arialbd"
185         SIZE 10
186         COLOR 140 66 28
187         POSITION cc
188         OUTLINECOLOR 255 255 255
189     END
190 END
191 CLASS
192     EXPRESSION /^09/
193     STYLE
194         COLOR 165 173 90
195     END
196     LABEL
197         TYPE truetype
198         FONT "arialbd"
199         SIZE 10
200         COLOR 140 66 28
```



```

201             POSITION cc
202             OUTLINECOLOR 255 255 255
203         END
204     END
205     CLASS
206         EXPRESSION /^10/
207         STYLE
208             COLOR 193 187 144
209         END
210         LABEL
211             TYPE truetype
212             FONT "arialbd"
213             SIZE 10
214             COLOR 140 66 28
215             POSITION cc
216             OUTLINECOLOR 255 255 255
217         END
218     END
219     CLASS
220         EXPRESSION /^11A/
221         STYLE
222             COLOR 175 175 223
223         END
224         LABEL
225             TYPE truetype
226             FONT "arialbd"
227             SIZE 10
228             COLOR 140 66 28
229             POSITION cc
230             OUTLINECOLOR 255 255 255
231         END
232     END
233     CLASS
234         EXPRESSION /^11B/
235         STYLE
236             COLOR 196 200 72
237         END
238         LABEL
239             TYPE truetype
240             FONT "arialbd"
241             SIZE 10
242             COLOR 140 66 28
243             POSITION cc
244             OUTLINECOLOR 255 255 255
245         END
246     END
247     CLASS

```

```

248         EXPRESSION /^12/
249         STYLE
250         COLOR 239 231 140
251     END
252     LABEL
253         TYPE truetype
254         FONT "arialbd"
255         SIZE 10
256         COLOR 140 66 28
257         POSITION cc
258         OUTLINECOLOR 255 255 0
259     END
260 END
261 END

262 #####
263 # hydrographic layer - rivers
264 #
265 LAYER
266     NAME "rivers"
267     DATA "waterp_geo"
268     STATUS on
269     TYPE polygon
270     CLASSITEM "HYDRO_NAME"
271     CLASS
272         EXPRESSION /RIVER*/
273         STYLE
274         COLOR 0 0 255
275     END
276 END
277 END

278 #####
279 # Road layer - majorstreets
280 #
281 LAYER
282     NAME "majorstreets"
283     DATA "roads_type"
284     STATUS on
285     TYPE line
286     LABELCACHE on
287     LABELITEM "NAME"
288     CLASS
289         EXPRESSION ([STATUS] = 1)
290         STYLE
291         SYMBOL "BigLine"
292         SIZE 2

```

```

293         COLOR 0 0 0
294     END
295     LABEL
296         TYPE truetype
297         FONT "arialbd"
298         SIZE 10
299         COLOR 0 0 0
300         OUTLINECOLOR 255 255 255
301         MINDISTANCE 300
302         POSITION auto
303         ANGLE auto
304         MINFEATURESIZE 25
305     END
306 END
307 END

308 #####
309 # Road layer - streets
310 #
311 LAYER
312     NAME "streets"
313     DATA "roads_type"
314     STATUS on
315     TYPE line
316     LABELCACHE on
317     LABELITEM "NAME"
318     MAXSCALE 120000
319     CLASS
320         EXPRESSION ([STATUS] != 1)
321         STYLE
322             SYMBOL "BigLine"
323             SIZE 1
324             COLOR 0 0 0
325         END
326         LABEL
327             TYPE truetype
328             FONT "arialbd"
329             SIZE 8
330             COLOR 0 0 0
331             OUTLINECOLOR 255 255 255
332             MINDISTANCE 300
333             POSITION auto
334             ANGLE auto
335             MINFEATURESIZE auto
336         END
337     END
338 END

```

```

339 #####
340 # POI layer - points of interest
341 #
342 LAYER
343     NAME "poi"
344     STATUS default
345     TYPE point
346     LABELCACHE on
347     TOLERANCEUNITS miles
348     CLASS
349         SYMBOL "Cup"
350         SIZE 40
351         STYLE
352             COLOR 255 0 0
353         END
354         TEMPLATE "dummy.html"
355         TEXT ""
356         LABEL
357             BUFFER 20
358         END
359     END
360 END

361 END # mapfile

```

Listing 9-2. *The PHP script for the restaurant application, `phpms_fifth.php`*

```

001 <?php

002 //-----
003 // Arrow - display navigation arrows for IE

004 function Arrow( $which ) {

005     // return javascript code to display navigation buttons
006     // if the browser is IE-like or nothing if it's not

007     $arrow = <<<ENDOFSCRIPT
008     <script>
009     if (navigator.appName!="Netscape")
010     {
011         document.write(
012             '<input name="$which" type="image" src="$which.png">'
013         )
014     }
015     </script>
016 ENDOFSCRIPT;

```

```

017     return $arrow;

018 } // end Arrow

019 //-----
020 // HandleIE - allow spatial query when using IE

021 function HandleIE($mode, $image_url) {

022     // browser is IE *and* mode is browse display map image in
023     // <img> tag otherwise display map image in <input> tag

024     if ($mode == "browse" ) {
025         $jscript = <<<ENDOFSCRIPT
026         <script>

027         if (navigator.appName!="Netscape")
028         {
029             document.write(
030                 ''
032             )
033         } else {
034             document.write(
035                 '<input name="img" type="image" src="$image_url" '
036                 +'width=640 height=480 usemap="#stores">'
037             )
038         }
039         </script>
040     ENDOFSCRIPT;
041     } else {
042         $jscript = <<<ENDOFSCRIPT
043         <script>
044             document.write(
045                 '<input name="img" type="image" src="$image_url" '
046                 +'width=640 height=480 usemap="#stores">'
047             )
048         </script>
049     ENDOFSCRIPT;
050     }
051     return $jscript;

052 } // end HandleIE

```

```
053 //-----
054 // img2map - convert image coords to map coords

055 function img2map($width,$height,$point,$ext) {

056     // valid point required

057     if ($point->x && $point->y){

058         // find degrees per pixel

059         $dpp_x = ($ext->maxx - $ext->minx)/$width;
060         $dpp_y = ($ext->maxy - $ext->miny)/$height;

061         // calculate map coordinates

062         $p[0] = $ext->minx + $dpp_x*$point->x;
063         $p[1] = $ext->maxy - $dpp_y*$point->y;
064     }

065     return $p;

066 } // end img2map

067 //-----
068 // map2img - convert map coords to image coords

069 function map2img($width,$height,$point,$ext) {

070     // valid point required

071     if ($point->x && $point->y){

072         // find pixels per degree

073         $ppd_x = $width/($ext->maxx - $ext->minx);
074         $ppd_y = $height/($ext->maxy - $ext->miny);

075         // calculate image coordinates

076         $p[0] = $ppd_x * ($point->x - $ext->minx);
077         $p[1] = $height - $ppd_y * ($point->y - $ext->miny);
078         settype($p[0],"integer");
079         settype($p[1],"integer");
080     }
```

```

081     return $p;

082 } // end map2img

083 //-----
084 // MarkSpot - return an HTML imagemap area tag

085 function MarkSpot($seq,$width,$height,$point,$ext,$row) {

086     // Given the map size in pixels and the geographic
087     // extent of the map returns an <area> tag that
088     // contains Javascript event handlers that popup
089     // and hide tooltips on mouseovers.

090     // hotspot is point coordinates +/- $size pixels

091     $size = 15;

092     // get hotspot coords in pixels

093     list($x, $y) = map2img($width,$height,$point,$ext);

094     // calculate coordinates of imagemap area

095     $xm = $x - $size; if ($xm < 0 ) {$xm = 0;}
096     $ym = $y - $size; if ($ym < 0 ) {$ym = 0;}
097     $xp = $x + $size; if ($xp > $width ) {$xp = $width;}
098     $yp = $y + $size; if ($yp > $height ) {$yp = $height;}

099     // create <area> tag

100     $area = "<area name=\"\$seq\" ";
101     $area = $area."onmouseover=\"return overlib(";
102     $area = $area."<font><table width=300>";
103     $area = $area."<tr><td><img src=steamingcup.gif></td>";
104     $area = $area."<td>Slurp & Burp Restaurants</td></tr>";
105     $area = $area."<tr><td></td><td><HR></td></tr>";
106     $area = $area."<tr><td>Store#:</td><td>$row[0]</td></tr>";
107     $area = $area."<tr><td>Address:</td><td>$row[1]</td></tr>";
108     $area = $area."<tr><td>Phone:</td><td>$row[4]</td></tr>";
109     $area = $area."<tr><td>Hours:</td><td>$row[5] - $row[6]</td></tr>";
110     $area = $area."</table>','FGCOLOR,'#FFFFFF',BGCOLOR,'#000000','";
111     $area = $area."WIDTH,300,HAUTO,VAUTO);\" ";
112     $area = $area."onmouseout=\"return nd();\" ";
113     $area = $area."onclick=\"return false;\" ";
114     $area = $area."coords=\"\$xm,$ym,$xp,$yp\" href=\"\#\">\n";

```

```

115     // return <area> tag

116     return $area;

117 } // end MarkSpot


118 //-----
119 // CreateTTimagemap - create atooltip imagemap

120 function CreateTTimagemap($qresult,$map) {

121     // return an imagemap with an <area> tag
122     // for each row of the query results.

123     $hotSpot = ms_newPointObj();

124     $imagemap = "<map name=\"stores\">";

125     // scan the query results

126     for ( $i = 0; $i < count($qresult); $i++) {

127         $row = $qresult[$i];
128         $hotSpot->setXY($row[3],$row[2]);
129         $newarea = MarkSpot($i,$map->width,$map->height,
130                             $hotSpot,$map->extent,$row);
131         $imagemap = $imagemap."\n".$newarea;
132     }

133     // close the imagemap tag

134     $imagemap = $imagemap."\n</map>\n";

135     return $imagemap;

136 } // end CreateTTimageMap


137 //-----
138 // GetStoreMenu - returns string containing store menu

139 function GetStoreMenu($store_id) {

140     // Retrieve products/services menu for a store

```



```
141     @mysql_connect("localhost", "mysql", "password")
142         or die("Could not connect to MySQL server!");

143     @mysql_select_db("restaurant")
144         or die("Could not select database");

145     $query = "SELECT product.description FROM store, menu, product ";
146     $query = $query."WHERE store.id=menu.store_id ";
147     $query = $query."AND menu.product_id=product.id ";
148     $query = $query."AND store.id=$store_id";

149     $result = mysql_query($query);

150     // save each menu item in an array element

151     $i = 0;
152     while ( $row = mysql_fetch_array($result,MYSQL_NUM) ) {
153         $item[$i] = $row[0];
154         $i++;
155     }

156     // return menu as a string

157     return join("|",$item);

158 } // end GetStoreMenu

159 //-----
160 // GetStoreTable - returns array containing store table

161 function GetStoreTable() {

162     // Retrieve store table from MySQL database

163     @mysql_connect("localhost", "mysql", "password")
164         or die("Could not connect to MySQL server!");

165     @mysql_select_db("restaurant")
166         or die("Could not select database");

167     $query = "SELECT * FROM store";
168     $result = mysql_query($query);

169     // save each row of result in an array
```

```

170     $i = 0;
171     while ( $row = mysql_fetch_array($result,MYSQL_NUM) ) {
172         $qresult[$i] = $row;
173         $i++;
174     }

175     // return array of results

176     return $qresult;

177 } // end GetStoreTable


178 //-----
179 // AddPoints - add store locations to 'poi' map layer

180 function AddPoints ( $map, $qresult ) {

181     // Use lat/long info from query results to add points
182     // to the points-of-interest layer of the map
183     // shape index is set to the store-id
184     // (this requires a patched version of Mapscript)

185     $this_layer = $map->getLayerByName('poi');

186     $i = 0;
187     foreach($qresult as $row) {
188         $poi[$i] = ms_newPointObj();
189         $ln[$i] = ms_newLineObj();
190         $shp[$i] = ms_newShapeObj(MS_SHAPE_POINT);
191         $poi[$i]->setXY($row[3],$row[2]);
192         $ln[$i]->add($poi[$i]);
193         $shp[$i]->add($ln[$i]);
194         $shp[$i]->set(index, $row[0]);
195         $this_layer->addFeature( $shp[$i] );
196         $i++;
197     }

198     return;

199 } // end AddPoints


200 //-----
201 // NearbyStores - return a list of stores near click point

202 function NearbyStores($point,$map,$radius) {

```

```

203      // get query layer

204      $qlayer = $map->getLayerByName('poi');
205      $qlayer->set("tolerance",$radius);

206      // query the query layer - $radius is set in browser
207      // queryByPoint ignores TOLERANCE units using native map units
208      // instead - in this case decimal degrees. The number of miles
209      // per degree is (approximately of course) 69.04 therefore
210      // the correction from degrees to miles. This would have to
211      // change if TOLERANCEUNITS, the map or scale units change.

212      @$qlayer->queryByPoint($point, MS_MULTIPLE, $radius/69.04);
213      $numResults = $qlayer->getNumResults();

214      // we've got results, store id equals shape index

215      if ($numResults != 0) {
216          for ($i = 0; $i < $numResults; $i++) {
217              $query_result = $qlayer->getResult($i);
218              $StoreList[$i] = $query_result->shapeindex;
219          }
220      } else {

221          $StoreList = "";      // no results

222      }

223      return $StoreList;

224  } // end NearbyStores

225  //-----
226  // BuildResultTable - build HTML table of nearby stores

227  function BuildResultTable($nearby,$qresult) {

228      // assemble the table of nearby stores

229      $result_table = "<table border=1>\n<tr>";
230      $result_table = $result_table."<th>Store</th>";
231      $result_table = $result_table."<th>Address</th>";
232      $result_table = $result_table."<th>Latitude</th>";
233      $result_table = $result_table."<th>Longitude</th>";
234      $result_table = $result_table."<th>Phone</th>";
235      $result_table = $result_table."<th>Open</th>";

```

```

236     $result_table = $result_table."<th>Close</th>";
237     $result_table = $result_table."<th>Menu</th></tr>\n";

238     // there are stores nearby

239     if ( $nearby ) {
240         foreach ( $nearby as $store ) {
241             $row = $qresult[$store - 1];
242             $menu = GetStoreMenu($store);
243             $result_table = $result_table."<tr>";
244             for ( $j = 0; $j < 7; $j++ ) {
245                 $result_table = $result_table."<td>$row[$j]</td>";
246             }
247             $result_table = $result_table."<td>$menu</td></tr>\n";
248         }

249     // there are NO stores nearby

250     } else {
251         $result_table = $result_table."<tr><td colspan=8>No results</td></tr>";
252     }
253     $result_table = $result_table."</table>";

254     return $result_table;

255 } // end BuildResultTable

256 //-----

257 // Who are we

258 $script_name = "phpms_fifth.php";

259 // Define some default values to use
260 // before form variables are available

261 // path defaults

262 $map_path = "/home/mapdata/";
263 $map_file = "fifth.map";
264 $img_path = "/var/www/htdocs/tmp/";

```

```
265 // Navigation defaults

266 $zoomsize = 2;
267 $pan = "CHECKED";
268 $zoomout = "";
269 $zoomin = "";

270 // Displayed layer defaults

271 $hoods = "CHECKED";
272 $rivers = "CHECKED";
273 $streets = "CHECKED";
274 $majorstreets = "CHECKED";

275 // Map mode

276 $browse = "CHECKED";
277 $nquery = "";
278 $radius = 1;           // sets TOLERANCE for point query
279                        // TOLERANCEUNITS miles specified in mapfile

280 // Default click point

281 $clickx = 320;
282 $clicky = 240;
283 $clkpoint = ms_newPointObj();
284 $old_extent = ms_newRectObj();

285 // Default extent & maximum extent are the same

286 $extent = array(-97.384655, 49.697475, -96.877772, 50.077168);
287 $max_extent = ms_newRectObj();
288 $max_extent->setextent(-97.384655, 49.697475, -96.877772, 50.077168);

289 // Retrieve mapfile and create a map from it

290 $map = ms_newMapObj($map_path.$map_file);
```

```
291 // First time we're invoked use default variable and drop
292 // through to create map image for the first time else use
293 // form variables to pan and zoom before creating map image

294 if (( $_POST['img_x'] and $_POST['img_y'] ) or
295     $_POST['refresh'] or $_POST['left_x'] or
296     $_POST['right_x'] or $_POST['up_x'] or $_POST['down_x']) {

297     // Refresh button clicked, fake the map click

298     if ( $_POST['refresh'] ) {
299         $clickx = 320;
300         $clicky = 240;

301     // left arrow clicked - pan left 1/4 image width

302     } elseif ( $_POST['left_x'] ) {
303         $clickx = 160;
304         $clicky = 240;

305     // right arrow clicked - pan right 1/4 image width

306     } elseif ( $_POST['right_x'] ) {
307         $clickx = 480;
308         $clicky = 240;

309     // up arrow clicked - pan up 1/4 image height

310     } elseif ( $_POST['up_x'] ) {
311         $clickx = 320;
312         $clicky = 120;

313     // down arrow clicked - pan down 1/4 image height

314     } elseif ( $_POST['down_x'] ) {
315         $clickx = 320;
316         $clicky = 360;

317     // map was clicked, get the real coordinates
```

```

318     } else {
319         $clickx = $_POST['img_x'];
320         $clicky = $_POST['img_y'];
321     }

322     // Set the mouse click location (we need it to zoom)

323     $clkpoint->setXY($clickx,$clicky);

324     // mode or search radius may have changed, update 'em

325     if ($_POST['mode'] == "nquery") {
326         $nquery = "CHECKED";
327         $browse = "";
328     } else {
329         $nquery = "";
330         $browse = "CHECKED";
331     }
332     $radius = abs( $_POST['radius'] );

333     // Selected layers may have changed, reset HTML 'checks'

334     $layers = join(" ", $_POST['layer']);

335     if (preg_match("/hoods/", $layers)){
336         $hoods = "CHECKED";
337         $this_layer = $map->getLayerByName('hoods');
338         $this_layer->set('status', MS_ON);
339     } else {
340         $hoods = "";
341         $this_layer = $map->getLayerByName('hoods');
342         $this_layer->set('status', MS_OFF);
343     }

344     if (preg_match("/rivers/", $layers)){
345         $rivers = "CHECKED";
346         $this_layer = $map->getLayerByName('rivers');
347         $this_layer->set('status', MS_ON);
348     } else {
349         $rivers = "";
350         $this_layer = $map->getLayerByName('rivers');
351         $this_layer->set('status', MS_OFF);
352     }

```

```
353     if (preg_match("/majorstreets/", $layers)){
354         $majorstreets = "CHECKED";
355         $this_layer = $map->getLayerByName('majorstreets');
356         $this_layer->set('status', MS_ON);
357     } else {
358         $majorstreets = "";
359         $this_layer = $map->getLayerByName('majorstreets');
360         $this_layer->set('status', MS_OFF);
361     }

362     if (preg_match("/streets/", $layers)){
363         $streets = "CHECKED";
364         $this_layer = $map->getLayerByName('streets');
365         $this_layer->set('status', MS_ON);
366     } else {
367         $streets = "";
368         $this_layer = $map->getLayerByName('streets');
369         $this_layer->set('status', MS_OFF);
370     }

371     // since we were invoked by the form - retrieve previous map extent

372     if ( $_POST['extent'] ) {
373         $extent = split(" ", $_POST['extent']);
374     }

375     // Set the map to the extent retrieved from the form

376     $map->setExtent($extent[0],$extent[1],$extent[2],$extent[3]);

377     // Save this extent as a rectObj, we need it to zoom.

378     $old_extent->setextent($extent[0],$extent[1],$extent[2],$extent[3]);

379     // convert click point to geo coordinates before zoom or pan
380     // we need it for point query

381     list($qx,$qy) = img2map($map->width,$map->height,$clkpoint,$old_extent);
382     $qpoint = ms_newPointObj();
383     $qpoint->setXY($qx,$qy);
```



```

384      // Calculate the zoom factor to pass to zoomPoint method
385      // and setup the pan and zoom variables for web page

386      //  zoomfactor = +/- N
387      //  if N > 0 zooms in - N < 0 zoom out - N = 0 pan
388      //
389      $zoom_factor = $_POST['zoom'] * $_POST['zsize'];

390      if ($zoom_factor == 0) {
391          $zoom_factor = 1;
392          $pan = "CHECKED";
393          $zoomout = "";
394          $zoomin = "";
395      } elseif ($zoom_factor < 0) {
396          $pan = "";
397          $zoomout = "CHECKED";
398          $zoomin = "";
399      } else {
400          $pan = "";
401          $zoomout = "";
402          $zoomin = "CHECKED";
403      }
404      $zoomsize = abs( $_POST['zsize'] );

405      // Zoom in (or out) to clkpoint

406      if ( $_POST['mode'] == "browse" ) {
407          $map->zoomPoint($zoom_factor,$clkpoint,$map->width,
408              $map->height,$old_extent,$max_extent);
409      }
410  }

411  // Retrieve store table,
412  // add points to the points-of-interest layer,
413  // build tooltip imagemap

414  $qresult = GetStoreTable();
415  AddPoints( $map, $qresult );
416  $image_map = CreateTtImagemap($qresult,$map);

417  // The points-of-interest layer has been populated
418  // and can now be queried if in query mode

```

```
419 if ( $_POST['mode'] == "nquery" ) {
420     // find nearby stores
421     $nearby = NearbyStores($qpoint,$map,$radius);
422     // build HTML table of nearby stores
423     $result_table = BuildResultTable($nearby,$qresult);
424 }

425 // create unique names for map and reference images

426 $map_id = sprintf("%0.6d",rand(0,999999));
427 $image_name = "fifth".$map_id.".png";
428 $image_url="/tmp/".$image_name;
429 $ref_name = "fifthref".$map_id.".gif";
430 $ref_url="/tmp/".$ref_name;

431 // Draw and save map image

432 $image=$map->draw();
433 $image->saveImage($img_path.$image_name);

434 // Draw and save reference image

435 $ref = $map->drawReferenceMap();
436 $ref->saveImage($img_path.$ref_name);

437 // Get new extent of map (we'll save it in a form variable)

438 $new_extent = sprintf("%.3.6f",$map->extent->minx)." "
439     .sprintf("%.3.6f",$map->extent->miny)." "
440     .sprintf("%.3.6f",$map->extent->maxx)." "
441     .sprintf("%.3.6f",$map->extent->maxy);

442 // get the scale of the image to display on the web page

443 $scale = sprintf("%10d",$map->scale);
```

```

444 // Convert mouse click from image coordinates to map coordinates

445 list($mx,$my) = img2map($map->width,$map->height,$clkpoint,$old_extent);
446 $mx_str = sprintf("%.6f",$mx);
447 $my_str = sprintf("%.6f",$my);

448 $NavigateIE = HandleIE($_POST['mode'],$image_url);

449 // We're done, output the HTML form

450 ?>

451 <html>
452 <head>
453 <title>MapScript Fifth Map</title>

454 <script type="text/javascript">
455     var ol_textsize = "5px";
456     var ol_width = 300;
457 </script>
458 <script type="text/javascript" src="overlib.js">
459 <!-- overLib (c) Erik Bosrup --> </script>

460 </head>

461 <body bgcolor="#E6E6E6">

462     <!-- overLib needs this tag right after body-->

463     <div id="overDiv"
464         style="position:absolute; visibility:hidden; z-index:1000;">
465     </div>

466 <!-- image map stores tooltip info and displays it -->
467 <?php echo $image_map; ?>

468 <form method=post action="<?php echo $script_name;?>">
469     <table width="100%" border="1">
470         <tr><td width="60%" rowspan="6">
471             <table border="0">

472 <!-- Display up arrow if browser is IE-like -->
473         <tr><td align="center" colspan="3"><?php echo Arrow("up"); ?>
474             </td></tr>

```

```

475 <!-- Display left arrow if browser is IE-like -->
476 <tr><td valign="center"><?php echo Arrow("left"); ?>
477 </td>
478 <!-- Displays image as <img> or <input> depending on mode and browser -->
479 <td><?php echo $NavigateIE; ?>
480 </td>
481 <!-- Display right arrow if browser is IE-like -->
482 <td valign="center"><?php echo Arrow("right"); ?>
483 </td></tr>
484 <!-- Display down arrow if browser is IE-like -->
485 <tr><td align="center" colspan="3"><?php echo Arrow("down"); ?>
486 </td></tr>
487 </table>
488 </td>
489 <td width="40%" align="center" colspan="3">
490 <!-- Displays reference image -->
491 <img SRC="<?php echo $ref_url; ?>" width=300 height=225 border=1>
492 </td></tr>
493 <!-- Displays map scale -->
494 <tr><td align="left" colspan="3"><font size="-1">
495 Map scale:&nbsp; &nbsp; &nbsp; 1:<?php echo $scale; ?>
496 </font></td></tr>
497 <!-- Display click coordinates -->
498 <tr><td align="left" colspan="3"><font size="-1">
499 Click x,y:&nbsp; &nbsp; &nbsp; &nbsp; &nbsp; <?php echo $mx_str; ?>,
500 <?php echo $my_str; ?>
501 </font></td></tr>
502 <!-- Display map extent and save it as hidden variable -->
503 <tr><td align="left" colspan="3"><font size="-1">
504 <input type="hidden" name="extent"
505 value="<?php echo $new_extent; ?>">
506 Map Extent:&nbsp; <?php echo $new_extent; ?></font></td></tr>
507 <tr><td><B><center>Mode</center></B></td>
508 <td><B><center>Navigation</center></B></td>
509 <td><B><center>Layers</center></B></td></tr>

```

```

510 <!-- Select map mode -->
511 <tr><td rowspan="2">
512     <input type="radio" name="mode"
513         value="browse" <?php echo $browse; ?> >Browse<BR>
514     <input type="radio" name="mode"
515         value="nquery" <?php echo $nquery; ?> >Query<BR>
516     Search radius:<BR>
517     <input type="text" name="radius" size="4"
518         value="<?php echo $radius; ?>">(miles)

519 <!-- Navigation controls -->
520 <td align="left"><font size="-1">
521     <input type="radio" name="zoom"
522         value=0 <?php echo $pan; ?>> Pan<br>
523     <input type="radio" name="zoom"
524         value=1 <?php echo $zoomin; ?>> Zoom In<br>
525     <input type="radio" name="zoom"
526         value=-1 <?php echo $zoomout; ?>> Zoom Out<br>
527     <input type="text" name="zsize"
528         value="<?php echo $zoomsize; ?>" SIZE=2>Size<br>
529 <center>
530     <input type="submit" name="refresh" value="Refresh">
531 </center></td>

532 <!-- Layer selection -->
533 <td align="top">
534     <input type="checkbox" name="layer[]"
535         value="hoods" <?php echo $hoods; ?> >
536         'hoods<BR>
537     <input type="checkbox" name="layer[]"
538         value="rivers" <?php echo $rivers; ?> >
539         Rivers<BR>
540     <input type="checkbox" name="layer[]"
541         value="majorstreets" <?php echo $majorstreets; ?> >
542         Major streets<BR>
543     <input type="checkbox" name="layer[]"
544         value="streets" <?php echo $streets; ?> >
545         Streets</font>
546 </td></tr>
547 </table>
548 </form>

549 <!-- Display table of nearby stores in query mode -->
550 <?php echo $result_table; ?>

551 </body>
552 </html>

```



Utility Programs

The source distributions of MapServer and several of the libraries contain many useful utility programs. These allow you to create maps directly from a mapfile, investigate the contents of a DBF file, and even create shapefiles. This chapter will be devoted to describing how to use these programs and why you might want to use them.

The use of each program will be described, the various command-line parameters explained, and a few short examples provided. The chapter will begin with the utilities bundled with MapServer, and then proceed through the shapelib, GDAL, and finally OGR utilities.

MapServer

The MapServer utilities consist mostly of programs for producing images directly from mapfiles. While it's certainly possible to let MapServer create images that are displayed in a browser, it's sometimes more convenient to do it from the command line. All of the following programs are found in the main MapServer directory, and are built along with the MapServer CGI binary `mapserv`.

A description of each utility is presented, followed by the usage syntax. Some programs have complex options or employ new concepts. For these, a more extensive discussion is provided that describes the options and explains the new concepts.

shp2img

The `shp2img` utility reads a mapfile and creates a map image based on its contents. The mandatory mapfile name is specified by the `-m` switch (e.g., `-m somemap.map`, where the extension, usually `.map`, is required). All other parameters are optional.

```
usage: shp2img -m mapfile -o outputimage -e minx miny maxx maxy -t -l layers
        -i format -all_debug n -map_debug n -layer_debug layer_name n -p n -v
```

If the `outputimage` name isn't specified, then the image will be sent to `STDOUT` should the user wish to pipe the image to another program for further processing. `-e minx miny maxx maxy` sets the extent of the map image to be created—the default is the mapfile value. `-t` enables transparency. Layers to be rendered can be specified by following `-l` with a space-delimited list of layer names. `-i format` sets the `IMAGETYPE` to `format`, overriding the value specified in the mapfile. Use `-p n` to pause `n` seconds after the mapfile is read. `-v` displays the MapServer version number and the libraries available. Debug levels for the map and all layers are specified by

`-all_debug n`, while `-map_debug n` and `-layer_debug layer_name n` set the debug levels for the map and specified layer, respectively.

Tip If you have a problem and want to get the attention of MapServer developers, make things easy for them. The best MapServer debugging tool is `shp2img` because it removes web server complications. Using the simplest mapfile that produces the error, demonstrate that `shp2img` fails to create the expected map image before submitting an error report.

legend

The `legend` utility reads a mapfile and creates a legend image based on its contents. The output format (either GIF or PNG) depends on the version of the GD library against which MapServer was built. The mapfile extension is required.

usage: `legend mapfile outputimage`

scalebar

The `scalebar` utility reads a mapfile and creates a scale bar image based on its contents. The output format (either GIF or PNG) depends on the version of the GD library against which MapServer was built. The mapfile extension is required.

usage: `scalebar mapfile outputimage`

sortshp

The mapfile keyword `MAXFEATURES` can be used to determine the number of features drawn when a layer is rendered. However, this doesn't take into account the importance a particular feature might have. But, by sorting a shapefile based on one of its attributes, you can endow the sequence number of a feature with a significance that allows the value assigned to `MAXFEATURES` to choose the most important features to render. In this way, the `sortshp` utility can be used to sort a shapefile based on the value of an attribute.

Consider, for example, a region with thousands of lakes that range in area from a few acres to a thousand square miles or more. If the shapefile that represents these features is sorted according to lake area, and `MAXFEATURES N` is specified, then only the `N` largest lakes in the current extent will be rendered. This can have a significant impact on response time.

usage: `sortshp [-v] inputshapefile sortedshapefile item ascending|descending`

The optional parameter `-v` displays the MapServer version and the various libraries supported. `inputshapefile` identifies the shapefile to be sorted, and `sortedshapefile` is the name of the sorted shapefile (the shapefile extension, `.shp`, isn't specified). `item` is the attribute on which the sort is performed. The parameters `ascending` and `descending` specify the order of the sort.

sym2img

The `sym2img` utility reads a symbol file and creates the image of a symbol based on its contents. The output format (either GIF or PNG) depends on the version of the GD library against which MapServer was built. The mapfile extension is required.

usage: `sym2img mapfile outputimage`

shptree

In order to render a layer based on the contents of a shapefile, MapServer reads the file sequentially. MapServer must examine every feature to determine if it intersects the displayed extent. If it does, it's selected for rendering; if not, it's ignored. If the displayed extent is a small fraction of the shapefile's maximum extent, most features will be ignored—however, the entire file must *still* be scanned. When shapefiles are large, this can increase response time significantly. However, it's possible to create an efficient indexing scheme that allows MapServer to reduce the number of features that must be examined, therefore reducing response time. This indexing scheme is called a quadtree index.

If a quadtree-based spatial index is present, MapServer will take advantage of it. The utility program `shptree` is used to create a quadtree index for a shapefile. The base name of the index file is the same as the base name of the shapefile, but the index file extension is `.qix`.

usage: `shptree shapefilename [depth] [indexformat]`

The shapefile name is specified without the `.shp` extension. The optional parameter `depth` determines the maximum depth of the index. If omitted or set to 0, `shptree` will determine the depth. Some architecture issues arise from the different ways that different processor technologies store numerical quantities. The optional parameter `indexformat` resolves these issues by specifying the byte ordering to use: `NL` indicates that the least significant byte will be stored first, and `NM` indicates most significant byte first.

A quadtree is the hierarchical structure that arises from the recursive partitioning of a two-dimensional extent into increasingly smaller areas. At each level, the area from the previous level is divided into four new areas, then each of these is partitioned into four more areas, and so on. The process is terminated when each area is so small that it contains only a few features. Each element of the final partition, called a quad, is a rectangular extent that's uniquely defined by the sequence of supersets that contains it. This is shown schematically in Figure 10-1. In this case, the map contains three features: a small circle, a rectangle, and a large circle. The sequence of partitions is shown only for the small circle.

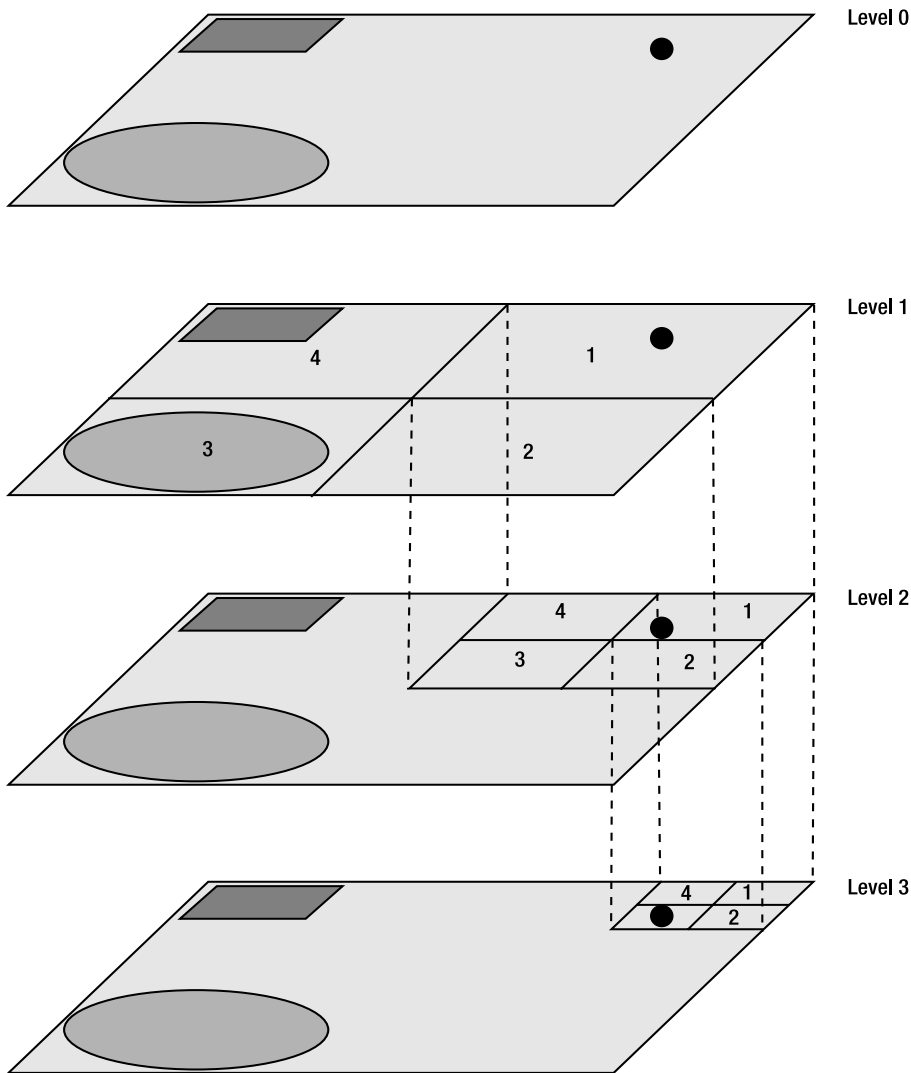


Figure 10-1. A conceptual view of the hierarchical extents that comprise a quadtree

shptreevis

If a quadtree index has been created for a shapefile, `shptreevis` will produce a another shapefile containing the quads generated by the index. This can be useful for understanding what the index is doing.

usage: `shptreevis inputfile outputfile`

`inputfile` is the name of the shapefile for which the quadtree index was created, and `outputfile` is the name of the new shapefile that will contain the quads. File extensions are required.

Note When feature density is high, quadtree depth and response time increase. A data set may not, however, be dense throughout—for example, some regions may contain many lakes while others contain only a few. You can display the quads used by the index to indicate how density varies across the extent. You can then use this knowledge to partition the data set into a collection of smaller extents called *tiles*, which, when used in conjunction with a quadtree index, can result in shorter response times. Tiling is discussed in the following section.

A quadtree index was created for the file `roads_type.shp` that was used in Chapter 9, and `shptreevis` used to create a shapefile containing the quads from that index with the following commands:

```
shptree roads_type.shp
shptreevis roads_type.shp quads.shp
```

The result is shown in Figure 10-2. The `roads_type` layer has been superimposed to demonstrate that the quads are large where features are sparse, and small where features are dense. As you can see, the northeast corner of the extent contains many road features, and the quad sizes are relatively small compared to the southeast corner (in which roads are few and quads are large).

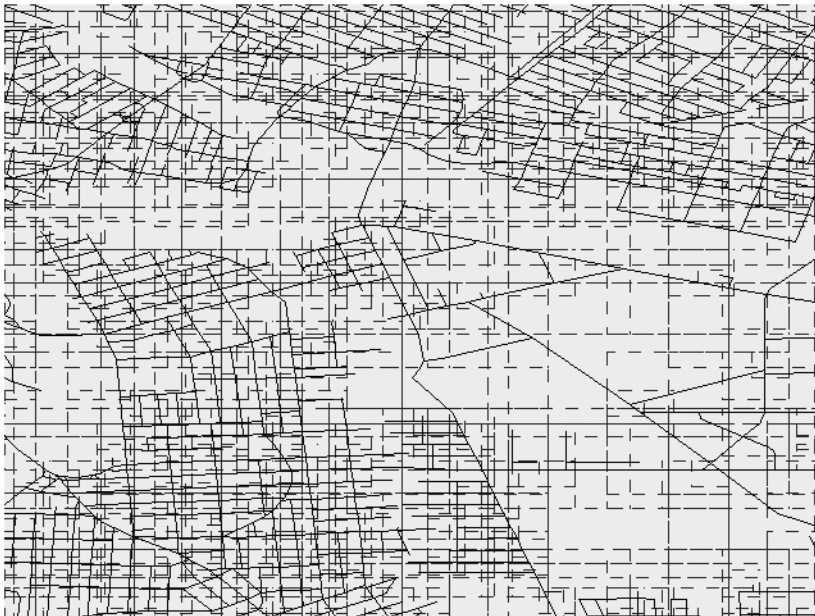


Figure 10-2. Roads (solid lines) superimposed on quads (dashed lines) generated by `shptreevis`, based on the quadtree index generated by `shptree`

tile4ms

It's frequently the case that the extent of a geographical area isn't covered by a single shapefile. This might arise because the spatial data has been acquired from several sources and no single shapefile provides complete coverage, or because you've used smaller shapefiles for faster rendering. The process of using multiple shapefiles to cover a geographical extent is called *tiling*, since the shapefiles cover the extent like tiles covering a floor—each shapefile is considered a tile. Tiles need not overlap and gaps can exist, but in any case, MapServer needs to be told how to access the multiple shapefiles.

This is done using the layer-level keywords `TILEINDEX` and `TILEITEM` in the mapfile. A tile index is simply a shapefile that contains a polygon feature for several shapefiles. Each polygon is a rectangle that has the same extent as the shapefile to which it refers. Associated with each of these rectangular features is an attribute containing the location of the shapefile.

In the mapfile, the keyword `TILEINDEX` specifies the name and location of the tile index shapefile, and the keyword `TILEITEM` specifies the name of the attribute containing the location of the tile. The utility program `tile4ms` is used to create the tile index shapefile.

```
usage: tile4ms shapefilelist tilefile [-tile-path-only]
```

The syntax is straightforward. A text file, `shapefilenames.txt`, is created, containing the following lines:

```
/home/mapdata/country1
/home/mapdata/country2
/home/mapdata/country3
```

Each line represents the path to a single shapefile. (Note that no file extensions are specified.) Executing the following command-line instruction will then create the tile index shapefile `/home/mapdata/countries`.

```
tilems ./shapefilenames.txt /home/mapdata/countries
```

The shapefile `countries` contains a rectangular extent for each of the three shapefiles identified in the file `shapefilenames.txt`. Its attribute table contains an attribute named `LOCATION`, the value of which is the location of the shapefile. You can now access these shapefiles in the mapfile by specifying the following values (the keyword `DATA` isn't used):

```
LAYER
    NAME 'somelayer'
    . . .
    TILEINDEX countries
    TILEITEM location
    . . .
END
```

Now, when MapServer renders this layer, it will only read the shapefiles that have extents (known from the tile index) that overlap the currently displayed extent of the map.

There's an optional command-line flag, `-tile-path-only` (note that only a single hyphen precedes the flag), which causes only the paths to the shapefiles to be stored in the location field, rather than the full file name. In this case, the value associated with the keyword `DATA` is appended to the path given by the `LOCATION` attribute of the tile index.

shapelib

The Shapefile C library provides a set of routines that can be used to write C programs that can read, write, and update shapefiles. Although using this library is beyond the scope of this book, I'd still like to mention that there are several very useful utility programs that accompany the distribution. These include programs to create and append records to shapefiles and the associated DBF files, programs to dump the contents of a shapefile to text, and programs to fix certain errors in shapefiles. In addition to this, there are several contributed applications that will be discussed later. Additional documentation is available online at <http://shapelib.maptools.org/shapelib-tools.html>.

There are other utilities available that can perform these tasks with additional functionality (notably `ogr2ogr`, which is described later). However, if you're not an experienced user, you may want to stick with the simple command-line parameters used by the shapelib utilities, and avoid the more powerful (and complex) utilities for now.

dbfcreate

The `dbfcreate` utility creates an empty DBF file.

usage: `dbfcreate filename -s fieldname width, -n fieldname width decimals, ...`

`filename` is the base name of the DBF file (the extension isn't specified). Following the file name is a list of comma-separated field definitions. String field definitions (introduced by `-s`) require a field name and width. Numeric field definitions (introduced by `-n`) require a field name, width, *and* number of decimal positions. The following command creates a DBF file named `example.dbf`, which contains two attribute values: a 10-character string `NAME` and a 10-digit `AREA`, which is a numeric field with 2 decimal places:

```
dbfcreate example -s NAME 10 -n AREA 10 2
```

dbfadd

The `dbfadd` utility adds a single record to a DBF file.

usage: `dbfadd filename fieldvalue1, fieldvalue2,...`

`filename` is the base name of the DBF file (the extension isn't specified). Following the file name is a comma-separated list of field values. If string values contain embedded blanks, they must be quoted (single or double quotes are both acceptable). The following command adds the string-valued attribute "A Square" and the numeric value 1.0 to the DBF file `example.dbf`, created previously:

```
dbfadd example "A Square" 1.0
```

dbfdump

The `dbfdump` utility dumps the contents of a DBF file to `STDOUT`.

usage: `dbfdump [-h] [-r] [-m] filename`

filename is the base name of the DBF file (the extension isn't specified). The optional command-line switches have the following effect: -h causes the field descriptions and field contents to be dumped, -r presents raw values (specifically, numeric fields are left unformatted), and -m dumps each field to a separate line. The effects are additive. Typing the following command (where example is the DBF file created earlier):

```
dbfdump -h -r -m example
```

produces the following output:

```
Field 0: Type=String, Title='NAME', Width=10, Decimals=0
Field 1: Type=Double, Title='AREA', Width=10, Decimals=2
Record: 0
NAME: A Square
AREA: 1.00
```

shpcreate

The shpcreate utility creates an empty shapefile.

usage: shpcreate filename featuretype

filename is the base name of the shapefile (the extension isn't specified). featuretype, which follows the file name, must be one of the following values: point, arc, polygon, or multipoint. arc is equivalent to the MapServer line layer type. The following command creates a polygon type shapefile named example.shp:

```
shpcreate example polygon
```

shpadd

The shpadd utility adds a single feature to a shapefile.

usage: shpadd filename x₁ y₁ x₂ y₂ ...

filename is the base name of the shapefile (the extension isn't specified). Following the file name is a list of space-delimited coordinate pairs. The number of coordinate pairs depends on the shapefile type. A point type shapefile requires a single pair. An arc requires at least two pairs. A polygon requires at least four pairs, with the last pair equal to the first. A multipoint shapefile requires a coordinate pair for each vertex. The following command adds a single polygon feature (a square) to the shapefile example.shp created earlier:

```
shpadd example 0 0 0 1 1 1 1 0 0 0
```

shpdump

The shpdump utility dumps the contents of a shapefile to STDOUT.

usage: shpdump [-validate] filename

filename is the base name of the shapefile (the extension isn't specified). The optional command-line switch `-validate` counts the number of features with invalid ring orderings. Consider the following command (where `example` is the shapefile created earlier):

```
shpdump -validate example
```

Executing this command produces the following output:

```
Shapefile Type: Polygon   # of Shapes: 1
File Bounds: (           0.000,           0.000,0,0)
              to (           1.000,           1.000,0,0)
Shape:0 (Polygon) nVertices=5, nParts=1
  Bounds:(           0.000,           0.000, 0, 0)
          to (           1.000,           1.000, 0, 0)
          (           0.000,           0.000, 0, 0) Ring
          (           0.000,           1.000, 0, 0)
          (           1.000,           1.000, 0, 0)
          (           1.000,           0.000, 0, 0)
          (           0.000,           0.000, 0, 0)
0 object has invalid ring orderings.
```

shprewind

A polygon consists of one or more rings, where a ring is defined as a closed, non-self-intersecting loop. According to the shapefile specification published by ESRI, when a ring is traversed in vertex order, the interior of the ring lies to the right. Occasionally, when points are added to or removed from a polygon, the vertex sequence can get reversed, which confuses software and leads to unexpected results. `shprewind` corrects the winding order.

```
usage: infilename outfilename
```

`infilename` is the base name of the defective shapefile, and `outfilename` is the base name of the corrected shapefile (extensions aren't specified).

dbfcat

The utility `dbfcat` appends the records in one DBF file to a second DBF file. The files must have the same number of fields. This is used in conjunction with `shpcat` (described later) to append the attributes and features of one shapefile to another.

```
usage: dbfcat [-f] [-v] fromfile tofile
```

`fromfile` is the base name of the source DBF and `tofile` is the base name of the destination DBF (extensions aren't specified). The optional command-line switches have the following effect: `-f` forces data conversion when the destination field types aren't the same as the source field types, or when there are null values in the source DBF. `-v` causes information about the field mappings to be displayed.

dbfinfo

The dbfinfo utility displays information about a DBF file.

usage: dbfinfo filename

filename is the base name of the DBF file (no extension is specified). Consider the following command (where example is the DBF file created earlier):

dbfinfo example

Executing this command produces the following output:

```
Info for example
2 Columns,  1 Records in file
      NAME      string  (10,0)
      AREA      float   (10,2)
```

shpcat

The shpcat utility appends the features in one shapefile to a second shapefile. The files must have the same shapefile type. This is used in conjunction with dbfcat (described earlier) to append the features and attributes of one shapefile to another.

usage: shpcat fromfile tofile

fromfile is the base name of the source shapefile and tofile is the base name of the destination shapefile (extensions aren't specified).

shpinfo

The shpinfo utility displays information about a shapefile.

usage: shpinfo filename

filename is the base name of the shapefile (no extension is specified). Typing the following command:

shpinfo example

produces the following output:

```
Info for example
Polygon(5),  1 Records in file
File Bounds: (              0,              0)
              (              1,              1)
```

shpcentrd

The `shpcentrd` utility computes the centroid of each polygon in a shapefile, and creates a point type shapefile containing each centroid as a feature.

usage: `shpcentrd infilename outfilename`

`infilename` is the base name of the polygon shapefile, and `outfilename` is the base name of the point shapefile containing the centroids (extensions aren't specified).

shpdxif

The `shpdxif` utility creates a DXF (Autocad Drawing Exchange Format) graphics file from a shapefile.

usage: `shpdxif filename`

`filename` is the full name of the shapefile—the extension *is* required. The base name of the shapefile with the extension `.dxf` appended becomes the name of the output file.

shpproj

The `shpproj` utility re-projects a shapefile using the Proj.4 library.

usage: `shpproj infilename outfilename (-i=inprojfile|-i="inparams"|-i=geographic)`
`(-o=outinfofile|-o="outparams"|o=geographic)`

`infilename` is the base name of the input shapefile, and `outfilename` is the base name of the *re-projected* shapefile. The projection parameters of the input shapefile can be read from `inprojfile` or specified on the command line and enclosed in quotes. If the input shapefile is unprojected (i.e., its coordinates are degrees of latitude and longitude), then the input projection is specified as `geographic`. The Appendix contains a section devoted to projections, which provides several `shpproj` examples.

GDAL/OGR

GDAL provides an abstract interface to various raster formats containing geospatial data. Such an interface allows a user to access a data set with a standard set of tools without having to be concerned with format-specific details. What's important here, however, is the library provided in the GDAL distribution that provides similar capabilities for accessing vector data sets. This library is called OGR (which at one time stood for **O**pen**G**IS Simple Features **R**eference **I**mplementation). This book hasn't made specific use of either GDAL or OGR, but OGR provides several utility programs that are useful even when the full functionality of the libraries isn't required. These programs were built as part of the GDAL build and can be found under the GDAL source directory `gdal-1.2.3/ogr/`. An in-depth description of OGR is beyond the scope of this book, so the descriptions given below will be less detailed than those in previous sections. The descriptions of the OGR utilities will be restricted, more or less, to their use with shapefiles. Additional documentation for the OGR utility programs described below is available online at www.gdal.org/ogr/ogr_utilities.html.

ogrinfo

The `ogrinfo` utility displays information about a data set in one of the OGR-supported formats. Although you've been concerned exclusively with shapefiles in this book, spatial data sets can come in many formats. Among the formats understood by OGR are the following: Arc/Info binary coverages; CSV (comma-separated variable) files; TIGER/Line files, and several spatially aware database engines including MySQL, Oracle Spatial, and PostgreSQL.

```
usage: ogrinfo [-ro] [-q] [-where restriction] [-spat xmin ymin xmax ymax]
              [-fid fid] [-sql statement] [-al] [-so] [--formats]
              datasource [layer [layer ...]]
```

There are several optional command-line switches: `-ro` opens the data set in read-only mode, `-al` lists everything (including the details of every feature), `-so` lists summary information (omitting feature details), and `-q` causes the program to operate in quiet mode, which suppresses some informational messages.

The switch `-where` introduces a simple attribute query, using the same syntax as a SQL WHERE clause. The restriction must be quoted, and any strings must be quoted as well. For example, to restrict the listing to records with a NAME attribute of "Bairdmore Blvd," you'd use the syntax `-where 'NAME="Bairdmore Blvd"'`. To report only records with a numeric attribute DIR greater than 0, you'd use `-where 'DIR>0'`.

The switch `-sql`, followed by a quoted SQL statement, causes `ogrinfo` to execute the statement and return the result. For example, `-sql "SELECT name,area FROM water WHERE area>10" water.shp` would return the name and area of all features in the shapefile `water.shp` with areas greater than 10.

You can also limit a report to those features found in a rectangular region by using the switch `-spat` and specifying the coordinates of the lower-left and upper-right corners of the bounding box. For example, `-spat -97.000 49.000 -96.000 50.000` would return only those features in the data set that lie within the rectangular region for which the lower-left and upper-right corners are (-97.000, 49.000) and (-96.000, 50.000), respectively.

A particular feature can be selected by using the `-fid` switch and the feature's record number. `-fid 49` would, for example, return the 49th feature in the data set.

You can use the `--format` switch (which takes no arguments) to produce a list of the formats that OGR can understand. Note that, unlike the others, the `format` switch requires two hyphens.

`datasource` is the name of the file (in the case of shapefiles) or directory (when accessing TIGER/Line data sets) containing the data set.

A list of layer names can be specified to restrict reporting to those layers—if this isn't done, all layers will be returned.

There are a number of useful tasks that `ogrinfo` can accomplish. Determining the attribute names available in a data set is one such task. If the data set consists of a shapefile, then the `shapelib` utility `dbfinfo` can be used. But if this isn't the case, then `ogrinfo` must be used. Specifying both `-al` and `-so` on the command line, as follows, will produce the output found in Listing 10-1:

```
ogrinfo -al -so roads_type.shp
```

Listing 10-1. *Using ogrinfo to find attribute names*

```
INFO: Open of 'roads_type.shp'
using driver 'ESRI Shapefile' successful.

Layer name: roads_type
Geometry: Line String
Feature Count: 15842
Extent: (-98.010101, 49.354854) - (-96.185323, 50.636213)
Layer SRS WKT:
(unknown)
ID: Integer (10.0)
LENGTH: Real (10.6)
DIR: Integer (19.0)
NAME: String (57.0)
SOURCE: String (5.0)
ORIGINALID: String (20.0)
DATE_ADD: String (10.0)
DATE_EDIT: String (10.0)
STATUS: Integer (6.0)
```

Frequently, the attribute that you wish to use to classify features will have many values. The following command pipes the output of `ogrinfo` to `grep`, which selects lines containing the string `STATUS`. These lines are then piped to `sort`, which (because of the `-u` switch) keeps only unique values and produces the report shown in Listing 10-2.

```
ogrinfo -al roads_type.shp | grep STATUS | sort u
```

Listing 10-2. *Using ogrinfo to show unique values*

```
STATUS (Integer) = 0
STATUS (Integer) = 1
STATUS: Integer (6.0)
```

Note that the last line in Listing 10-2 results from the definition of the attribute `STATUS` (shown in Listing 10-1), rather than the feature details that show the value of the `STATUS` attribute for each feature.

Another useful task `ogrinfo` can perform is finding all features in a data set that share an attribute value. You might want, for example, to access all the segments of a line feature, such as a road. Invoking `ogrinfo` with the following parameters will display all the features in the `roads_type.shp` shapefile that have a `NAME` attribute that equals `Bairdmore Blvd`:

```
ogrinfo -a1 -where 'NAME="Bairdmore Blvd"' roads_type.shp
```

Listing 10-3 shows the first few lines of that report.

Listing 10-3. *Using ogrinfo to explore the contents of a spatial data set*

```
INFO: Open of 'roads_type.shp'
using driver 'ESRI Shapefile' successful.
```

```
Layer name: roads_type
Geometry: Line String
Feature Count: 25
Extent: (-98.010101, 49.354854) - (-96.185323, 50.636213)
Layer SRS WKT:
(unknown)
ID: Integer (10.0)
LENGTH: Real (10.6)
DIR: Integer (19.0)
NAME: String (57.0)
SOURCE: String (5.0)
ORIGINALID: String (20.0)
DATE_ADD: String (10.0)
DATE_EDIT: String (10.0)
STATUS: Integer (6.0)
OGRFeature(roads_type):216
  ID (Integer) = 213
  LENGTH (Real) = 0.000000
  DIR (Integer) = 0
  NAME (String) = Bairdmore Blvd
  SOURCE (String) = LBIS
  ORIGINALID (String) = 1498
  DATE_ADD (String) = 20000201
  DATE_EDIT (String) = 20000201
  STATUS (Integer) = 0
  LINESTRING (-97.16952500 49.78065500,-97.16883400 49.77871000)
```

ogr2ogr

ogr2ogr is used to convert spatial data sets from one format to another. A summary of available formats is shown in Table 10-1. ogr2ogr can also be used to change the projection of a data set, to select features based on spatial or attribute criteria, or to reduce the number of feature attributes. Note that ogr2ogr can't write (or create) all the formats that it can read. This utility isn't used in this book, but it's included here since, sooner or later, you'll need to convert or manipulate a data set.

```
Usage: ogr2ogr [-skipfailures] [-append] [-update] [-f format]
              [-select field_list] [-where restriction] [-sql statement]
              [-spat xmin ymin xmax ymax] [-preserve_fid] [-fid FID]
              [-a_srs assigned_def] [-t_srs target_def]
              [-s_srs source_def]
              [ [-dsco NAME=VALUE] ...]
              output_dataset_name
              input_dataset_name
              [-lco NAME=VALUE] [-nln name] [-nlt type] layer [layer ...]]
```

The command-line switch `-skipfailures` causes `ogr2ogr` to continue processing if errors occur. The switch `-update` opens a data source in update mode, and the switch `-append` causes `ogr2ogr` to append records to an existing layer rather than create a new one.

`-f format` specifies the output format name. The format string must be enclosed in quotes—for example, `-f "ESRI Shapefile"`.

The attribute fields to be copied to the output data set can be specified with the parameter `-select field`, followed by a comma-delimited list of field names. If omitted, the default behavior copies all fields.

The syntax and use of parameters `-where`, `-sql`, and `-spat` are the same as their `ogrinfo` counterparts.

If a feature ID (i.e., record number) is specified with `-f FID`, then only that feature is selected. To my knowledge, the switch `-preserve_fid` has no effect on shapefiles.

The parameter `-dsco` defines a data set creation option. This option is specified as `NAME=VALUE`. The `VALUE` is format dependent, and the shapefile format you've used in this book has no such options. Similarly, the parameter `-lco NAME=VALUE` specifies the value of *layer* creation options. Shapefiles don't have these either.

An alternate name and a geometry type can be assigned to a new layer by using `-nln name` and `-nlt type`, where `name` is the name assigned to the layer and `type` is one of the following: `NONE`, `GEOMETRY`, `POINT`, `LINESTRING`, `POLYGON`, `GEOMETRYCOLLECTION`, `MULTIPOINT`, `MULTILINE`, `MULTIPOLYGON`, or `MULTILINESTRING`.

The projection of the output data set can be assigned or changed. The projection needs to be assigned, for example, when converting from a format that doesn't contain explicit projection information (such as the shapefile format). The value of the command-line parameter `-a_srs` specifies the assigned projection. The value `assigned_def` represents a WKT (well-known text) definition, the name of a file containing such a definition, or an EPSG code. The parameter `-t_srs` specifies the projection to which the output data set will be converted. The value `target_def` also represents a WKT, a file name, or an EPSG code. The value of parameter `s_srs` specifies the projection of the input data set or a projection that overrides the actual projection. The value `source_def` is used in the same manner as `assigned_def` and `target_def`.

Table 10-1. *OGR Formats*

Format Name	Can Be Created by ogr2ogr
Arc/Info binary coverage	No
CSV	Yes
DODS/OPeNDAP	No
ESRI Shapefile	Yes
FMEObjects Gateway	No
GML	Yes
IHO S-57 (ENC)	No
MapInfo	Yes
Microstation DGN	No
MySQL	No
ODBC	No
OGDI Vectors	No
Oracle Spatial	Yes
PostgreSQL	Yes
SDTS	No
SQLite	Yes
UK .NTF	No
US Census TIGER/Line	No
VRT (virtual data source)	No

Although the OGR library has many capabilities that allow MapServer to access a large number of additional vector formats, it has utility even when you restrict yourself to shapefile format, which is one of MapServer’s native formats. In the following example, features in a shapefile are extracted to another shapefile based on an attribute value—and in the process, the number of attributes is also reduced. (You might do this, for example, to reduce the size of the shapefile in order to improve response time.)

```
ogr2ogr -sql "SELECT id,name,status FROM roads WHERE status=1" /  
-f "ESRI Shapefile" newroads.shp roads.shp
```

The SQL SELECT statement will cause ogr2ogr to copy only those features for which status=1; and for each of the copied features, only the attributes id, name, and status will be created in the new shapefile.

ogrindex

MapServer possesses several native spatial data formats. The ESRI Shapefile format is used exclusively in this book. A description of tile indexes and how to create them was covered earlier in this chapter, but the functionality described was exclusive to shapefiles. The program `ogrindex` provides the same capabilities for data sets accessed through OGR. (Using OGR to access vector formats not otherwise supported by MapServer is beyond the scope of this book.)

```
usage: ogrindex [-lnum n]... [-lname name]...  
           [-f output_format] output_dataset src_dataset...
```

A shapefile consists of a single layer of a single type, but other vector formats allow a data source to include several layers, which may be of different geometrical types. `ogrindex` selects the layers for which a tile index is to be created—based on either the layer number (starting from zero) or the layer name. `-lnum n` specifies the layer number and `-lname name` specifies the layer name. The default selection is all layers.

Since a tile index is just a spatial data set that points to other data sets, `ogrindex` allows the user to determine the format using `-f format`, where `format` is one of OGR's supported output types (see Table 10-1). `output_dataset` is the name of the tile index to be created, and `src_dataset` is the name of the data set over which the tile index will be created. If the tile index doesn't exist, it's created; if it does, it's appended to.

Summary

This chapter has described several utility programs that are distributed with MapServer, `shapelib`, and `GDAL/OGR`. You should now be able to explore the contents of your spatial data sets, create and populate shapefiles, and modify shapefile contents. The next chapter is a reference to MapServer. All the mapfile keywords and their possible values, substitution strings, and CGI form variables will be described briefly, but comprehensively.



MapServer Reference

This chapter provides a comprehensive reference for mapfile keywords, CGI form variables, and substitution strings, current as of MapServer version 4.4.1. The elements of these three categories control the functionality, information content, and visual presentation provided by MapServer when operating in CGI mode. Before beginning this summary, I'd like to reiterate the purposes of these three important MapServer components.

- The mapfile specifies the graphical elements of the map and the spatial data sets from which they're constructed, and identifies the HTML templates that are used to display these graphical elements.
- CGI form variables embedded in the templates contain control and user information that enable MapServer's interactive capabilities.
- Substitution strings (also embedded in the templates) provide the connections that allow MapServer to populate the templates with graphical map elements and ancillary text information.

Many of these items haven't been used in the examples and applications presented so far, but it's hoped that the descriptions offered here will allow you to incorporate the functionality that they provide into your own applications without too much trouble.

Some of the functionality described in the following sections allows MapServer to interact with the broader world of GIS (for example, the use of the EPSG parameter files used by Proj.4). In these cases, the syntax is described, but detailed usage instructions aren't provided. A comprehensive treatment of all GIS techniques, facilities, and standards that might be accessible to a MapServer application would interfere with the primary goal, which is to provide an introduction to MapServer. If you recognize the acronyms and features, which are employed without explanation, and you know what to do with them, the text will indicate the syntax used to invoke the associated MapServer functionality. If they are, however, unknown to you, it would be more appropriate to seek out other resources to learn how (and why) to use them.

The canonical documents describing mapfile keywords, CGI variables, and template substitution strings are maintained by Jean-Francois Doyon, Jeff McKenna, Steve Lime, and Frank Koormann. They're available at <http://ms.gis.umn.edu/docs/reference>. The MapServer website is in transition, so it's a good idea to look at what's available at the older version, which can be found at <http://mapserver.gis.umn.edu/doc.html>.

Mapfile Keywords

The *mapfile* is the basic configuration tool used by MapServer operating in CGI mode. It possesses a hierarchical structure that conforms broadly to a hierarchical conception of a map. Specifically, a map (the mapfile) contains layers that are laid down in sequence, with each layer containing a particular set of features. Each layer is further broken down into classes, each of which represents a subset of the layer features that are rendered and labeled in the same style. Each style contains a set of components that specify the symbol used to render a feature, its color and size, and other characteristics.

It isn't possible to list the keywords in this glossary solely according to this hierarchical structure—embedding each object within its parent would complicate the presentation. On the other hand, listing every keyword in alphabetical sequence would lead to confusion, since keywords with different parents can share the same name but possess slightly different syntax. The organization I've chosen therefore represents a compromise. Each *simple* keyword (with no substructure) is presented at the level of the mapfile at which it's used, and its syntax is described at that point. However, each *structured* keyword, containing other keywords, is noted—but a detailed description of the contents of each is presented later in its own subsection. These sections and subsections are presented in alphabetical sequence.

Map Object

The map object isn't explicitly defined within the mapfile—it *is* the mapfile. It's the parent of all other mapfile objects and defines application characteristics that have global scope.

CONFIG

CONFIG [key][value]

Default: n/a

Specifies the values of environment variables for use by MapServer. For example, to set configuration parameters for some GDAL and OGR drivers, use CONFIG PROJ_LIB /somepath/, where PROJ_LIB is the key, and the value /somepath/ is the path to the library.

DATAPATTERN

DATAPATTERN [regular expression]

Default: n/a

Specifies a regular expression that's used to validate URL requests to change the value associated with the layer-level keyword DATA. If an attempt is made to change the value of DATA by inserting map_layername_data=some_path_and_filename into a URL, the regular expression must match the string some_path_and_filename.

DEBUG

DEBUG [on | off]

Default: off

Turns debugging on or off. If turned on, output is sent to STDERR (which is directed to the error log when using Apache or IIS) or the log file specified in the WEB object.

EXTENT

EXTENT [minx][miny][maxx][maxy]

Default: n/a

Specifies the extent of the map. A map extent is defined by the coordinates of the map's lower-left corner (minx, miny) and upper-right corner (maxx, maxy). Correctly calculating map scale requires that the floating-point coordinate values represent the units specified by the keyword UNITS. Errors in defining extents can result in blank maps, distorted maps, or errors. An extent must be defined, and it must be in the same coordinate system as the map's PROJECTION object (if one exists).

FONTSET

FONTSET [filename]

Default: n/a

Specifies the path (absolute or relative to the location of the mapfile) to the file defining the mapping from font aliases to TrueType font files.

IMAGECOLOR

IMAGECOLOR [int r][int g][int b]

Default: n/a

Specifies the background color of the map image. This color becomes transparent if TRANSPARENT is specified. The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

IMAGEQUALITY

IMAGEQUALITY [int N]

Default: 75

Specifies image quality for JPEG images. This usage is deprecated—use FORMATOPTION "QUALITY=n" in an OUTPUTFORMAT declaration instead.

IMAGETYPE

IMAGETYPE [gif | png | jpeg | wbmp | gtiff | swf | userdefined]

Default: n/a

Specifies the format of the output image. The image type can be one of the implicit OUTPUTFORMAT types recognized by MapServer (described later in this section), or it can be a user-defined type identified by its name as defined by the keyword NAME in the appropriate OUTPUTFORMAT declaration.

INTERLACE

INTERLACE [on | off]

Default: on

Turns image interlace on or off. This usage is deprecated—use FORMATOPTION "INTERLACE=on" in an OUTPUTFORMAT declaration instead.

LAYER

LAYER

Default: n/a

Indicates the start of a LAYER object.

LEGEND

LEGEND

Default: n/a

Indicates the start of a LEGEND object.

NAME

NAME [name]

Default: n/a

Specifies the name used to identify map output. An identification number (generated by concatenating the system time and process ID) is appended to this name to provide a unique ID.

PROJECTION

PROJECTION

Default: n/a

Indicates the start of a PROJECTION object.

QUERYMAP

QUERYMAP

Default: n/a

Indicates the start of a QUERYMAP object.

REFERENCE

REFERENCE

Default: n/a

Indicates the start of a REFERENCE object.

RESOLUTION

RESOLUTION [int N]

Default: 72

Specifies the resolution of the output display in pixels per inch. It's used in scale calculations only.

SCALE

SCALE [double N]

Default: n/a

Sets the scale of the map. Specifying SCALE 1000000 sets the map scale to 1:1,000,000. This value is usually generated by MapServer (which takes into account the extent, image size, units, and resolution) rather than specified by the application.

SCALEBAR

SCALEBAR

Default: n/a

Indicates the start of a SCALEBAR object.

SHAPEPATH

SHAPEPATH [path]

Default: n/a

Specifies the path to shapefiles. The value assigned to SHAPEPATH is prefixed to the data set specified by the layer-level keyword DATA.

SIZE

SIZE [int x][int y]

Default: n/a

Specifies the width and height of the map image in pixels.

STATUS

STATUS [on | off]

Default: on

Specifies whether the map image is created.

SYMBOL

SYMBOL

Default: n/a

Indicates the start of a SYMBOL object. Symbols can be defined in the mapfile itself or moved to a file identified by the keyword SYMBOLSET.

SYMBOLSET

SYMBOLSET [filename]

Default: n/a

Contains symbol definitions. These can be accessed by the symbol name or sequence number of the symbol, starting at 1. A value of 0 indicates the default symbol. Default symbols are single pixels for point features, 1-pixel-wide lines for line features, and solid fills for polygon features.

TEMPLATEPATTERN

TEMPLATEPATTERN [regular expression]

Default: n/a

Specifies a regular expression that's used to validate URL requests to change the value associated with the layer-level keyword TEMPLATE. If an attempt is made to change the value of TEMPLATE by inserting `map_layername_template=some_path_and_filename` into a URL, the regular expression must match the string `some_path_and_filename`.

TRANSPARENT

TRANSPARENT [on | off]

Default: off

Makes the background color of the map transparent. This usage is deprecated—set transparency in an OUTPUTFORMAT declaration instead.

UNITS

UNITS [feet | inches | kilometers | meters | miles | dd]

Default: n/a

Specifies map distance units. This keyword affects scale calculations and scale bars. *dd* indicates decimal degrees.

WEB

WEB

Default: n/a

Indicates the start of a WEB object.

CLASS Object

The CLASS object determines the appearance and labeling properties of features. Every layer must specify one or more classes. A CLASS object is introduced by the keyword CLASS and terminated by the keyword END.

BACKGROUND_COLOR

BACKGROUND_COLOR [int r][int g][int b]

Default: n/a

Specifies the color to be used to render non-transparent symbols. The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

COLOR

COLOR [int r][int g][int b]

Default: 0 0 0

Specifies the color to be used to render features. The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

DEBUG

DEBUG

Default: n/a

Turns class debugging on. The debug output is sent to STDERR or the log file specified in the WEB object.

EXPRESSION

EXPRESSION [string]

Default: n/a

Specifies the expression to evaluate to determine whether a feature should be included in the class. The value assigned to `EXPRESSION` represents one of the following types of expressions: a string comparison, a regular expression, a logical expression, or a string function.

A string comparison matches the contents of the attribute identified by the keyword `CLASSITEM` with `string`. The comparison is case sensitive. If an exact match is found, the feature is included in the class. If `string` contains embedded blanks or special characters (like tabs or new lines), it must be quoted.

A regular expression matches the contents of the attribute identified by the keyword `CLASSITEM` with the regular expression specified by `string`. The regular expression must be delimited by slashes, (e.g., `/regular expression/`).

A logical expression consists of a parenthesis-delimited combination of attribute names delimited by square brackets; the Boolean operators `AND` and `OR`; the numeric comparison operators `=`, `>`, `<`, `<=`, `>=`, and `!=`; the string comparison operators `eq`, `lt`, `gt`, `le`, `ge`, `eq`, and `ne`; and comparison values. For example, `EXPRESSION ([area]>100)` will select features with area *greater* than 100, and `EXPRESSION ([status]!=0)` will select records with status *not equal* to 0. String-valued attribute names and comparison values must be quoted. For example, `EXPRESSION ('[type]' ne 'river')` will select only those features with type not equal to river.

Currently, the only string function supported by MapServer is `length()`, which computes the length of a string-valued attribute. `EXPRESSION (length('[name]') < 2)` selects features with short names.

JOIN

JOIN

Default: n/a

Indicates the start of a JOIN object. Current documentation erroneously includes JOIN objects in the CLASS object, and states that JOIN objects are “defined within a QUERY object.” But this is *not* the case. JOIN objects must be defined at the layer level.

LABEL

LABEL

Default: n/a

Indicates the start of a LABEL object.

MAXSCALE

MAXSCALE [double N]

Default: n/a

Specifies the maximum scale at which the CLASS will be rendered.

MAXSIZE

MAXSIZE [int N]

Default: 50

Specifies the maximum size (in pixels) at which a symbol will be drawn.

MINSCALE

MINSCALE [double N]

Default: n/a

Specifies the minimum scale at which the CLASS will be rendered.

MINSIZE

MINSIZE [int N]

Default: 0

Specifies the minimum size (in pixels) at which a symbol will be drawn.

NAME

NAME [string]

Default: n/a

Specifies the name for the class for use in the legend. If a name isn't specified, the feature will still be drawn, but it won't be displayed in the legend.

OUTLINECOLOR

OUTLINECOLOR [int r][int g][int b]

Default: n/a

Specifies the outline color (for polygons only). The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

OVERLAYBACKGROUNDCOLOR

OVERLAYBACKGROUNDCOLOR [int r][int g][int b]

Default: n/a

Specifies the color used to render overlay symbols. The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

OVERLAYCOLOR

OVERLAYCOLOR [int r][int g][int b]

Default: n/a

Specifies the color to be used for drawing features with an overlay symbol. The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

OVERLAYMAXSIZE

OVERLAYMAXSIZE [int N]

Default: 50

Specifies the maximum size (in pixels) that an overlay symbol can be drawn.

OVERLAYMINSIZE

OVERLAYMINSIZE [int N]

Default: 0

Specifies the minimum size (in pixels) that an overlay symbol can be drawn.

OVERLAYOUTLINECOLOR

OVERLAYOUTLINECOLOR [int r][int g][int b]

Default: n/a

Specifies the outline color of an overlay symbol (for polygons only). The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

OVERLAYSIZE

OVERLAYSIZE [int N]

Default: vector and ellipse types: the range of y values; pixmaps: the vertical size of the image; TrueType font symbols: 1

Specifies the height (in pixels) of an overlay symbol.

OVERLAYSYMBOL

OVERLAYSYMBOL [integer | string | filename]

Default: 0

Specifies the overlay symbol used for drawing features, identified by name or number. The name is the value associated with the keyword NAME in the SYMBOL definition. The number is the sequence number of the symbol, starting at 1. The path (absolute or relative to the mapfile) to a file containing a GIF or PNG image can be specified.

SIZE

SIZE [int N]

Default: vector and ellipse types: the range of y values; pixmaps: the vertical size of the image; TrueType font symbols: 1
Specifies the height of a symbol (in pixels).

STYLE

STYLE

Default: n/a
Indicates the start of a STYLE object.

SYMBOL

SYMBOL [integer | string | filename]

Default: 0
Specifies the symbol to use for drawing features, identified by name or number. The name is the value associated with the keyword NAME in the SYMBOL definition. The number is the sequence number of the symbol, starting at 1. The path (absolute or relative to the mapfile) to a file containing a GIF or PNG image can be specified.

TEMPLATE

TEMPLATE [file | url]

Default: n/a
Specifies the name of the HTML file containing substitution strings used to display individual query results. Instead of a file name, a string representing a URL containing substitution strings may be specified. In one of the single query modes (for example, QUERY, FEATUREQUERY, etc.) MapServer will report a query result by replacing any substitution strings and then redirecting the browser to the modified URL.

TEXT

TEXT [string]

Default: n/a
Specifies the text to be used to label features in a class. TEXT can be used instead of the value of the attribute specified by LABELITEM. Bracket-delimited attribute names enclosed in parentheses can be used to concatenate attributes. For example, TEXT ([NAME]:[ROADTYPE]) would label features with the feature's NAME and ROADTYPE, separated by a colon. If both LABELITEM and TEXT have been specified, TEXT takes precedence.

FEATURE Object

The **FEATURE** object is used to define an inline feature by specifying the coordinates of its vertices. It's introduced by the keyword **FEATURE** and terminated by the keyword **END**. For a layer containing an inline feature, MapServer will ignore any other data source specified in the layer.

POINTS

POINTS $x_1 y_1 x_2 y_2 \dots x_n y_n$ **END**

Default: n/a

Specifies a series of coordinate pairs representing the vertices of a shape. In a polygon layer, the first and last vertices must be the same.

TEXT

TEXT [string]

Default: n/a

Specifies the text string used to label a feature.

GRID Object

The **GRID** object is used to define a map grid within a layer. It begins with the keyword **GRID** and is terminated by the keyword **END**. A layer containing a **GRID** object requires no **DATA** source and must be of **TYPE** `line`, but is otherwise unexceptional. A single class is defined to specify symbol, label, and color. If an output projection has been defined, the same projection must be defined in the **GRID** object. There are two ways to control the number of grid lines drawn. You can use **MAXARCS** and **MINARCS** to specify a range of acceptable values, or you can use **MAXINTERVAL** and **MININTERVAL** to specify the interval between grid lines.

LABELFORMAT

LABELFORMAT [DDMM | DDMMSS]

Default: decimal representation of SRS (Spatial Reference System)

Specifies the format of the grid labels as degrees and minutes (DDMM) or degrees, minutes, and seconds (DDMMSS). The strings are case sensitive and uppercase is mandatory.

MAXARCS

MAXARCS [double]

Default: n/a

Specifies the maximum number of arcs to be drawn.

MAXINTERVAL

MAXINTERVAL [double]

Default: n/a

Specifies the maximum interval between grid lines.

MAXSUBDIVIDE

MAXSUBDIVIDE [double]

Default: n/a

Specifies the maximum number of segments used to render a grid line.

MINARCS

MINARCS [double]

Default: n/a

Specifies the minimum number of arcs to be drawn.

MININTERVAL

MININTERVAL [double]

Default: n/a

Specifies the minimum interval between grid lines.

MINSUBDIVIDE

MINSUBDIVIDE [double]

Default: n/a

Specifies the minimum number of segments used to render a grid line.

JOIN Object

The JOIN object specifies how an external DBF table will be joined to a shapefile attribute table for query purposes. It's introduced by the keyword JOIN and terminated by the keyword END.

When a feature is included in a query result set, and a join has been defined for that layer, the joined table is scanned for records for which the TO attribute equals the FROM attribute of the selected feature. There are two types of join: one-to-one and one-to-many. The results from each type are handled differently.

If a one-to-one join has been defined, then items from the matching record can be reported in the class- or layer-level query template. Item values are accessed by means of a substitution string composed of the concatenated join name, an underscore character, and the item name from the external table (e.g., [test-join_item] retrieves the value of item from the external table defined in test-join).

If a one-to-many join has been defined, then there's possibly more than one matching record. In this case, the result isn't reported directly in the query template, but in a join template defined in the JOIN object itself. The query template is still used to format items from the attribute table—but now, only a reference to the join is included, instead of substitution strings representing external table items. This reference is the concatenation of the string "join", the underscore character, and the join name (e.g., for test-join, the reference in the query template would be [join_test-join]).

The template defined in the JOIN object can only contain references to items in the external table—not the attribute table. The format of the substitution strings used to access these items is identical to the one-to-one format.

The join template is processed once for each matching record. The results are concatenated, forming a space-delimited list. This list is returned and substituted for the join reference (e.g., [join_test-join]) in the query template.

Although current documentation states that joins are defined within a QUERY object, this is *not* the case—joins are defined within a LAYER object.

FROM

FROM [itemname]

Default: n/a

Specifies the name of the item in the shapefile that will be used for the join.

NAME

NAME [string]

Default: n/a

Specifies the join name. This name is used to reference the join from the template and must be unique.

TABLE

TABLE [filename]

Default: n/a

Specifies the full path to the external dBase file that will be joined to the attribute table.

TEMPLATE

TEMPLATE [filename]

Default: n/a

Specifies the template used for reporting results from one-to-many joins. Any substitution strings that reference table items must reference items in the joined table only.

TO

TO [itemname]

Default: n/a

Specifies the name of the join item in the table to be joined.

TYPE

TYPE [multiple | single]

Default: single

Specifies whether the join type is one-to-one (single) or one-to-many (multiple).

LABEL Object

The LABEL object defines a text string or symbol used to label a feature. It begins with the keyword LABEL and is terminated by the keyword END.

ANGLE

ANGLE [auto | double N]

Default: n/a

Specifies the angle (in decimal degrees) at which a label will be drawn. An angle of 0 will cause the label to be drawn parallel to the bottom of the map. For line layers only, the value auto causes the label to be aligned with the feature.

ANTIALIAS

ANTIALIAS [true | false]

Default: false

Causes the label text to be antialiased.

BACKGROUNDCOLOR

BACKGROUNDCOLOR [int r][int g][int b]

Default: no background

Specifies the color of a background rectangle used to highlight a label. The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

BACKGROUNDSHADOWCOLOR

BACKGROUNDSHADOWCOLOR [int r][int g][int b]

Default: no background shadow

Specifies the shadow color of a background rectangle. The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

BACKGROUNDSHADOWSIZE

BACKGROUNDSHADOWSIZE [int x][int y]

Default: 1 1

Specifies the offset of the background shadow in pixels.

BUFFER

BUFFER [int N]

Default: 0

Specifies (in pixels) the amount of space that's left around text labels. BUFFER is available for cached labels only.

COLOR

COLOR [int r][int g][int b]

Default: 0 0 0

Specifies the color used to render the label text. The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

ENCODING

ENCODING [string]

Default: n/a

Specifies the encoding format for the label. If the format specified by the string isn't supported, then the label won't be drawn.

FONT

FONT [name]

Default: none

Specifies the alias of the font used to label a feature. MapServer translates the alias to the corresponding path found in the file specified by the keyword FONTSET.

FORCE

FORCE [true | false]

Default: false

Forces a label to be rendered even if it collides with a label that has already been rendered. FORCE is available for cached labels only.

MAXSIZE

MAXSIZE [int N]

Default: 256

Specifies the maximum font size (in pixels) for scaled labels.

MINDISTANCE

MINDISTANCE [int N]

Default: n/a

Specifies the minimum distance (in pixels) between identical labels of features in the same class.

MINFEATURESIZE

MINFEATURESIZE [int N | auto]

Default: n/a

Specifies the minimum size (in pixels) at which a feature will be labeled. For line features, this is the length; for polygons, this is the smallest dimension of the bounding box. If MINFEATURESIZE is set to auto, only features larger than their labels will be labeled. MINFEATURESIZE is only available for cached labels.

MINSIZE

MINSIZE [int N]

Default: 4

Specifies the minimum font size (in pixels) for scaled labels.

OFFSET

OFFSET [int x][int y]

Default: 0 0

Specifies the offset (in pixels) of the lower-left corner of the label from the label point.

OUTLINECOLOR

OUTLINECOLOR [int r][int g][int b]

Default: n/a

Specifies the color used to create a 1-pixel-wide outline around the label text. The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

PARTIALS

PARTIALS [true | false]

Default: true

Renders partial labels for labels that would otherwise extend beyond the edge of the map.

POSITION

POSITION [ul | uc | ur | cl | cc | cr | ll | lc | lr | auto]

Default: lc

Specifies the position of the label with respect to the label point. Points and polygons can use the eight outer positions, but not the center position, cc. Lines can use only uc or lc. If POSITION is set to auto, MapServer will attempt to position the label such that it doesn't collide with previously drawn labels. If such a position can't be found, the label won't be drawn unless the value of the keyword FORCE is true. Auto-placement is available for cached labels only.

SHADOWCOLOR

SHADOWCOLOR [int r][int g][int b]

Default: no shadow

Specifies the shadow color. If SHADOWSIZE is specified but SHADOWCOLOR isn't, no shadow is created. The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

SHADOWSIZE

SHADOWSIZE [x][y]

Default: 1 1

Specifies the offset (in pixels) of the shadow behind the label text.

SIZE

SIZE [int N] | [tiny | small | medium | large | giant]

Default: n/a

Specifies the size of label text. TrueType label size is specified in pixels with an integer value. The size of a bitmapped label is specified with a value of tiny, small, medium, large, or giant.

TYPE

TYPE [bitmap | truetype]

Default: bitmap

Renders labels using either bitmapped or TrueType fonts.

WRAP

WRAP [character]

Default: n/a

Specifies the character that will cause the label to wrap and create a multiline label.

LAYER Object

Elements of the LAYER object determine what spatial data is to be rendered and how it's to be classified. It begins with the keyword LAYER and is terminated by the keyword END. Layers are drawn in the order found in the mapfile, and subsequent layers are rendered on top of those rendered earlier. Care must be taken to avoid obscuring previously rendered features.

CLASS

CLASS

Default: n/a

Indicates the start of a CLASS object.

CLASSITEM

CLASSITEM [attribute]

Default: none

Specifies the name of the attribute used to classify features when the keyword EXPRESSION is used to perform string or regular expression comparisons.

CONNECTION

CONNECTION [string]

Default: n/a

Specifies the connection string used to retrieve data from a remote database.

An *SDE* connection string is a comma-delimited list containing hostname, instancename, databasename, username, and password.

A *PostGIS* connection string consists of "user=username password=***** databasename=dbname host=hostname port=5432".

An *Oracle* connection string consists of "user/pass[@db]".

CONNECTIONTYPE

CONNECTIONTYPE [local | sde | ogr | postgis | oraclespatial | wms]

Default: local

Specifies the type of connection.

DATA

DATA [filename] | [sde parameters][postgis table/column][oracle table/column]

Default: n/a

Specifies the path to the shapefile (without extension) relative to the location specified by the keyword SHAPEPATH or relative to the location of the mapfile. When accessing SDE layers, sde parameters specifies the layer name and geometry column of the database as a quoted string in the form "layername,geometry". For PostGIS layers, postgis table/column specifies the geometry column and table name as a quoted string in the form "columnname from tablename". Accessing an Oracle database requires the specification of an Oracle-compliant query, such as "shape FROM table".

DEBUG

DEBUG [on | off]

Default: off

Turns layer debugging on. Debug output is sent to STDERR (frequently directed to the error log of the web server) or the log file specified in the WEB object.

DUMP

DUMP [true | false]

Default: false

Returns data in GML or raw raster format (used for WFS and WCS access if these features are enabled).

FEATURE

FEATURE

Default: n/a

Indicates the start of an inline FEATURE object.

FILTER

FILTER [string]

Default: none

Causes MapServer to perform attribute filtering of a layer when spatial filtering is done, but before evaluation of class expressions. string is a regular expression for OGR and shapefile access, and it's a SQL WHERE clause for spatial databases. This is typically used to filter out null attributes or shapes from a query.

FILTERITEM

`FILTERITEM [attribute]`

Default: n/a

Specifies the name of an attribute whose value is compared to the regular expression specified by the keyword `FILTER`. `FILTERITEM` is available for OGR and shapefile access only.

FOOTER

`FOOTER [filename]`

Default: n/a

Specifies the name of the HTML file displayed after all query results for a layer have been presented. Used for multi-result query modes.

GRID

`GRID`

Default: n/a

Indicates the start of a `GRID` object.

GROUP

`GROUP [name]`

Default: n/a

Specifies the name of the group to which a layer belongs. The group name can be used instead of the layer name as the value of the CGI form variable `layer` (or `layers`), in order to allow multiple layers to be turned off and on together.

HEADER

`HEADER [filename]`

Default: n/a

Specifies the name of the HTML file displayed before any query results for the layer have been presented. `HEADER` is used for multi-result query modes.

JOIN

`JOIN`

Default: n/a

Indicates the start of a `JOIN` object. Current documentation includes joins in the `CLASS` object and states that joins are “defined within a query object.” But this is *not* the case—joins must be defined at the layer level.

LABELANGLEITEM

LABELANGLEITEM [attribute]

Default: n/a

Specifies the name of the attribute whose value (in degrees) is used to set the angle at which the label is rendered. An angle of 0 will cause the label to be drawn parallel to the bottom of the map.

LABELCACHE

LABELCACHE [on | off]

Default: on

Turns the label cache on or off. If set to off, then labels are drawn when the features are drawn. Otherwise, the labels are cached and drawn after all layers have been drawn.

LABELITEM

LABELITEM [attribute]

Default: n/a

Specifies the name of the attribute whose value is used to label a feature.

LABELMAXSCALE

LABELMAXSCALE [double N]

Default: none

Specifies the maximum scale at which labels for a layer will be rendered.

LABELMINSCALE

LABELMINSCALE [double N]

Default: none

Specifies the minimum scale at which labels for a layer will be rendered.

LABELREQUIRES

LABELREQUIRES [expression]

Default: none

Controls the drawing of labels for a layer based on the status of other layers, where *expression* is a quoted Boolean string. Layer names are enclosed in square brackets, and the operators AND and OR are available. If the layer is on, the bracketed name is replaced with 1—otherwise, it's replaced by 0. If the expression evaluates to 1, the layer will be labeled—otherwise, labels won't be drawn. For example, LABELREQUIRES "[interstates] AND [roads]" causes features in the current layer to be labeled when the STATUS of both the interstates layer and the roads layer is on.

LABELSIZEITEM

LABELSIZEITEM [attribute]

Default: none

Specifies the name of the attribute whose value (in pixels) specifies the size at which the label will be rendered.

MAXFEATURES

MAXFEATURES [integer]

Default: all features

Specifies the maximum number of features that will be rendered in a layer.

MAXSCALE

MAXSCALE [double N]

Default: n/a

Specifies the maximum scale at which a layer will be rendered.

METADATA

```
METADATA
    title "A title"
    author "Someone"
END
```

Default: n/a

Allows data to be stored as name-value pairs, accessible by means of template tags. In the example shown, the values of title and author are available as [title] and [author].

MINSCALE

MINSCALE [double N]

Default: n/a

Specifies the minimum scale at which a layer will be rendered.

NAME

NAME [string]

Default: none

Specifies the name of the layer (maximum 20 characters). This name is used as the value of the CGI form variable layer to allow the layer to be turned on and off interactively.

OFFSITE

OFFSITE [int r][int g][int b]

Default: n/a

Sets the transparent color for raster layers. The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

POSTLABELCACHE

POSTLABELCACHE [true | false]

Default: false

Specifies that a layer will be drawn after all the labels in the cache have been drawn.

PROCESSING

PROCESSING [string]

Default: n/a

Specifies a processing directive for a layer. For raster layers processed by GDAL, SCALE, BANDS, and DITHER are available. For more information, consult “Raster Data in MapServer 4.4,” at http://ms.gis.umn.edu/docs/howto/raster_data.

PROJECTION

PROJECTION

Default: n/a

Indicates the start of a PROJECTION object.

REQUIRES

REQUIRES [expression]

Default: renders the layer

Controls the drawing of a layer based on the status of other layers, where expression is a quoted Boolean string. Layer names are enclosed in square brackets, and the operators AND and OR are available. If the layer is on, the bracketed name is replaced with 1—otherwise, it’s replaced by 0. If the expression evaluates to 1, the layer will be rendered—if not, the layer won’t be drawn. For example, REQUIRES "[hydro] AND [poi]" causes the current layer to be drawn when the STATUS of both the hydro layer and the poi layer is on.

SIZEUNITS

SIZEUNITS [pixels | feet | inches | kilometers | meters | miles]

Default: pixels

Sets the units of the CLASS object SIZE, which specifies the size of symbols used to draw features.

STATUS

STATUS [on | off | default]

Default: n/a

Specifies whether a layer will be displayed and whether it can be toggled on or off. `default` specifies that a layer will always be displayed; `on` specifies that a layer will be displayed, but can be turned off; and `off` specifies that a layer won't be displayed, but can be turned on.

STYLEITEM

STYLEITEM [attribute]

Default: n/a

Specifies the name of the attribute whose value determines how a feature will be rendered. This functionality is considered experimental in version 4.4.

SYMBOLSCALE

SYMBOLSCALE [double N]

Default: n/a

Specifies the scale at which a symbol appears at its full size. Symbols are scaled within the range specified by `MINSIZE` and `MAXSIZE`.

TEMPLATE

TEMPLATE [file | url]

Default: n/a

Specifies the name of the HTML file used to display individual query results. It's the `LAYER` equivalent of the class-level `TEMPLATE`. It's used instead of separate identical templates for each class.

TILEINDEX

TILEINDEX [filename]

Default: n/a

Specifies the name of the tile index file for the layer.

TILEITEM

TILEITEM [attribute]

Default: location

Specifies the name of the item in the tile index that contains the location of a tile.

TOLERANCE

TOLERANCE [double N]

Default: 3 pixels

Specifies the search radius or sensitivity for queries. Units are specified by the keyword TOLERANCEUNITS. To restrict a polygon search to points within the polygon, set TOLERANCE to 0.

TOLERANCEUNITS

TOLERANCEUNITS [pixels | feet | inches | kilometers | meters | miles | dd]

Default: pixels

Specifies units of TOLERANCE. dd represents decimal degrees.

TRANSFORM

TRANSFORM [true | false]

Default: true

Transforms spatial data from map coordinates to image coordinates for rendering purposes. If TRANSFORM false is set, however, no transformation is performed. In such a case, the features in a shapefile that has been created in image coordinates will always be drawn at the same location on the map. This is useful for placing logos and other fixed items in a map.

TRANSPARENCY

TRANSPARENCY [int | alpha]

Default: n/a

Sets opacity for a layer. A value of 100 is opaque and a value of 0 is transparent. alpha causes MapServer to honor the alpha transparency of pixmap symbols for RGB output formats.

TYPE

TYPE [point | line | polygon | circle | annotation | raster | query]

Default: n/a

Specifies the layer type, which determines how the layer should be drawn.

LEGEND Object

Elements of the LEGEND object determine the appearance and location of the legend. It's introduced by the keyword LEGEND and terminated by the keyword END. Named classes and associated symbols are incorporated into the legend image—unnamed classes aren't. Unlike other images created by MapServer, the size of the legend image is *unknown* until the image is created; therefore, hard-coding dimensions in the tag isn't advised.

IMAGECOLOR

IMAGECOLOR [int r][int g][int b]

Default: n/a

Specifies the background color of the legend image. The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

INTERLACE

INTERLACE [on | off]

Default: on

Turns the legend image interlace on or off. This usage is deprecated—use `FORMATOPTION "INTERLACE=on"` in an `OUTPUTFORMAT` declaration instead.

KEYSIZE

KEYSIZE [int x][int y]

Default: 20 10

Specifies the size (in pixels) of symbol key boxes.

KEYSPACING

KEYSPACING [int x][int y]

Default: 5 5

Specifies spacing between labels and symbols in a legend. [x] represents the horizontal distance (in pixels) between a legend label and its symbol key box. [y] represents the vertical distance (in pixels) between adjacent symbol key boxes.

LABEL

LABEL

Default: n/a

Indicates the start of a LABEL object.

OUTLINECOLOR

OUTLINECOLOR [int r][int g][int b]

Default: n/a

Specifies the outline color for symbol key boxes. The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

POSITION

POSITION [ul | uc | ur | ll | lc | lr]

Default: lr

Specifies the position (in the map image) of the embedded legend. Any one of the six positions along the top or bottom of the map image is valid.

POSTLABELCACHE

POSTLABELCACHE [true | false]

Default: false

Specifies whether the legend is drawn after all the labels in the cache have been drawn. A value of `true` specifies that the legend will be drawn after, and a value of `false` specifies that it will be drawn before.

STATUS

STATUS [on | off | embed]

Default: on

Specifies whether a separate legend image will be created (`on`), not created (`off`), or embedded in the map image (`embed`).

TRANSPARENT

TRANSPARENT [on | off]

Default: off

Makes the background color of the legend transparent. This usage is deprecated—set transparency in an `OUTPUTFORMAT` declaration instead.

OUTPUTFORMAT Object

The `OUTPUTFORMAT` object is used to define and name output formats. An output format declaration begins with the keyword `OUTPUTFORMAT` and is terminated by the keyword `END`. It extends the concept of an image format (such as GIF or JPEG) to include details about image structure and color representation, and even which graphics driver should be used to generate an image. Keywords and values that can be used to create user-defined formats are described first, and then MapServer's implicit output formats are described.

DRIVER

DRIVER ["name"]

Default: n/a

Specifies the quote-delimited name of the graphics driver used to generate the image. The following GD drivers are supported: "GD/Gif", "GD/PNG", "GD/WBMP", and "GD/JPEG". The Flash driver name is "SWF". For GDAL-supported output, the name consists of the string "GDAL" concatenated with the GDAL shortname for the format—for example, "GDAL/GTiff".

EXTENSION

EXTENSION [extension]

Default: n/a

Specifies the file extension used for the image generated by this output format.

FORMATOPTION

FORMATOPTION [option]

Default: n/a

Allows the specification of driver- or format-specific options. May occur zero or more times in an OUTPUTFORMAT declaration. The following options are supported:

GD/JPEG. "QUALITY=n" sets JPEG image quality between 0 and 100.

GD/PNG. "INTERLACE=[on/off]" turns interlacing on or off.

GD/GIF. "INTERLACE=[on/off]" turns interlacing on or off.

GDAL/GTiff. "TILED=yes", "BLOCKXSIZE=n", "BLOCKYSIZE=n", "INTERLEAVE=[pixel/band]", "COMPRESS=[none,packbits,jpeg,lzw,deflate]".

*GDAL/**. All format options are passed on to the GDAL create function.

Note For example, you could use GDAL/HFA (HFA is a GDAL format type) to specify ERDAS Imagine as the output format in order to provide WCS services that require raw output of the data. Generally, format options are used to control how GDAL actually creates the raster.

IMAGEMODE

IMAGEMODE [PC256/RGB/RGBA/INT16/FLOAT32]

Default: n/a

Specifies the image mode used for output. The following image modes are available:

PC256. Generates a pseudo-colored image with a 256 (maximum) color palette

RGB. Creates a 24-bit RGB image with no transparency

RGBA. Creates a 32-bit RGB/alpha image with alpha-based transparency

INT16. Renders one band of data in 16-bit integer depth (restricted to raster layers that use GDAL and WMS layers)

FLOAT32. Renders one band of data in 32-bit floating-point depth (restricted to raster layers that use GDAL and WMS layers)

MIMETYPE

MIMETYPE [mimetype]

Default: n/a

Specifies the mime type used for the result.

NAME

NAME [name]

Default: n/a

Specifies the name used by the mapfile keyword `IMAGETYPE` to reference the format.

TRANSPARENT

TRANSPARENT [on | off]

Default: n/a

Specifies whether transparency is turned on or off for a particular format. It's not available for `IMAGEMODE RGB`. If `TRANSPARENCY on` and `IMAGEMODE PC256` are specified, `IMAGECOLOR` becomes the transparent color for any component of the map image.

Implicit Declarations

There are several implicit `OUTPUTFORMAT` declarations that MapServer will make if no explicit declarations are found in the mapfile. Only those formats included by default or specified during the build configuration are available.

IMAGETYPE gif

Native support via GD

```
OUTPUTFORMAT
  NAME gif
  DRIVER "GD/GIF"
  MIMETYPE "image/gif"
  IMAGEMODE PC256
  EXTENSION "gif"
END
```

IMAGETYPE GTiff

Requires GDAL support to be declared during build configuration

```
OUTPUTFORMAT
  NAME GTiff
  DRIVER "GDAL/GTiff"
  MIMETYPE "image/tiff"
  IMAGEMODE RGB
  EXTENSION "tif"
END
```

IMAGETYPE jpeg

Native support via GD

```
OUTPUTFORMAT
  NAME jpeg
  DRIVER "GD/JPEG"
  MIMETYPE "image/jpeg"
  IMAGEMODE RGB
  EXTENSION "jpg"
END
```

IMAGETYPE png

Native support via GD

```
OUTPUTFORMAT
  NAME png
  DRIVER "GD/PNG"
  MIMETYPE "image/png"
  IMAGEMODE PC256
  EXTENSION "png"
END
```

IMAGETYPE swf

Requires MING library support to be declared during build configuration

```
OUTPUTFORMAT
    NAME swf
    DRIVER "SWF"
    MIMETYPE "application/x-shockwave-flash"
    EXTENSION "swf"
    IMAGEMODE PC256
    FORMATOPTION "OUTPUT_MOVIE=SINGLE"
END
```

IMAGETYPE wbmp

Native support via GD

```
OUTPUTFORMAT
    NAME wbmp
    DRIVER "GD/WBMP"
    MIMETYPE "image/wbmp"
    IMAGEMODE PC256
    EXTENSION "wbmp"
END
```

PROJECTION Object

The PROJECTION object specifies the map projection used for displaying or describing spatial data. It begins with the keyword PROJECTION and is terminated by the keyword END.

The spatial information contained in any data set is either unprojected (in which case, its coordinates represent decimal degrees of latitude and longitude) *or* it has been projected onto a flat surface (in which case, its coordinates represent some sort of distance measure like meters or miles).

There are numerous methods for projecting a more or less spherical earth onto different kinds of flat surfaces, and a specific set of parameters is required to describe each of them. A more extensive description of projections is provided in the Appendix, and the treatment here is restricted to the general syntax for specifying projections to MapServer.

If a spatial data set is in a format that MapServer understands, then it can be rendered without specific use of projections, no matter what projection the data possesses. This leads to maps that exhibit different kinds of distortion, but in most cases, features are generally recognizable. However, if the spatial data set includes data with different projections, or if the distortion is undesirable, MapServer can project data on the fly to some common projection.

To do this, MapServer must know the projection to be used for the map images it creates, and it must also know the projection of the underlying data. The PROJECTION object is used to provide this information. The first PROJECTION object is defined at the level of the mapfile, and specifies the output projection of the map. Subsequently, each layer with a projection that differs from the output projection must then use a PROJECTION object to specify the projection of the data it accesses.

Caution On-the-fly projection is available to MapServer only if the Proj.4 library was specified during the build configuration.

The PROJECTION object begins with the keyword PROJECTION and is terminated by the keyword END. Within the PROJECTION object, quoted strings containing Proj.4 keywords are used to describe the projection.

As an example, suppose that your data set consists of several shapefiles, each containing unprojected data (i.e., the locations of the features are expressed in geographical coordinates or decimal degrees of latitude and longitude). If this spatial data is used to generate a map directly, the map features will seem to be compressed along the north-south axis. Suppose that this distortion is unacceptable and you decide that the data should be re-projected using a Lambert Conformal Conic projection.

A PROJECTION object defined at the map level specifies the output projection. Assuming that the map should be centered on 90 degrees West longitude, the output projection (using default values for several parameters) will look like this:

```
PROJECTION
    "proj=lcc"
    "lon_0=90w"
END
```

Within each layer, you need to define a PROJECTION object that describes the projection of the source data. Since the shapefiles contain unprojected spatial data, each layer-level PROJECTION object will look like the following:

```
PROJECTION
    "proj=latlong"
END
```

The Proj.4 syntax required to specify projections is straightforward, but the repertoire of projections is extensive (many are of interest only to specialists), and the arcana of map projections is beyond the scope of this book. The brief overview provided in the Appendix will describe what projections are and why they're used, and it will present the detailed syntax of several commonly used projections.

QUERYMAP Object

The elements of the QUERYMAP object determine how results of a query will be rendered. It's introduced by the keyword QUERYMAP and terminated by the keyword END.

COLOR

```
COLOR [int r][int g][int b]
```

Default: 255 255 0 (yellow)

Specifies the color used to highlight features. The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

SIZE

SIZE [int x][int y]

Default: size specified in the map object by the keyword SIZE
Specifies the size (in pixels) of the querymap image.

STATUS

STATUS [on | off]

Default: n/a
Specifies whether a querymap image will be created (on) or not created (off).

STYLE

STYLE [normal | hilite | selected]

Default: n/a
Specifies how selected features are rendered, according to the following values:

normal. Features are drawn normally, without highlighting.

hilite. Selected features are drawn in COLOR, while other features are drawn normally.

selected. Selected features are drawn normally, while other features aren't drawn.

Reference Map Object

The reference map object determines the characteristics of the reference map. It starts with the keyword REFERENCE and is terminated by the keyword END. A reference map shows the context of the currently displayed map image by outlining it on a image showing the initial map extent. Reference maps can also be used to highlight the results of a query in context just as they do with the map image. Finally, it's possible to make reference maps interactive objects and provide them with the same interactive controls as conventional map images.

COLOR

COLOR [int r][int g][int b]

Default: 255 0 0
Specifies the color used to draw the reference box. This is the fill color that obscures the underlying map area. Setting any component to -1 produces no fill. The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

EXTENT

EXTENT [int minx][int miny][int maxx][int maxy]

Default: n/a
Specifies the spatial extent of the reference image.

IMAGE

IMAGE [filename]

Default: n/a

Specifies the path (absolute or relative to the mapfile) to a file containing the GIF reference image.

MARKER

MARKER [int N | string name]

Default: crosshair

Specifies the symbol used when the reference box becomes too small. The name is the value associated with the keyword NAME in the SYMBOL definition. The number is the sequence number of the symbol, starting at 1.

MARKERSIZE

MARKERSIZE [int x]

Default: n/a

Specifies the size (in pixels) of the symbol (specified by the keyword MARKER) used when the box is too small.

MAXBOXSIZE

MAXBOXSIZE [int x]

Default: n/a

Specifies the maximum reference box size (in pixels). If the size of a box would otherwise exceed this maximum, nothing is drawn.

MINBOXSIZE

MINBOXSIZE [int x]

Default: n/a

Specifies the smallest reference box size (in pixels). For smaller sizes, the symbol specified by the keyword MARKER is used instead of a box.

OUTLINECOLOR

OUTLINECOLOR [int r][int g][int b]

Default: n/a

Specifies the color used to outline the reference box. Setting any component to -1 suppresses the outline. The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

SIZE

SIZE [int x][int y]

Default: n/a

Specifies the width and height of the reference image in pixels.

STATUS

STATUS [on | off]

Default: on

Specifies whether a reference image is created (on) or not (off).

SCALEBAR Object

The SCALEBAR object defines the scale bar. It's introduced by the keyword SCALEBAR and terminated by the keyword END. The use of TrueType fonts isn't supported, and the size of the scale bar image isn't known before it's rendered.

The SCALEBAR object has several components, and the keywords used to specify the colors of these components are the source of some confusion. First, consider the scale bar image itself as a background layer on which all the other components are drawn. The color of this layer is specified by the keyword IMAGECOLOR. The color of the scale bar itself alternates (depending on the number of intervals specified) between the color specified by the keyword BACKGROUNDCOLOR and the color specified by the keyword COLOR. In addition to this, an outline can be drawn around each interval in the color specified by the keyword OUTLINECOLOR.

BACKGROUNDCOLOR

BACKGROUNDCOLOR [int r][int g][int b]

Default: n/a

Specifies the background color of the scale bar itself, one of two alternating colors that make up the bar (the other color is specified by the keyword COLOR). The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

COLOR

COLOR [int r][int g][int b]

Default: 0 0 0

Specifies the foreground color of the scale bar itself, one of two alternating colors that make up the bar (the other color is specified by the keyword BACKGROUNDCOLOR). The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

IMAGECOLOR

IMAGECOLOR [int r][int g][int b]

Default: n/a

Specifies the background color of the image on which the scale bar is drawn. The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

INTERLACE

INTERLACE [on | off]

Default: on

Turns scale bar image interlace on or off. This usage is deprecated—use `FORMATOPTION "INTERLACE=on"` in an `OUTPUTFORMAT` declaration instead.

INTERVALS

INTERVALS [int x]

Default: 4

Specifies the number of intervals shown on the scale bar.

LABEL

LABEL

Default: n/a

Indicates the start of a LABEL object.

OUTLINECOLOR

OUTLINECOLOR [int r][int g][int b]

Default: n/a

Specifies the color used to outline intervals. Setting any component to -1 suppresses outlining. The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

POSITION

POSITION [ul | uc | ur | ll | lc | lr]

Default: lr

Specifies the position (in the map image) of the embedded scale bar. Any one of the six positions along the top or bottom of the map image are valid.

POSTLABELCACHE

POSTLABELCACHE [true | false]

Default: false

Specifies whether the scale bar is drawn after all the labels in the cache have been drawn (true), or before (false). POSTLABELCACHE is valid for embedded scale bars only.

SIZE

SIZE [int x][int y]

Default: n/a

Specifies the size (in pixels) of the scale bar (not including labels).

STATUS

STATUS [on | off | embed]

Default: on

Specifies whether a separate scale bar image will be created (on), not created (off), or embedded in the map image (embed).

STYLE

STYLE [int x]

Default: n/a

Specifies the scale bar style. Supported options are 0 and 1.

TRANSPARENT

TRANSPARENT [on | off]

Default: off

Makes the background color of the scale bar image transparent. This usage is deprecated—set the transparency in the OUTPUTFORMAT object instead.

UNITS

UNITS [feet | inches | kilometers | meters | miles]

Default: miles

Specifies the scale bar units. If scale bar units and map units differ, conversion is done automatically.

STYLE Object

Elements of the STYLE object determine how symbols will be rendered. A class may contain multiple STYLE objects, which are applied in sequence. The STYLE object begins with the keyword STYLE and is terminated by the keyword END.

ANTIALIAS

ANTIALIAS [true | false]

Default: n/a

Indicates that TrueType fonts and CARTOLINE symbols are to be antialiased.

BACKGROUNDCOLOR

BACKGROUNDCOLOR [int r][int g][int b]

Default: n/a

Specifies the color used to render non-transparent symbols. The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

COLOR

COLOR [int r][int g][int b]

Default: n/a

Specifies the color used for drawing features. The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

MAXSIZE

MAXSIZE [int N]

Default: 50

Specifies the maximum size (in pixels) at which a symbol will be drawn.

MINSIZE

MINSIZE [int N]

Default: 0

Specifies the minimum size (in pixels) at which a symbol will be drawn.

OFFSET

OFFSET [int x][int y]

Default: n/a

Specifies the offset (in pixels) for shadows.

OUTLINECOLOR

OUTLINECOLOR [int r][int g][int b]

Default: n/a

Specifies the outline color (for polygons only). The values are 1-byte integers in the range of 0 to 255, representing relative amounts of red, green, and blue.

SIZE

SIZE [int N]

Default: vector and ellipse types: the range of y values; pixmaps: the vertical size of the image; TrueType font symbols: 1
Specifies the height of a symbol.

SYMBOL

SYMBOL [int | string | filename]

Default: 0

Specifies the symbol used for drawing features, identified by name or number. The name is the value associated with the keyword NAME in the SYMBOL definition. The number is the sequence number of the symbol, starting at 1. filename specifies the path (absolute or relative to the mapfile) to a file containing a GIF or PNG image.

WEB Object

The WEB object specifies the web interface, including paths, URLs, template files, and other details that affect the way the application responds to the user.

EMPTY

EMPTY [url]

Default: value specified by keyword ERROR

Specifies the URL (not the path) of the web page to be displayed if a query returns no results.

ERROR

ERROR [url]

Default: n/a

Specifies the URL (not the path) of the web page to be displayed if a MapServer error appears.

FOOTER

FOOTER [filename]

Default: n/a

Specifies the name of the HTML template file displayed after all query results have been presented. It's used for multi-result query modes.

HEADER

HEADER [filename]

Default: n/a

Specifies the name of the HTML template file displayed before any query results have been presented. It's used for multi-result query modes.

IMAGEPATH

IMAGEPATH [path]

Default: n/a

Specifies the path to the directory where images and temporary files are written. The path must end with a / or \ (depending on the platform).

IMAGEURL

IMAGEURL [path]

Default: n/a

Specifies the base URL that points to the directory where images are written. The browser uses this path to retrieve images.

LOG

LOG [filename]

Default: n/a

Specifies the file where MapServer activity will be logged. It must be writable by the web server.

MAXSCALE

MAXSCALE [double]

Default: n/a

Specifies the maximum scale at which a map will be returned. Requests for larger scales will return this scale.

MAXTEMPLATE

MAXTEMPLATE [file | url]

Default: n/a

Specifies the template to be used if the requested scale exceeds the value specified by the keyword MAXSCALE.

METADATA

METADATA

```
"wms_ext" "minx miny maxx maxy"
"wms_title" "A title"
END
```

Allows data to be stored as name-value pairs, accessible by means of template tags. In the example just displayed, the values of "wms_ext" and "wms_title" are available as [wms_ext] and [wms_title]. This is also the method used to provide WMS output for MapServer. In this case, substitute appropriate values for minx, miny, maxx, and maxy.

MINSCALE

MINSCALE [double]

Default: n/a

Specifies the minimum scale at which a map will be returned. Requests for smaller scales will return the minimum scale.

MINTEMPLATE

MINTEMPLATE [file | url]

Default: n/a

Specifies the template to be used if the requested scale is smaller than the value specified by the keyword MINSCALE.

TEMPLATE

TEMPLATE [filename | url]

Default: n/a

Specifies the main HTML template file used to display a map in interactive mode.

CGI Variables

MapServer recognizes certain CGI variables returned from HTML forms, and the values of these variables can affect the way that the application operates. The variables indicate

(among other things) the coordinates of the location of a mouse click, which layers should be displayed, and what zoom factor and direction will be applied to the map.

BUFFER

`BUFFER [distance]`

Contains a distance (in the same coordinates as the mapfile) and is used to create a new map extent around the point specified by the variable `MAPXY`. It's used as an alternative to `SCALE`.

CONTEXT

`CONTEXT [filename]`

Specifies the name (relative to the mapfile) of a context file. Context files contain information used to request WMS layers (`CONNECTIONTYPE wms`). More information is available at <http://mapserver.gis.umn.edu/doc/mapcontext-howto.html>.

ID

`ID [id-string]`

Specifies a replacement for the default session ID. MapServer generates a unique ID for each session by concatenating the number of seconds since January 1, 1970 at 0:00:00, and the process ID. This variable replaces that default value.

IMG

`IMG`

Is used to identify the input image to MapServer. The name specified in an input tag containing the map image (`<input type="image" name="img" . . . >`) is the base name of the two variables (`img.x` and `img.y`) used to return the image coordinates of a mouse click. MapServer expects that the name associated with the image is `img`.

IMGBOX

`IMGBOX [int x1][int y1][int x2][int y2]`

Represents the coordinates (in pixels) of opposite corners (upper-left to lower-right) of an image drag box as a space-delimited list. It's used by Java-based front ends.

IMGEXT

`IMGEXT [int minx][int miny][int maxx][int maxy]`

Contains the spatial extent of the current map image as a space-delimited list representing the lower-left and upper-right coordinates of the extent.

IMGSHAPE

IMGSHAPE [int x_1 int y_1][int x_2 int y_2][int x_3 int y_3] . . .

Specifies the vertex coordinates of a user-defined polygon (in image coordinates—i.e., pixels) as a space-delimited list. It's used for query purposes.

IMGSIZE

IMGSIZE [int cols][int rows]

Specifies the size of the map image (in pixels).

IMGXY

IMGXY [int x][int y]

Contains the image coordinates (in pixels) of a mouse click on the map image.

LAYER

LAYER [name]

Specifies the name of a layer, thereby setting the layer's STATUS to on.

LAYERS

LAYERS [name name . . .]

Specifies a space-delimited list of layer names, setting the STATUS of all layers to on.

MAP

MAP [filename]

Contains the full path to the mapfile.

MAPEXT

MAPEXT [int minx][int miny][int maxx][int maxy], MAPEXT (shape)

Contains the spatial extent of a map to be created as a space-delimited list. It's used as an alternative to creating an extent with MAPXY and BUFFER or SCALE. When the value shape is used in a query mode, MapServer creates an extent that's slightly larger than the matching shape.

MAPSIZE

MAPSIZE [int cols][int rows]

Contains the image size (in pixels) of the map to be created. It's used to change the map resolution interactively.

MAPSHAPE

MAPSHAPE [int x_1 int y_1][int x_2 int y_2][int x_3 int y_3] . . .

Contains a user-defined polygon in map coordinates (i.e., real-world coordinates) as a space-delimited list. It's used for query purposes.

MAPXY

MAPXY [[int x][int y] | [shape]]

Contains the coordinates of a point in the same units as the underlying shapefile. MapServer creates a new map extent around the point. The extent is determined either by the value specified by the variable `BUFFER` (specified in the same units as `MAPXY`), or by setting the map scale to the value specified by the variable `SCALE`. In query mode, specifying `shape` sets the new extent to the extent of the selected shape.

MAXX

MAXX [number]

Contains the maximum x coordinate of the spatial extent for a new map or query. `MAXX` is one of the components of `MAPEXT`.

MAXY

MAXY [number]

Contains the maximum y coordinate of the spatial extent for a new map or query. `MAXY` is one of the components of `MAPEXT`.

MINX

MINX [number]

Contains the minimum x coordinate of the spatial extent for a new map or query. `MINX` is one of the components of `MAPEXT`.

MINY

MINY [number]

Contains the minimum y coordinate of the spatial extent for a new map or query. `MINY` is one of the components of `MAPEXT`.

MODE

MODE [value]

Contains the mode of operation. The following mode values are supported:

BROWSE. Default mode providing interactive navigation of the map.

QUERY. Spatial search of all queriable layers that returns the closest feature (within a tolerance specified by the mapfile layer-level keyword *TOLERANCE*). *QUERYMAP* mode performs the same query, but produces only a map and doesn't return an attribute list.

NQUERY. Spatial search of all queriable layers that returns all features (within a tolerance specified by the mapfile layer-level keyword *TOLERANCE*). *NQUERYMAP* mode performs the same query, but produces only a map and doesn't return an attribute list.

ITEMQUERY. Attribute search that returns the first feature that has an attribute that matches the value of the form variable *QSTRING*. The value of *QSTRING* is an expression with syntax identical to the mapfile class-level keyword *EXPRESSION*. The form variable *QLAYER* restricts the search to a single layer, and *QITEM* restricts the search to a single attribute. If *QLAYER* is omitted, all layers are searched. If *QITEM* is omitted, all attributes are searched. *ITEMQUERYMAP* mode performs the same query, but produces only a map and doesn't return an attribute list.

ITEMNQUERY. Attribute search that returns all features that have an attribute that matches the value of the form variable *QSTRING*. The value of *QSTRING* is an expression with syntax identical to the mapfile class-level keyword *EXPRESSION*. The form variable *QLAYER* restricts the search to a single layer, and *QITEM* restricts the search to a single attribute. If *QLAYER* is omitted, all layers are searched. If *QITEM* is omitted, all attributes are searched. *ITEMNQUERYMAP* mode performs the same query, but produces only a map and doesn't return an attribute list.

FEATUREQUERY. Spatial search that selects a single polygon feature in a layer specified by the form variable *SLAYER*, and returns all features in all other queriable layers that fall inside that polygon or within the distance specified by each layer's *TOLERANCE*. The *TOLERANCE* of each queriable layer can be different. The *TOLERANCE* of *SLAYER* is irrelevant since only a single feature from *SLAYER* is returned. *FEATUREQUERYMAP* mode performs the same query, but produces only a map and doesn't return an attribute list.

FEATURENQUERY. Spatial search that selects all polygon features in a layer specified by the form variable *SLAYER* that fall within the *TOLERANCE* distance specified for *SLAYER*. It returns all features in all other queriable layers that fall inside the selected polygons or within the distance specified by each layer's *TOLERANCE*. The *TOLERANCE* of each queriable layer can be different. If the *TOLERANCE* of *SLAYER* is 0, this mode is functionally equivalent to *FEATUREQUERY* mode. *FEATURENQUERYMAP* mode performs the same query, but produces only a map and doesn't return an attribute list.

ITEMFEATUREQUERY. Attribute search that selects the first polygon feature in a layer specified by the form variable *SLAYER* that has an attribute that matches the value of the form variable *QSTRING*. The value of *QSTRING* is an expression with syntax identical to the mapfile class-level keyword *EXPRESSION*. It returns all features in all other queriable layers that fall inside the selected polygon or within the distance specified by each layer's *TOLERANCE*. The *TOLERANCE* of each queriable layer can be different. The form variable *QLAYER* restricts the search to a single layer. If *QLAYER* isn't specified, all layers are searched. *ITEMFEATUREQUERYMAP* mode performs the same query, but produces only a map and doesn't return an attribute list.

ITEMFEATUREQUERY. Attribute search that selects all polygon features in a layer specified by the form variable *SLAYER* that have an attribute that matches the value of the form variable *QSTRING*. The value of *QSTRING* is an expression with syntax identical to the mapfile class-level keyword *EXPRESSION*. It returns all features in all other queriable layers that fall inside the selected polygons or within the distance specified by each layer's *TOLERANCE*. The *TOLERANCE* of each queriable layer can be different. The form variable *QLAYER* restricts the search to a single layer. If *QLAYER* isn't specified, all layers are searched. *ITEMFEATUREQUERYMAP* mode performs the same query, but produces only a map and doesn't return an attribute list.

INDEXQUERY. Search that selects the feature in a layer specified by the form variable *QLAYER* that has a shape index (i.e., sequence number in the shapefile) equal to the value specified by the form variable *SHAPEINDEX*. Optionally, the form variable *TILEINDEX* can be used to specify the tile that contains the shapefile to be queried. In such a case, if *TILEINDEX* is 2 and *SHAPEINDEX* is 17, MapServer will return the feature with a shape index of 17 in the shapefile pointed to by tile number 2 in the tile index. *INDEXQUERYMAP* mode performs the same query, but produces only a map and doesn't return an attribute list.

MAP. Image-only mode that produces a map image. Interactive navigation isn't supported.

REFERENCE. Image-only mode that produces a reference map image.

SCALEBAR. Image-only mode that produces a scale bar image.

LEGEND. Image-only mode that produces a legend image.

ZOOMIN. Browse mode that sets *ZOOMDIR* to 1.

ZOOMOUT. Browse mode that sets *ZOOMDIR* to 1.

QITEM

QITEM [name]

Contains the name of the attribute to be searched in one of the *ITEMQUERY* modes. MapServer compares the value of the form variable *QSTRING* with the value of the specified attribute.

QLAYER

QLAYER [name]

Restricts a search to a single layer, where [name] is a layer name specified by the layer-level keyword *NAME*. If it's not supplied, then all layers are searched in sequence.

QSTRING

QSTRING [expression]

Contains the query string. The value of *QSTRING* is an expression with syntax identical to the mapfile class-level keyword *EXPRESSION*.

QUERYFILE

QUERYFILE [filename]

Specifies a queryfile that's loaded before any other processing, when in BROWSE or NQUERY mode.

REF

REF

Identifies the input tag containing the reference map image (<input type="image" name="ref" . . . >). The coordinates of a mouse click on the reference map image are returned in the two variables (ref.x and ref.y).

REFXY

REFXY [x][y]

Contains the image coordinates (in pixels) of a mouse click on the reference map image. REFXY is used by Java-based front ends.

SAVEQUERY

SAVEQUERY

Indicates that MapServer should save query results to a temporary file that can be used later.

SCALE

SCALE [number]

Contains a map scale and is used to create a new map extent around the point specified by the variable MAPXY. SCALE is used as an alternative to BUFFER. The scale is specified by the denominator only, so if the scale is 1:1,000,000, SCALE has a value of 1000000.

SEARCHMAP

SEARCHMAP

Causes MapServer to search the new extent when searching a querymap in NQUERY mode. Usually, a query will search the map as previously displayed in the browser. However, if a querymap is navigated with pan and zoom, specifying SEARCHMAP causes the search to be executed on the new extent—not the extent that existed before panning and zooming.

SHAPEINDEX

SHAPEINDEX [index]

Contains the shape index (i.e., sequence number) of a shape in the layer specified by the variable QLayer. SHAPEINDEX is used to perform index queries in INDEXQUERY mode. Use of the variable TILEINDEX is only required with tiled layers.

SLAYER

SLAYER [name]

Contains the name of the select layer for feature query modes. The select layer type must be polygon.

TILEINDEX

TILEINDEX [index]

Contains the tile index (i.e., the sequence number of the tile) of a tiled shapefile used to perform index queries with INDEXQUERY mode. TILEINDEX is used with the variable SHAPEINDEX.

ZOOM

ZOOM [number]

Contains the zoom factor to apply to the new map extent. A positive factor zooms in, a negative factor zooms out, and a factor of 0 pans. ZOOM is used as an alternative to the combination of ZOOMDIR and ZOOMSIZE. Minimum zoom is -25 and maximum zoom is 25.

ZOOMDIR

ZOOMDIR [1 | 0 | -1]

Contains the zoom direction. Setting ZOOMDIR to 1 zooms in, setting ZOOMDIR to -1 zooms out, and setting ZOOMDIR to 0 pans.

ZOOMSIZE

ZOOMSIZE [number]

Contains the zoom factor, which is always a positive number. ZOOMSIZE and ZOOMDIR must be used together.

Substitution Strings

MapServer doesn't generate the HTML tags it uses to display its responses—instead, it searches HTML templates for embedded substitution strings and replaces them with the current values of the corresponding MapServer variables. Substitution strings are used to present information to the application user, to maintain the state of the application across invocations, and to provide a link between the application and information stored in spatial data sets, external dBse files, or the mapfile itself.

Syntactically, a substitution string is just a sequence of characters delimited by square brackets, that MapServer recognizes, and to which it can assign a specific value or range of possible values. A value might be a numeric quantity representing the current scale of the map, for example, or it might be a space-delimited list of numeric quantities representing the map extent. Some values are HTML keywords that specify the checked state of a check box or the URL of an image.

In all these cases, the substitution string itself is a fixed, invariant string similar to a CGI form variable. In fact, substitution strings sometimes bear a strong resemblance to CGI form variables. For example, the string `[scale]`, embedded in a template, will be replaced with the current scale of the map, whereas the form variable `scale` is the value returned to MapServer that specifies what the scale of the new map extent should be.

There is, however, a more complicated usage that employs variable strings to identify external resources and track application-specific information generated by user input. In these cases, a part—or even all—of a substitution string will depend on the factors specific to the local environment, such as attribute names, layer names, or database names. Each of these variable substitution strings is preceded by an asterisk in the section that follows.

***[item]**

Use: queries

The value of an item in the attribute table of a queried layer, encoded for HTML.

***[item_esc]**

Use: queries

The escaped form of the value of an item in the attribute table of a queried layer.

***[item_raw]**

Use: queries

The raw form of the value of an item in the attribute table of a queried layer, not escaped or encoded for HTML.

***[join_joinname]**

Use: queries

The results of a one-to-many join, consisting of the concatenation of join-level templates (one for each result).

***[joinname_item]**

Use: queries

The HTML-encoded value of an item in the DBF file specified in the join.

***[joinname_item_esc]**

Use: queries

The escaped form of the value of an item in the DBF file specified in the join.

***[joinname_item_raw]**

Use: queries

The raw form of the value of an item in the DBF file specified in the join.

***[layername_check]**

Use: layer reference

The current checked status of the check box associated with the layer `layername`.

***[layername_metadatakey]**

Use: layer reference

The metadata associated with the key `metadatakey` specified in layer `layername`. The underscore character is mandatory.

***[layername_select]**

Use: layer reference

The current selected status of the select tag associated with the layer `layername`.

***[metadatakey]**

Use: general

The metadata associated with the key `metadatakey` specified in the WEB object.

***[metadatakey_esc]**

Use: general

The escaped version of the metadata associated with the key `metadatakey` specified in the WEB object.

***[variablename]**

Use: general

The value of the form variable `variablename`, passed to MapServer on the previous invocation.

***[variablename_esc]**

Use: general

The escaped version of the value of the form variable `variablename`, passed to MapServer on the previous invocation.

***[zoom_NN_check]**

Use: zoom state

The current checked status of the check box associated with zoom level `NN`. For example, if the current zoom level is 12, then the string `[zoom_12_check]` is replaced with the string `CHECKED`—otherwise, it's replaced with `""`. If the current zoom level is -6, then `[zoom_-6_check]` is replaced with `CHECKED`. Zoom levels range from -25 to 25.

***[zoom_NN_select]**

Use: zoom state

The current selected status of the `select` tag associated with zoom level `NN`. For example, if the current zoom level is 19, then `[zoom_19_select]` will be replaced with `SELECTED`; if it's not, it will be replaced by `""`.

[cellsize]

Use: image geometry

The size of a pixel in world units.

[center]

Use: image geometry

The space-delimited `x y` coordinates (in pixels) of the center of the map image.

[center_x]

Use: image geometry

The `x` coordinate (in pixels) of the center of the map image.

[center_y]

Use: image geometry

The `y` coordinate (in pixels) of the center of the map image.

[cl]

Use: queries

The name of the current layer. `[cl]` is available only when processing query results.

[dx]

Use: map geometry

The width of the extent (`maxx - minx`) in map units.

[dy]

Use: map geometry

The height of the extent (`maxy - miny`) in map units.

[host]

Use: general

The web server host name.

[id]

Use: general

The unique session ID.

[img]

Use: file reference

The path to the map image (the concatenation of IMAGE_URL and the file name) with respect to the Apache DocumentRoot.

[layers]

Use: layer reference

The list of space-delimited active map layers. Used for a "POST" request.

[layers_esc]

Use: layer reference

The escaped version of the list of space-delimited active map layers. Used for a "POST" request.

[legend]

Use: file reference

The path to the new legend image, with respect to the Apache DocumentRoot.

[lrn]

Use: queries

The sequence number of the result in the current layer (beginning at 1).

[map]

Use: file reference

The path to the mapfile.

[mapext]

Use: map geometry

The space-delimited extent of the map in map units.

[mapext_esc]

Use: map geometry

The escaped version of the space-delimited extent of the map in map units.

[mapext_latlon]

Use: projections

The space-delimited extent of the map, reported in degrees of latitude and longitude.

[mapext_latlon_esc]

Use: projections

The escaped version of the space-delimited extent of the map, reported in degrees of latitude and longitude.

[mapheight]

Use: image geometry
The image height (in pixels).

[maplat]

Use: projections
The latitude of a mouse click when a projection is used.

[maplon]

Use: projections
The longitude of a mouse click when a projection is used.

[mapsize]

Use: image geometry
The space-delimited image width and height (in pixels).

[mapsize_esc]

Use: image geometry
The escaped version of the space-delimited image width and height (in pixels).

[mapwidth]

Use: image geometry
The image width (in pixels).

[mapx]

Use: map geometry
The x coordinate of a mouse click in map units.

[mapy]

Use: map geometry
The y coordinate of a mouse click in map units.

[maxlat]

Use: projections
The maximum latitude of the map extent (a component of [mapext_latlon]).

[maxlon]

Use: projections
The maximum longitude of the map extent (a component of [mapext_latlon]).

[maxx]

Use: map geometry

The maximum x coordinate of the extent of the map in map units.

[maxy]

Use: map geometry

The maximum y coordinate of the extent of the map in map units.

[minlat]

Use: projections

The minimum latitude of the map extent (a component of [mapext_latlon]).

[minlon]

Use: projections

The minimum longitude of the map extent (a component of [mapext_latlon]).

[minx]

Use: map geometry

The minimum x coordinate of the extent of the map in map units.

[miny]

Use: map geometry

The minimum y coordinate of the extent of the map in map units.

[nl]

Use: queries

The number of layers that returned results.

[nlr]

Use: queries

The number of results returned from the current layer.

[nr]

Use: queries

The total number of results returned across all layers.

[port]

Use: general

The web server port number.

[queryfile]

Use: file reference

The path to the queryfile if SAVEQUERY was specified.

[rawext]

Use: map geometry

The space-delimited raw extent of the map in map units.

[rawext_esc]

Use: map geometry

The escaped version of the space-delimited raw extent of the map in map units.

[rawmaxx]

Use: map geometry

The maximum x coordinate of the raw extent of the map in map units.

[rawmaxy]

Use: map geometry

The maximum y coordinate of the raw extent of the map in map units.

[rawminx]

Use: map geometry

The minimum x coordinate of the raw extent of the map in map units.

[rawminy]

Use: map geometry

The minimum y coordinate of the raw extent of the map in map units.

[ref]

Use: file reference

The path to the reference image with respect to the Apache DocumentRoot.

[rn]

Use: queries

The sequence number of a result over all layers (beginning at 1).

[scale]

Use: image geometry

The scale of the map image.

[scalebar]

Use: file reference

The path to the scale bar image with respect to the Apache DocumentRoot.

[shpclass]

Use: queries

The class index (sequence number within the layer) of the current shape.

[shpext]

Use: queries

The space-delimited extent of the current shape, with a 5-percent buffer.

[shpext_esc]

Use: queries

The escaped version of the space-delimited extent of the current shape, with a 5-percent buffer.

[shpidx]

Use: queries

The shape index of the current shape (starting from 0).

[shpmaxx]

Use: queries

The maximum x coordinate of the extent of the current shape.

[shpmaxy]

Use: queries

The maximum y coordinate of the extent of the current shape.

[shpmid]

Use: queries

The space-delimited coordinates of the center of the current shape's extent.

[shpmidx]

Use: queries

The x coordinate of the center of the current shape's extent.

[shpmidy]

Use: queries

The y coordinate of the center of the current shape's extent.

[shpminx]

Use: queries

The minimum x coordinate of the current shape's extent.

[shpminy]

Use: queries

The minimum y coordinate of the current shape's extent.

[shpxy options]

Use: queries

A list of coordinates of the vertices comprising a shape. By default, this is comma-delimited, but other formatting options can also be specified. It's formatted according to the options specified. Its attributes are h (header), f (footer), and s (separator). The options are as follows:

cs specifies coordinate separator; the default is a comma: cs=",".

xh specifies characters to display before the x coordinate; the default is null: xh="".

xf specifies characters to display after the x coordinate; the default is space: xf=" ".

yh specifies characters to display before the y coordinate; the default is null: yh="".

yf specifies characters to display after the y coordinate; the default is null: yf="".

ph specifies the characters to display before each part of a multipart feature; the default is null: ph="".

pf specifies the characters to display after each part of a multipart feature; the default is null: pf="".

ps specifies the characters to display between parts of a multipart feature; the default is null: ps="".

sh specifies the characters to display before a feature; the default is null: sh="".

sf specifies the characters to display after a feature; the default is null: sf="".

precision specifies the number of decimals of precision for displayed coordinates; the default is 0: precision=0.

proj specifies the output projection of the coordinates using Proj.4 syntax; the default is none. proj=image converts world coordinates to image coordinates.

As an example, [shpxy xh=":" xf=" | " yh="(" yf=")" precision=2 proj=image] will format the coordinates of a sequence of points as follows: 320.00 | (240.00):320.00 | (235.00).

[tileindex]

Use: queries

The tile index of the current tile, or -1 if the shapefile isn't tiled.

[toggle_layers]

Use: layer reference

The list of all layers that have STATUS on or STATUS off (i.e., those layers that can be toggled).

[toggle_layers_esc]

Use: layer reference

The escaped version of the list of all layers that have STATUS on or STATUS off (i.e., those layers that can be toggled).

[version]

Use: general

The MapServer version number.

[zoomdir_0_check]

Use: zoom state

The current checked status of the check box associated with zoom direction 0 (pan). For example, if the current zoom direction is 0, then [zoomdir_0_check] will be replaced with "CHECKED"; if not, then it will be replaced with "".

[zoomdir_0_select]

Use: zoom state

The current selected status of the select tag associated with zoom direction 0 (pan). For example, if the current zoom direction is 0, then [zoomdir_0_select] will be replaced with "SELECTED" ; if not, then it will be replaced with "".

[zoomdir_1_check]

Use: zoom state

The current checked status of the check box associated with zoom direction 1 (zoom in). For example, if the current zoom direction is 1, then [zoomdir_1_check] will be replaced with "CHECKED"; if not, then it will be replaced with "".

[zoomdir_-1_check]

Use: zoom state

The current checked status of the check box associated with zoom direction -1 (zoom out). For example, if the current zoom direction is -1, then [zoomdir_-1_check] will be replaced with "CHECKED"; if not, then it will be replaced with "".

[zoomdir_1_select]

Use: zoom state

The current selected status of the select tag associated with zoom direction 1 (zoom in). For example, if the current zoom direction is 1, then [zoomdir_1_select] will be replaced with "SELECTED"; if not, then it will be replaced with "".

[zoomdir_-1_select]

Use: zoom state

The current selected status of the select tag associated with zoom direction -1 (zoom out).

For example, if the current zoom direction is -1, then [zoomdir_-1_select] will be replaced with "SELECTED"; if not, then it will be replaced with "".

Index

■ Numbers and Symbols

- # (pound sign)
 - using to insert comments, 17
- / (forward slash)
 - for delimiting regular expressions, 74–75
 - strings delimited by, 42
- ./Configure -h
 - running to see a list of available Perl configuration options, 168

■ A

- AddType directive
 - removing .php from, 209
- ANGLE keyword
 - for specifying angle at which all labels are drawn, 67–68, 323
- ANNOTATION layer
 - code for specifying in the fourth.map mapfile, 144
- annotation layers
 - using, 81–83
 - using to label features, 26
- ANTIALIAS keyword
 - for causing label text to be antialiased, 323
 - for specifying that TrueType font symbols should be antialiased, 381
- ANTIALIAS STYLE object
 - function of, 347
- Apache
 - forcing to load CGI version of PHP, 208
- Apache DocumentRoot
 - in the example environment, 13
- Apress website
 - for downloading
 - mapserver_create_restaurant code, 236
 - perlms_hello.pl code available from, 169
- area searches and point queries
 - performed by NQUERY mode, 130–131
- attribute queries
 - provided by MapServer, 103–104

- attribute searches
 - performed by ITEMQUERY mode, 131
- azimuthal projections
 - function of, 378–379
 - schematic representation of, 379

■ B

- BACKGROUNDCLASS object
 - for specifying color to be used to render non-transparent symbols, 315
- BACKGROUNDCLASS keyword
 - setting background color of a legend image with, 86–87
 - for setting the background color of scale bar, 84
- BACKGROUNDCLASS LABEL object
 - for specifying the background rectangle used to highlight a label, 323
- BACKGROUNDCLASS SCALEBAR object
 - for specifying background color of a scale bar, 344
- BACKGROUNDCLASS STYLE object
 - for specifying color used to render non-transparent symbols, 347
- BACKGROUNDSHADOWCOLOR LABEL object
 - for specifying shadow color of a background rectangle, 323
- BACKGROUNDSHADOWSIZE LABEL object
 - for specifying the offset of the background shadow in pixels, 324
- Beginning Perl, Second Edition* (Apress, 2004)
 - by James Lee and Simon Cozens, 167
- Beginning PHP 5 and MySQL: From Novice to Professional* (Apress, 2004)
 - by W. Jason Gilmore, 207, 235
- BigLine symbol
 - using to draw roads, 80
- bitmapped fonts
 - how they are rendered, 19
 - vs. TrueType fonts, 66–67

- Bostrup, Erik
 - overlib JavaScript library created by, 238
- Boutell website
 - GD page address, 13
- Browse mode
 - as MapServer default mode, 104
- browsers
 - performing spatial queries for
 - Mozilla-like, 260–263
- BUFFER CGI variable
 - used as an alternative to SCALE, 351
- BUFFER LABEL object
 - for specifying amount of space left around
 - text labels, 324
- building and installing
 - FreeType, 8–9
 - GD library, 9
 - GDAL (Geospatial Data Abstraction Library), 9–10
 - libJPEG, 7–8
 - libpng, 7
 - Proj.4, 9
 - shapelib, 10–11
 - zlib, 6–7

C

- Cartographic Projection Procedures for the UNIX Environment—A User's Manual*
 - by Gerald Evenden, 379
- cartographic projections, 373–381
- CARTOLINE type symbols
 - properties of, 384–385
- Central Cylindrical projection, 375
- CGI (Common Gateway Interface)
 - website address for detailed information
 - about, 15
- CGI MapServer web application
 - a broken image rendered by, 21–22
 - building the first map in, 23–30
 - components of, 15
 - creating, 31–54
- CGI variables, 350–357
 - and substitution strings, 137–139
- CHARACTER keyword
 - for specifying character used to render a
 - symbol, 381
- checkboxes
 - code for defining in second.html template
 - file, 46

- CHECKED state
 - setting for several layers in
 - perlms_third.pl, 174
- Circle symbol
 - using, 81
- Cities-layer templates
 - in MapServer Fourth Application, 154–155
- [cl]
 - function of in queries, 138
- CLASS keyword
 - for introducing a class, 26–27
- class labels
 - effect of failing to use easily
 - distinguishable, 62
- CLASS LAYER object
 - for indicating the start of a CLASS object, 327
- CLASS object
 - for determining appearance and labeling
 - properties of features, 315–319
 - for specifying map features and how they
 - are rendered, 34
- classes
 - using, 76–79
 - using expressions to define, 74–76
 - using to distinguish features of the same
 - type, 77
 - using to restrict features rendered at a
 - specific scale, 78–79
- CLASSITEM keyword
 - identifying the name of the attribute used
 - to classify features with, 74–76
 - using to select the attributes you wish to
 - render, 39
- CLASSITEM LAYER object
 - function of, 327
- class-level query templates
 - for displaying tabular query results for
 - Countries layer, 153–154
 - in fourth_cities_header.html file, 155
 - function of, 135
- click point
 - code for returning image coordinates
 - of, 178
- code listing
 - for accessing mapObj coordinates
 - individually, 194
 - for adding a point to a shape object, 254
 - of beginning of perlms_third.pl
 - application, 173

- for building PHP with MySQL configure option, 236
- for calculating map width and height of extent in map units, 180
- for calculating the zoom factor to pass to zoomPoint() method, 179
- to check the status and set variables \$nquery and \$browse, 261
- for checking that PHP is a loadable module, 208
- of class-level query template fourth_cities_query.html, 166
- of class-level query template fourth_countries_query.html, 165–166
- of the class-level query template in the fourth-countries_query.html file, 153–154
- for closing open tags in fourth.html file, 150
- for closing open tags in phpms_third.php file, 217
- commands for building the Perl MapScript, 169
- of complete mapfile first.map, 20, 27–28
- for complete phpms_hello.php, 222
- to configure and build the Python library, 188
- for configuring, building, testing, and installing zlib, 6
- for connecting database server to localhost, 253
- containing second.html substitution string, 45
- converting click point coordinates from image to geographic, 261–262
- for converting from image to map coordinates, 213
- for converting mouse-click point from image to map coordinates, 176, 216
- for converting pixels to map coordinates using proportions, 198
- for creating a chain of references to access extent coordinates, 215
- for creating and saving reference map and legend images to disk, 175
- for creating a new majObj map based on third.map, 193
- for creating a new mapObj based on third.map, 214
- for creating a PHP/MapScript rectObj(), 214
- for creating a Python MapScript CGI object referenced by parms, 193
- for creating a Python MapScript rectangle object, 192–193
- creating a queryCacheObj for a layer, 262–263
- for creating a reference to a new map image, \$map, 170
- for creating a table and placing it in the main application template, 387
- for creating a unique file name for the map image, 170
- for creating a unique identifier for various map images, 193, 215
- for creating a vector symbol referenced by the name box, 383
- for creating a virtual click point in perlms_third.pl, 174
- for creating a virtual click point in phpms_third.php, 214
- for creating pointObj() and rectObj() MapScript objects, 174
- for creating the module php_mapscript.so in /mapscript/php3/, 209
- for creating the virtual click point for python_third.py file, 192
- to define a URL template that saves current state of application, 151–152
- for defining a LABEL in the fourth.map mapfile, 143
- for defining a line layer in the fourth.map mapfile, 143–144
- for defining an unnamed CLASS in fourth.map mapfile, 142
- defining a PHP/MapScript PointObj(), 214
- for defining a point layer named Cities in fourth.map mapfile, 142
- defining a reference map, 87
- for defining checkboxes for second.html template file, 46
- defining CreateTTimagemap() function, 255
- for defining file names and URLs for map image, reference map image, and legend image, 175
- defining majorstreets layer in Slurp and Burp Restaurants application, 249

- defining points-of-interest layer in Slurp and Burp Restaurants application, 250
- defining QUERYMAP object in fourth.map mapfile, 141
- for defining second.map WEB object parameters, 36
- defining streets layer in Slurp and Burp Restaurants application, 250
- for defining symbol for Slurp and Burp Restaurants application, 247–248
- for defining the default extent as an array in perlms_third.pl, 175
- for defining the legend template, 387
- for defining the MapScript pointObj() object, 192
- for defining the path to the Python MapScript mapfile, 189
- for defining the PHP/MapScript default extent as an array, 214
- for defining the Python MapScript default extent as an array, 193
- for defining the submit button, 217
- describing hydrographic layer in Slurp and Burp Restaurants application, 249
- for determining if the script has been invoked by a form or not, 175, 193
- for displaying the initialization file in browser, 47
- for doing longitude and height-to-latitude conversions, 199
- for drawing a non-contiguous vector symbol, 384
- for drawing interstate highway layers, 73
- for drawing urban area polygons, 72–73
- for embedding initial values in an HTML template file, 21–23
- employing the zoomPoint() method in PHP/MapScript, 220
- employing the zoomPoint() method in Python MapScript, 198
- for employing zoomPoint() method to center map on click point, 179
- for finishing the MapScript web page, 176–177
- for first invocation for Slurp and Burp Restaurants application, 251–252
- for formatting the query string and executing the query, 253
- for fourth.map for the map name, units, size, background color, and image type, 140
- for fourth.map mapfile WEB object, 140
- for generating and opening HTML tags and writing header, 170
- for generating HTML tags for displaying map image, 190, 211
- for generating the third.map web page, 194
- for header and initial HTML for creating web page, 176
- of HTML initialization file fourth_i.html, 160
- for the HTML initialization file second_i.html, 53
- for HTML initialization file third_i.htm, 100
- for HTML template second.html, 53–54
- for HTML template third.html, 100–101
- for identifying HEADER and FOOTER in fourth.map mapfile, 141
- for identifying the path to the Python MapScript mapfile, 192
- for indicating an EMPTY result set, 152
- for initializing query-related CGI variables to nulls, 146
- for inserting the legend image and navigation variables, 217
- for inserting the map scale, click point coordinates, and map extent, 217
- for installing PHP, 208
- for invoking CreateTImagemap() function, 255
- JavaScript code for tailoring navigation interface to a browser, 260
- JavaScript creating a table with three each rows and columns, 259
- for joining an external table in the fourth.map mapfile, 143
- for the layer-level query HEADER for the Countries layer, 152–153
- of layer-level query template fourth_cities_footer.html, 166
- of layer-level query template fourth_cities_header.html, 166
- of layer-level query template fourth_countries_header.html, 165
- for legend, 86–87
- for loading the Python MapScript and random modules, 189
- for looping through result set elements, 253

- for making MapScript and CGI modules available, 170
- of mapfile `fourth.map`, 156–159
- for mapfile `second.map`, 51–53
- the mapfile `third.map`, 88–99
- of map-level "no results" web page
 - `fourth_empty.html`, 165
- of map-level query template
 - `fourth_web_footer.html`, 164
- of map-level query template
 - `fourth_web_header.html`, 160–164
- for obtaining map coordinate conversions, 180
- for opening the PHP script and naming images, 210
- for performing the spatial query, 262
- the Perl MapScript "Hello World" application, 172
- for Perl MapScript version of `perlms_third.pl`, 181–186
- for PHP/MapScript version of third application, 223–229
- for `phpms_fifth.php`, 273–290
- for placing the php binary in Apache's script directory, 208
- presenting map and reference images, map information, and navigation controls, 147
- for printing the preamble and opening tags for the web page, 195, 216
- for producing form for user interaction with MapServer, 45
- for providing for a virtual mouse click for `second.html` template file, 47
- providing zoom controls for `second.html` template file, 45–46
- for the Python MapScript version of "Hello World" application, 200–201
- for query HEADER and FOOTER templates in `fourth.map` mapfile, 142
- for rebuilding MapScript directly, 238–239
- for reference map for `fourth.map` mapfile, 140–141
- for removing previous Python build configuration options, 188
- for reporting query summary results in `fourth_we_header.html`, 151
- for retrieving a list of layers that user chooses to display, 178, 196, 219
- for retrieving extent as an instance of `rectObj`, 176
- for retrieving table information and creating pop-up tool tips, 252
- for retrieving the extent of the map just saved to disk, 176
- for retrieving the form variable extent, 178, 220
- for retrieving the map scale from the map object, 176, 194
- for returning image coordinates of click point, 178
- for `second.map` application initialization file, 44
- for `second.map` application layer 1: urban areas, 38–39
- for `second.map` application layer 2: water features, 40
- for `second.map` application layer 3: state boundaries, 41
- for `second.map` application layer 4: road network, 42
- for selecting a database for retrieving information, 253
- for setting Countries layer TOLERANCE value in `fourth.html` file, 149
- for setting current values of hidden form variables in `fourth.html` file, 150
- for setting `imgshape` coordinates in `fourth.html` file, 149
- for setting mapfile parameters based on CGI form variables, 145
- for setting `mapshape` coordinates in `fourth.html` file, 149
- setting navigation defaults for `perlms_third.pl`, 174
- for setting PHP/MapScript navigation defaults, 213
- for setting `PointObj` coordinate values in PHP/MapScript, 219
- for setting query string in `fourth.html` file, 149
- for setting the binary check box values in `fourth.html` file, 147
- for setting the CHECKED state for several layers, 192
- for setting the CHECKED state for several layers in PHP/MapScript, 214

- for setting the Cities layer TOLERANCE value in fourth.html file, 149
- for setting the extent of the map in Python MapScript, 197
- for setting the extent of the Perl MapScript map, 178–179
- for setting the imgbox coordinates in fourth.html file, 148
- for setting the navigation defaults for python_third.py file, 192
- for setting the navigation variables to be saved in a form, 197, 220
- for setting the query item in fourth.html file, 148
- for setting the query layer in fourth.html file, 148
- for setting the shapeindex in fourth.html file, 149
- for setting the value of zoomsize in fourth_i.html file, 145
- for setting the values of the rectObj old_extent components, 197, 220
- for setting up second.map image parameters, 36
- showing a list of modes that users can select in fourth.html file, 148
- showing HTML preamble for second.html template file, 44
- for Slurp and Burp Restaurants application mapfile, 265–273
- for specifying a class-level query template for presenting results, 143
- for specifying an ANNOTATION layer in the fourth.map mapfile, 144
- specifying a polygon layer named Countries in fourth.map mapfile, 141
- for specifying hoods layer in Slurp and Burp Restaurants application, 248
- for specifying the element containing the map image, 216
- for specifying the PHP/MapScript layer selection controls, 217
- for starting your Apache server, 21
- of template file fourth.html, 160–164
- for terminating the class, layer, and mapfile, 19–20
- for testing if an arrow key has been clicked, 261
- to untar the FreeType tarball, 8
- to untar the GDAL tarball, 9
- to untar the GD tarball, 9
- to untar the libJPEG tarball, 7
- to untar the libpng tarball, 7
- to untar the Proj.4 tarball, 9
- to untar the shapelib tarball, 10
- using annotation layers, 81–83
- using BigLine symbol, 80
- using Circle symbol, 81
- using CLASSITEM and EXPRESSION keywords, 74
- using DashedLine symbol, 80
- using getLayerByName() method, 254
- using imageObj() constructor draw() to create image map, 175
- using LABELMINSIZE in an annotation layer, 83
- using mapObj() method to create a new Python MapScript object, 190
- using newMapObj() method to create a PHP/MapScript map object, 210
- using ogrinfo to explore the contents of a spatial data set, 304
- using ogrinfo to find attribute names, 303
- using ogrinfo to show unique values, 303
- using OVERLAYSMBOL, OVERLAYSIZE, and OVERLAYCOLOR, 81
- using parms.getFirst() method, 196
- using PHP Perl function fpreg_match() to search for strings, 219
- using Python string-comparison method find(), 196–197
- using randrange() method in Python "Hello World" application, 190
- using SCALEBAR keyword, 83–84
- using the class-level query template in fourth_cities_header.html, 155
- using the draw() method to create the third.map map image, 194, 215
- using the getLayerByName() method, 262
- using the img2map() function in Python MapScript, 198
- for using the setMode() function in fourth.html template, 146–150
- using the strict package in perlms_hello.pl, 169

- using TOLERANCE and
TOLERANCEUNITS in fourth.map
mapfile, 142
- of WEB object, 32
- COLOR CLASS object
 - for specifying color to be used to render
features, 315
- COLOR keyword
 - for determining color in which a feature is
drawn, 19
 - for setting the foreground color of scale
bar, 84
 - for setting value for the RGB color of label
text, 67
 - use of in QUERYMAP object, 135–136
- COLOR LABEL object
 - for specifying color used to render label
text, 324
- COLOR QUERYMAP object
 - for specifying the color used to highlight
features, 341
- COLOR reference map object
 - for specifying color used to draw the
reference box, 342
- COLOR SCALEBAR object
 - for specifying foreground color of a scale
bar, 344
- COLOR STYLE object
 - for specifying color used for drawing
features, 347
- comments
 - using # (pound sign) to denote in
mapfiles, 17
- comparison operators
 - table of for logical expressions, 76
- CONFIG keyword
 - specifying values of environment
variables for use by MapServer
with, 310
- conic projection
 - function of, 376–378
 - schematic representation of, 377
- CONNECTION LAYER object
 - for specifying connection string used to
retrieve data, 327
- CONNECTIONTYPE LAYER object
 - for specifying the type of connection, 327
- CONTEXT CGI variable
 - for specifying the name of a context file, 351

- Countries-Layer templates
 - in MapServer Fourth Application, 152–154
- Cozens, Simon and James Lee
Beginning Perl, Second Edition (Apress,
2004) by, 167
- cylindrical projections
 - PROJECTION object syntax for using, 375
 - pros and cons of using, 374–376
 - schematic representation of, 375

D

- DashedLine symbol
 - using to draw dashed lines, 80
- DATA keyword
 - for identifying the base name of a
shapefile, 25
- DATA LAYER object
 - function of, 328
- data structures
 - for shapefiles, 371–373
- DATAPATTERN keyword
 - function of, 310
- dBase (DBF) files
 - joining, 136–137
- .dbf filename extension, 24
- dbfadd utility
 - for adding single records to a DBF file, 297
- dbfcat utility
 - for appending records in one DBF file to a
second DBF file, 299
- dbfcreate utility
 - creating an empty DBF file with, 297
- dbfdump utility
 - for dumping contents of a DBF file to
STDOUT, 297–298
- dbfinfo utility
 - for displaying information about a DBF
file, 300
 - for finding the names of attributes, 38
- de configuration option
 - function of, 168
- DEBUG CLASS object
 - for turning class debugging on, 315
- DEBUG keyword
 - for turning debugging on and off, 311
- DEBUG LAYER object
 - for turning layer debugging on, 328
- default extent
 - defining as an array in perlms_third.pl, 175

- developable surfaces
 - utility provided by, 373
- development environment
 - for examples in book, 5
- Doyon, Jean-Francois
 - maintenance of MapServer documents by, 309
- DRIVER OUTPUTFORMAT object
 - function of, 337
- DSO (dynamic shared object), 207–208
- DUMP LAYER object
 - return data in GML or raw raster format, 328
- dynamic shared object. *See* DSO (dynamic shared object)

E

- earth
 - the figure of, 380
- ellipse
 - symbol type, 79–81
- ellipsoid
 - defined, 380
- EMPTY keyword
 - use of in map-level query templates, 134
- EMPTY page
 - for indicating an empty result set, 152
- EMPTY WEB-level keyword
 - use of in MapServer, 106
- EMPTY WEB object
 - for specifying the URL of the web page to display if query returns no results, 348
- ENCODING LABEL object
 - for specifying encoding format for a label, 324
- END keyword
 - function of in CGI MapServer web application, 18
- error messages
 - can't be executed because of wrong permissions, 171
 - when script can't find the mapfile, 172
- ERROR WEB object
 - for specifying URL of web page to display if MapServer error appears, 348
- ESRI's Spatial Data Warehouse
 - SDE client libraries as part of, 4

- EXPRESSION CLASS object
 - function of, 315–316
- EXPRESSION keyword
 - possible forms of and function of each form, 39–40
 - specifying a comparison string with, 74–76
 - use of in queries, 104
- expressions
 - using to define classes, 74–76
- EXTENSION OUTPUTFORMAT object
 - function of, 337
- EXTENT keyword
 - for specifying the extent of a map, 311
 - for specifying the geographic extent of a map, 17
 - specifying the geographic extent of the map with, 25
- EXTENT reference map object
 - for specifying the spatial extend of the reference image, 342
- external libraries
 - used by MapServer, 2

F

- FEATURE attribute
 - Unix command-line for displaying values for, 42
- FEATURE keyword
 - specifying an inline geographical feature with, 18–19
- FEATURE LAYER object
 - for indicating the start of an inline FEATURE object, 328
- FEATURE object
 - used for defining an inline feature, 320
- FEATUREQUERY mode
 - vs. FEATUREQUERY mode, 123
 - query example, 120–122
 - spatial queries performed by and all matching features returned, 132
- FEATUREQUERY mode
 - vs. FEATUREQUERY mode, 123
 - query example, 119–120
 - spatial query performed by, 131–132
- file extensions
 - associated with shapefiles, 371

- file header
 - as shapefile component, 371
 - FILLED keyword
 - setting to fill a symbol with a specified color, 80
 - specifying that symbol is to be filled with color defined in CLASS object, 382
 - FILTER keyword
 - use of in queries, 104
 - FILTER LAYER object
 - function of, 328
 - FILTERITEM LAYER object
 - function of, 329
 - FIPS (Federal information processing standards) codes, 37
 - first.html file
 - building for the first map, 28–30
 - loading, 28
 - first.map mapfile
 - creating, 24–28
 - font attributes
 - assigning to labels, 69–70
 - FONT keyword
 - setting font to be used with, 67
 - for specifying the font alias from which CHARACTER will be drawn, 382
 - FONT LABEL object
 - for specifying the alias of the font used to label a feature, 324
 - fonts
 - setting point size for, 67
 - use of in MapServer, 66–67
 - FONTSET examples, 385–386
 - fontset file
 - for telling MapServer where to find a font, 12
 - FONTSET keyword
 - function of, 19, 311
 - FOOTER keyword
 - for specifying template file processed after everything else is done, 134
 - FOOTER LAYER object
 - function of, 329
 - FOOTER WEB object
 - function of, 349
 - FORCE LABEL object
 - for forcing a label to be rendered (available for cached labels only), 324
 - FORMATOPTION OUTPUTFORMAT object
 - for allowing specification of drive- or format-specific options, 337
 - forward slash (/)
 - for delimiting regular expressions, 74–75
 - strings delimited by, 42
 - fourth_cities_footer.html
 - for closing the table opened in the HEADER template, 155
 - fourth_countries_footer.html
 - for sending the FOOTER template back to the browser, 154
 - fourth_i.html
 - initialization file contained in, 145–146
 - fourth_web_header.html
 - code to display result set information as a whole, 151
 - fourth.html template file
 - for MapServer Fourth Application, 146–150
 - fourth.map mapfile
 - for a query application, 139–144
 - FreeType
 - building and installing, 8–9
 - used by GD for rendering fonts, 2
 - website for downloads and documentation, 13
 - FROM JOIN object
 - function of, 322
- ## G
- GAP keyword
 - for specifying distance between characters when rendering TrueType symbols, 382
 - GD library
 - building and installing, 9
 - used by MapServer to render images, 2
 - GDAL (Geospatial Data Abstraction Library)
 - building and installing, 9–10
 - translator library for raster data, 3
 - website for downloads and documentation, 14
 - GDAL/OGR utilities, 301–307
 - getLayerByName() method
 - using to retrieve a reference to layer named poi, 254
 - getLayersByGroupName() method
 - lacking in Python MapScript, 199–200

- Gilmore, W. Jason
 - Beginning PHP 5 and MySQL: From Novice to Professional* (Apress, 2004) by, 207, 235
- GNU Make
 - absolute requirement for building FreeType, 5
- graphical distinction
 - importance of between map features, 65
- GRID LAYER object
 - for indicating the start of a GRID object, 329
- GRID object
 - for defining a map grid within a layer, 320
- GROUP LAYER object
 - for specifying the name of the group to which a layer belongs, 329
- H**
- HandleIE() function
 - code defining, 258–259
- HEADER keyword
 - specifying the path to a template file with, 134
- HEADER LAYER object
 - function of, 329
- HEADER WEB object
 - function of, 349
- “Hello World” application
 - building, 16–23
 - creating the mapfile for, 16–20
 - in Perl MapScript, 169–172
 - PHP/MapScript version of, 211
- Hello World image, 23
- hello.map file
 - code for creating, 17–20
 - using specifications from for building Perl MapScript application, 169–172
- HTML initialization form
 - MapServer invoked by Apache from, 15–16
- HTML legends
 - example generated from the mapfile third.map, 389
 - function of, 386–389
- HTML tags
 - code for generating for displaying a Python MapScript image, 190
- HTML template
 - for second.map application, 43–50
- HTML template and initialization file
 - creating, 20–23
- HTML template file
 - building for the first map, 28–30
 - embedding initial values in, 21–23
 - for MapServer Fourth Application, 146–150
 - for second.map application, 44–50
- I**
- [id]
 - function of in queries, 138
- ID CGI variable
 - for specifying a replacement for the default session ID, 351
- IMAGE keyword
 - specifying file name of GIF or PNG image for pixmap symbols, 383
- image map
 - using imageObj() constructor draw() to create, 175
- IMAGE reference map object
 - for specifying path to a file containing the GIF reference image, 343
- IMAGECOLOR keyword
 - for defining the background color for map images, 17
 - for specifying background color of scale bar image, 84
 - for specifying background color of a map image, 311
 - specifying your image background color with, 24
- IMAGECOLOR LEGEND object
 - for specifying the background color of a legend image, 335
- IMAGECOLOR SCALEBAR object
 - for specifying background color of image on which scale bar is drawn, 345
- imagemap
 - creating for the Slurp and Burp Restaurants application, 255–257
- IMAGEMODE OUTPUTFORMAT object
 - for specifying the image mode used for output, 338
- IMAGEPATH keyword
 - for telling MapServer where to put images it creates, 18

- IMAGEPATH WEB object
 - for specifying path to directory where images and files are written, 349
- IMAGEQUALITY keyword
 - for specifying image quality for JPEG images, 311
- images
 - library used for rendering in MapServer, 2
- images file
 - for MapServer to save images in, 12
- IMAGETYPE keyword
 - for specifying format of the output image, 312
 - for specifying the format of the map image, 17
 - specifying your image type with, 24
- IMAGEURL keyword
 - specifying a URL showing where to an image, 18
- IMAGEURL WEB object
 - for specifying URL pointing to directory where images are written, 349
- IMG CGI variable
 - using to identify input image to MapServer, 351
- img, img.x, and img.y query CGI variables
 - function of, 138
- IMGBOX CGI variable
 - representing the coordinates of opposite corners of an image drag box, 351
- imgbox coordinates
 - code for setting in fourth.html file, 148
- imgext [minx] [miny] [maxx] [maxy] query CGI variables
 - function of, 138
- IMGEXT CGI variable
 - function of, 351
- IMGSHAPE CGI variable
 - for specifying vertex coordinates of a user-defined polygon, 352
- imgshape coordinates
 - code for setting in fourth.html file, 149
- IMGSIZE CGI variable
 - for specifying the size of the map image, 352
- imgxy [x] [y] query CGI variable
 - function of, 138
- IMGXY CGI variable
 - containing the image coordinates of a mouse click on a map image, 352
- Independent JPEG Group website
 - directory list, 13
- INDEXQUERY mode
 - query example, 126–127
 - retrieval of a single feature based in shape index, 133
- INDEXQUERYMAP mode
 - map image produced by, 114
- initialization file
 - contained in the fourth_i.html file, 145–146
 - for second.map application, 43–44
- initialization file and HTML template
 - creating, 20–23
- INTERLACE keyword
 - for turning image interlace on or off, 312
- INTERLACE LEGEND object
 - for turning legend image interlace on or off, 335
- INTERLACE SCALEBAR object
 - for turning scale bar image interlace on or off, 345
- INTERNAL keyword
 - values associated with, 84
- interstate highway layers
 - code for drawing, 73
- interstate1 layer
 - adding labels to, 71
- INTERVALS SCALEBAR object
 - for specifying number of intervals shown on a scale bar, 345
- [itemname], [itemname_esc], and [itemname_raw]
 - function of in queries, 137
- ITEMFEATUREQUERY mode
 - query example, 124–126
 - searches performed by and matches returned, 133
- ITEMFEATUREQUERY mode
 - query example, 123–124
 - searches performed by and matches returned, 132
- ITEMNQUERY mode
 - attribute searches performed by and all matches returned, 131
 - query example, 117–119
 - for searching all queriable layers and returning results, 105

ITEMQUERY mode

- attribute searches performed by and first match returned, 131
- query example, 116–117
- setting TOLERANCE value in, 116–117
- using in MapServer, 105

J

JavaScript

- using to fake out IE in Slurp and Burp Restaurants application, 257–260

JavaScript library

- overlib created by Erik Bostrup, 238

JavaScript tool tip code

- installing, 238

JOIN CLASS object

- for indicating the start of a JOIN object, 316

JOIN LAYER object

- for indicating the start of a JOIN object, 329

JOIN object

- function of, 136–137, 321–323

[joinname_itemname],

[joinname_itemname_esc], and

[joinname_itemname_raw]

- function of in queries, 137

K

KEYSIZE LEGEND object

- for specifying the size of symbol key boxes, 335

KEYSPACING LEGEND object

- for specifying spacing between labels and symbols in a legend, 335

keyword-value pairs

- in mapfile definitions, 17

Knuth, Donald

The TeXbook (Addison Wesley, 1986) by, 61

Koormann, Frank

- maintenance of MapServer documents by, 309

L

LABEL CLASS object

- for indicating the start of a LABEL object, 316

LABEL LEGEND object

- for indicating the start of a LABEL object, 335

LABEL object

- assigning font attributes to labels in, 69–70
- code for defining in the fourth.map mapfile, 143

for defining a text string or symbol used to label a feature, 323

for specifying font type, size, and color of a label, 19, 27

LABEL SCALEBAR object

- for indicating the start of a LABEL object, 345

LABELANGLEITEM LAYER object

- function of, 330

LABELCACHE keyword

- for caching labels, 70

LABELCACHE LAYER object

- for turning the label cache on or off, 330

LABELFORMAT GRID object

- function of, 320

LABELITEM keyword

- specifying the attribute name with, 71

LABELITEM LAYER object

- function of, 330

LABELMAXSCALE keyword

- for setting the maximum scale at which labels will be rendered, 72

LABELMAXSCALE LAYER object

- for specifying maximum scale at which labels will be rendered, 330

LABELMINSCALE keyword

- setting the minimum scale at which labels will be rendered, 72

LABELMINSCALE LAYER object

- for specifying minimum scale at which labels will be rendered, 330

LABELREQUIRES LAYER object

- function of, 330

labels

adding text to, 71

assigning font attributes to, 69–70

caching, 70

positioning, 68–69

positioning with the POSITION keyword, 70

setting orientation of, 67–71

setting the maximum scale at which they will be rendered, 72

specifying number of pixels between, 69

wrapping, 70

LABELSIZEITEM LAYER object

- for specifying the size at which a label will be rendered, 331

lakes. *See* layer 2: water features; water features: layer 2

- Lambert Conformal conic projection
 - syntax for, 377–378
- layer 2: water features
 - contents of for second.map application, 39
- layer 3: state boundaries
 - code for, 41
- layer 4: road network
 - contents of for second.map application, 41–42
- LAYER CGI variable
 - for specifying the name of a layer, thereby setting its STATUS to on, 352
- LAYER keyword
 - for indicating the start of a LAYER object, 312
 - for introducing a layer, 18
- LAYER object
 - for determining what spatial data is to be rendered, 327
 - function of in digital mapping processes, 33–34
 - for telling MapServer what to render, 25
- layer-level FOOTER template
 - for closing the table opened in the HEADER template, 155
 - for sending the FOOTER template back to the browser, 153–154
- layer-level HEADER template
 - in fourth_cities_header.html file, 154–155
 - in the fourth-countries_header.html file, 152–153
- layer-level query templates
 - useful for multi-result queries, 134
- layers
 - setting the maximum scale at which they will be rendered, 72
- LAYERS CGI variable
 - for specifying a space-delimited list of layer names, 352
- Lee, James and Simon Cozens
 - Beginning Perl, Second Edition* (Apress, 2004) by, 167
- legend image
 - code for defining file name and URL for in perlms_third.pl, 175
- LEGEND keyword
 - beginning a legend with, 85
 - for indicating the start of a LEGEND object, 312
- LEGEND object
 - for determining the appearance and location of a legend, 334–336
- legend utility
 - for reading a mapfile and creating a legend image on its contents, 292
- legends
 - creating in MapServer, 85–87
 - example of embedded, 86
- libcurl
 - client-side URL-transfer library, 3
- libJPEG
 - building and installing, 7–8
 - used by MapServer to render JPEG images, 3
- libpng
 - building and installing, 7
 - library of routines for rendering PNG images, 3
 - website for downloads and documentation, 13
- libraries
 - selecting supporting used by MapServer, 1–4
 - specifying paths to with command-line options, 11
 - steps for building, 6
- Lime, Steve
 - maintenance of MapServer documents by, 309
- line layer
 - code for annotation layer, 82
 - for rendering a series of points as a connected sequence, 25
- line type shapefile, 371
- LINECAP keyword
 - specifying how a CARTOLINE symbol is terminated, 382
- LINEJOIN keyword
 - specifying how the intersection of two lines will be rendered, 382
- LINEJOINMAXSIZE keyword
 - specifying the length of overrun on miter type line joins, 382
- LOG WEB object
 - for specifying file where MapServer activity will be logged, 349
- logical expression comparison operators
 - table of, 76

M

- man pages
 - website address for, 74
- map
 - building the first in your MapServer application, 23–30
 - building the HTML template for the first map, 28–30
- MAP CGI variable
 - containing the full path to the mapfile, 352
- map coordinates
 - code for calculating map width and height of extent in map units, 180
- map images
 - code for defining file name and URL for in `perlms_third.pl`, 175
 - creating and saving in Slurp and Burp Restaurants application, 257
- map object
 - mapfile keywords, 310–315
 - simple and structured items in, 32–33
- map projections
 - ways to categorize, 373–381
- map symbols
 - creating legends for, 85–87
- MAPEXT CGI variable
 - function of, 352
- mapfile
 - code for Slurp and Burp Restaurants application, 265–273
 - concepts, 31–34
 - creating for the Hello World application, 16–20
 - function of in CGI MapServer web application, 15–16
 - with navigation controls and layer selection, 35–43
 - objects included in each, 17
 - structure of, 32–33
 - syntax, 34
- mapfile keywords, 310–315
- "MapFile Reference - MapServer 4.4" document
 - website address for, 34
- map-level query templates
 - keywords that specify query templates, 133–134
 - in MapServer Fourth Application, 150–152
 - steps for creating a complete web page in, 134
- map-only query modes
 - function of, 107–108
- maps
 - busy map produced by the second mapping application, 56
 - digital vs. paper, 66
 - effect of further scale increases on feature density, 59
 - effect of increasing scale by a factor of two, 58
 - factors involved in optimum information density, 60–61
 - the graphic design of, 61–66
 - importance of graphical distinction between features, 65
 - labeling for clarity, 66–71
 - modifying the look and feel of, 55–101
 - reducing detail by deselecting Roads layer, 56–57
 - using scale to reduce clutter on, 71–74
- MapScript
 - extending the capabilities of with MySQL, 231–290
- MapScript web page
 - code for finishing, 176–177
- MapServer
 - basic concepts, 15–16
 - basic knowledge needed before building, 1
 - building and installing, 1–14
 - classifying features in, 74–81
 - fourth.map mapfile, 139–144
 - how queries are processed by, 103–108
 - HTML required by, 20–21
 - making the executable accessible to Apache, 11–12
 - OUTPUTFORMAT implicit declarations, 338–340
 - planning the installation, 1–4
 - query types in, 104–105
 - replacement of a substitution string with a value, 137
 - selecting supporting libraries used by, 1–4
 - setting label color for, 67
 - simple examples, 15–30
 - substitution strings used in, 357–368
 - telling it where to find a font, 12
 - use of fonts in, 66–67
- MapServer and Apache
 - configuring, 12–13

- MapServer Application Gallery
 - the website address, 232
- MapServer components
 - purposes of, 309–368
- MapServer documents
 - website address for, 309
- MapServer Fourth Application
 - FEATUREQUERY mode query example
 - and results, 120–122
 - FEATUREQUERY mode query results, 120
 - INDEXQUERY mode returns a single
 - feature based on shape index, 127
 - initial display of the query definition
 - page, 111
 - ITEMFEATUREQUERY mode results
 - with larger TOLERANCE for the Cities layer, 126
 - ITEMQUERY mode results when
 - searching Countries layer, 119
 - mapfile for, 139–144
 - NQUERY mode multiple results produced
 - and states and cities displayed, 115
 - query initialization page, 110
 - query templates in, 150–155
 - result page for ITEMNQUERY mode
 - matches from Cities layer, 118
 - result page for QUERY mode displaying a single city, 112
 - result page for QUERY mode displaying a single state, 113
 - results page from
 - ITEMFEATUREQUERY mode
 - query, 125
 - results page from ITEMFEATUREQUERY mode query, 124
 - showing an NQUERY mode spatial search
 - using a doughnut-shaped search, 129
 - showing an NQUERY mode spatial search
 - using a polygon search, 128
- MapServer reference, 309–368
- MapServer utilities, 291–296
- MapServer Version 4.4.1
 - case sensitivity of, 34
- MAPSHAPE CGI variable
 - containing the user-defined polygon in map coordinates, 353
- mapshape coordinates
 - code for setting in fourth.html file, 149
- MAPSIZE CGI variable
 - containing the image size of the map to be created, 352
- MAPXY CGI variable
 - function of, 353
- MARKER reference map object
 - for specifying symbol used when reference box is too small, 343
- MARKERSIZE reference map object
 - for specifying size of symbol used when reference box is too small, 343
- MarkSpot() function
 - invoking in the Slurp and Burp Restaurants application, 255–256
- MAXARCS GRID object
 - for specifying the maximum number of arcs to be drawn, 320
- MAXBOXSIZE reference map object
 - for specifying the maximum reference box size, 343
- MAXFEATURES LAYER object
 - for specifying maximum number of features rendered in a layer, 331
- MAXINTERVAL GRID object
 - for specifying the maximum interval between grid lines, 321
- MAXSCALE CLASS object
 - for specifying maximum scale at which the class will be rendered, 316
- MAXSCALE keyword
 - for setting the maximum scale at which a layer will be rendered, 72
- MAXSCALE LAYER object
 - for specifying the maximum scale at which a layer will be rendered, 331
- MAXSCALE WEB object
 - for specifying maximum scale at which a map will be returned, 349
- MAXSIZE CLASS object
 - for specifying maximum size at which a symbol will be drawn, 317
- MAXSIZE LABEL object
 - for specifying the maximum font size for scaled labels, 325
- MAXSIZE STYLE object
 - for specifying maximum size at which a symbol will be drawn, 347

- MAXSUBDIVIDE GRID object
 - for specifying the maximum number of segment to render a grid line, 321
 - MAXTEMPLATE WEB object
 - function of, 350
 - MAXX CGI variable
 - function of, 353
 - MAXY CGI variable
 - function of, 353
 - McKenna, Jeff
 - maintenance of MapServer documents by, 309
 - Mercator projection, 375–376
 - syntax for, 376
 - METADATA LAYER object
 - for allowing data to be stored as name-value pairs, 331
 - METADATE WEB object
 - function of, 350
 - MIMETYPE OUTPUTFORMAT object
 - for specifying the mime type used for the result, 338
 - MINARCS GRID object
 - for specifying the minimum number of arcs to be drawn, 321
 - MINBOXSIZE reference map object
 - for specifying the smallest reference box size, 343
 - MINDISTANCE keyword
 - for specifying number of pixels between label, 69
 - MINDISTANCE LABEL object
 - for specifying the minimum distance between identical labels, 325
 - MINFEATURESIZE keyword
 - for specifying size of smallest feature to be labeled, 69
 - MINFEATURESIZE LABEL object
 - for specifying the minimum size at which a feature will be labeled, 325
 - Ming
 - for creating SWF (Shockwave Flash) movies, 4
 - MININTERVAL GRID object
 - for specifying the minimum interval between grid lines, 321
 - MINSCALE CLASS object
 - for specifying minimum scale at which the class will be rendered, 317
 - MINSCALE keyword
 - setting the minimum scale at which layers will be rendered, 72
 - MINSCALE LAYER object
 - for specifying the minimum scale at which a layer will be rendered, 331
 - MINSCALE WEB object
 - for specifying minimum scale at which a map will be returned, 350
 - MINSIZE CLASS object
 - for specifying minimum size at which a symbol will be drawn, 317
 - MINSIZE LABEL object
 - for specifying the minimum font size for scaled labels, 325
 - MINSIZE STYLE object
 - for specifying minimum size at which a symbol will be drawn, 347
 - MINSUBDIVIDE GRID object
 - for specifying the minimum number of segments to render a grid lines, 321
 - MINTEMPLATE WEB object
 - function of, 350
 - MINX CGI variable
 - function of, 353
 - MINY CGI variable
 - function of, 353
 - MODE CGI variable
 - function of and mode values supported, 353–355
 - mouse-click point
 - converting from image coordinates to map coordinates, 176
 - Mozilla vs. IE
 - design issues regarding use of, 233–234
 - MySQL
 - excellent reference book for, 235
 - extending the capabilities of MapScript with, 231–290
 - MySQL database
 - creating, 234–237
- N**
- NAME CLASS object
 - for specifying name for the class for use in the legend, 317
 - NAME JOIN object
 - for specifying the join name, 322

NAME keyword

- for assigning a name to a symbol, 79–81
- function of in CGI MapServer web application, 17
- naming your first.map mapfile with, 24
- specifying name of layer with, 25
- specifying the name used to access a symbol, 383
- for specifying the name used to identify map output, 312

NAME LAYER object

- for specifying the name of the layer, 331

NAME OUTPUTFORMAT object

- used by the mapfile keyword IMAGETYPE to reference the format, 338

National Institute of Standards and Technology (NIST)

- codes issued by, 37

navigation defaults

- setting for perlms_third.pl, 174

newMapObj() method

- using to create a new PHP/MapScript map object, 210

NQUERY mode

- area searches and point queries performed by, 130–131
- matches returned in a higher layer TOLERANCE, 116
- query example, 114–116
- for searching all queriable layers and returning results, 105

NQUERY mode with polygon search region

- query example, 127–129

[nr], [nl], and [nlr]

- function of in queries, 137

O**oblate spheroid**

- the figure of the earth, 380

Oblique Mercator projection, 375**OFFSET LABEL object**

- for specifying offset of lower-left corner of a label from label point, 325

OFFSET STYLE object

- for specifying the offset for shadows, 347

OFFSITE LAYER object

- for setting the transparent color for raster layers, 332

OGR Simple Features Library

- for access to read and write vector formats, 3

ogr2ogr utility

- for converting spatial data sets from one format to another, 304–306

ogrinfo utility

- for displaying information about a data set in a OGR-supported format, 302–304
- for geographic information and feature values, 38

ogrindex utility

- function of, 307

online resources

- list of for MapServer builds, installation, and usage, 13–14

Oracle Spatial client libraries

- for giving MapServer access to Oracle Spatial Data Warehouse, 4

OUTLINECOLOR CLASS object

- for specifying the outline color (for polygons only), 317

OUTLINECOLOR keyword

- for specifying color of the border around the scale bar, 84

OUTLINECOLOR LABEL object

- for specifying color used to create outline around label text, 325

OUTLINECOLOR LEGEND object

- for specifying the outline color for symbol key boxes, 335

OUTLINECOLOR reference map object

- for specifying color used to outline reference box, 343

OUTLINECOLOR SCALEBAR object

- for specifying the color used to outline intervals, 345

OUTLINECOLOR STYLE object

- for specifying the outline color (for polygons only), 348

OUTLINECOLOR value

- for drawing an outline around text, 67

output formats

- supported by libraries, 1

OUTPUTFORMAT implicit declarations

- made if no explicit declarations are found in the mapfile, 338–340

OUTPUTFORMAT object
 for defining and naming output formats, 336–340

OVERLAYBACKGROUNDCLASS object
 for specifying color used to render overlay symbols, 317

OVERLAYCOLOR CLASS object
 for specifying color used for drawing features with overlay symbols, 318

OVERLAYCOLOR keyword
 example using, 81

OVERLAYMAXSIZE CLASS object
 for specifying maximum size in pixels an overlay symbol can be drawn, 318

OVERLAYMINSIZE CLASS object
 for specifying minimum size at which an overlay symbol can be drawn, 318

OVERLAYOUTLINECOLOR CLASS object
 for specifying the outline color for an overlay symbol (for polygons only), 318

OVERLAYSIZE CLASS object
 function of, 318

OVERLAYSIZE keyword
 example using, 81

OVERLAYSYMBOL CLASS object
 for specifying the overlay symbol used for drawing features, 318

OVERLAYSYMBOL keyword
 example using, 81

P

parms.getFirst() method
 code showing use of, 196

PARTIALS LABEL object
 for rendering partial labels, 326

passwords
 importance of using unique for security purposes, 253

PDFlib
 provides ability to produce output as PDFs, 4

Perl
 building, 168–169
 configuring, 168
 created by Larry Wall in late 1980s, 167–168

Perl MapScript
 building, 169
 building and installing, 168–169
 figure of the "Hello World" application, 171
 "HelloWorld" application, 169–172
 importance of checking for typos, 172
 an interactive map application, 172–180
 troubleshooting the "Hello World" application, 171
 using, 167–186

Perl MapScript "Hello World" application
 code for, 172

Perl MapScript Third Map application
 perlms_third.pl version, 173
 perlms_third.pl
 Perl MapScript Third Map application version, 173
 setting the CHECKED state for several layers, 174

PHP
 building, 208–209
 website address for downloading, 208

PHP MapScript
 patching to retrieve the shape index, 238–239

PHP script
 first invocation for Slurp and Burp Restaurants application, 251–252

phpinfo.php page
 showing the MapScript and MySQL modules are loaded, 235

PHP/MapScript
 building and installing, 207–210
 using, 207–229

PHP/MapScript application
 checking if MySQL support is included in, 235
 code for navigation defaults, 213
 img2map height-to-latitude conversion, 212–213
 img2map width-to-longitude conversion, 212–213
 original projection of the spatial data used for, 380
 producing an interactive map, 212–222

PHP/MapScript "Hello World" application
 creating, 210–212

- phpms_fifth.php
 - code for Slurp and Burp Restaurants application, 273–290
- phpms_hello.php
 - website for code distribution, 210
- phpms_third.php
 - downloadable from Apress website, 212
 - PHP/MapScript version of the third application, 218
- pixmap
 - symbol type, 79–81
- poi mapfile layer
 - for Slurp and Burp Restaurants application. *See* points-of-interest layer
- point layer
 - defining one named Cities in fourth.map mapfile, 142
 - used to render spatial data as isolated points, 25
- point queries
 - performed by QUERY mode, 130
- point type shapefile, 371
- POINTS FEATURE object
 - for specifying coordinate pairs representing vertices of a shape, 320
- POINTS keyword
 - specifying the coordinates of points that constitute a vector symbol, 383
 - using to describe a list of coordinate pairs, 18–19
- points-of-interest layer
 - for Slurp and Burp Restaurants application, 250
- polygon layer
 - for rendering a series of points as an area-enclosing figure, 26
- polygon type shapefile, 372–373
- POSITION keyword
 - for specifying the placement of a label, 68–69
 - using to position labels, 70
 - values associated with, 84
- POSITION LABEL object
 - for specifying position of label with respect to the label point, 326
- POSITION LEGEND object
 - for specifying the position of the embedded legend, 336
- POSITION SCALEBAR object
 - for specifying position of embedded scale bar, 345
- PostgreSQL client libraries
 - for giving MapServer access to PostGIS data, 4
- POSTLABELCACHE LAYER object
 - function of, 332
- POSTLABELCACHE LEGEND object
 - function of, 336
- POSTLABELCACHE SCALEBAR object
 - function of, 346
- PROCESSING LAYER object
 - for specifying a processing directive for a layer, 332
- Proj.4
 - building and installing, 9
 - library of cartographic projection routines, 3
 - website for downloads and documentation, 14
- Proj.4 library
 - projections available in, 379
- projection categories, 373–381
- PROJECTION keyword
 - for indicating the start of a PROJECTION object, 312
- PROJECTION LAYER object
 - for indicating the start of a PROJECTION object, 332
- PROJECTION object
 - specifying map projection used for spatial data, 340–341
- Python
 - building and installing from the source distribution, 187–188
- Python MapScript
 - building and installing, 187–189
 - code for defining the default extent as an array, 193
 - code for defining the pointObj() object, 192
 - code for "Hello World" application, pythonms_hello.py, 200–201
 - code for pythonms_third.py, 201–206
 - displaying resulting web page, 199
 - figure showing the "Hello World" application, 191
 - vs. MapServer, 187

- a practical application for producing an interactive map, 191–200
 - using, 187–206
- Python MapScript application
 - code for identifying the path to the mapfile, 192
- Python MapScript "Hello World" application
 - creating, 189–191
 - using mapObj() method to create a new object in, 190
 - using randrange() method in, 190
- pythonms_hello.py
 - website address for downloading, 189
- pythonms_third.py
 - figure showing the Python MapScript version of, 195
- Python MapScript code for, 201–206

Q

- qitem [name] query CGI variable
 - function of, 139
- QITEM CGI variable
 - function of, 355
- qlayer [name] query CGI variable
 - function of, 139
- QLAYER CGI variable
 - for restricting a search to a single layer, 355
- QSTRING CGI variable
 - containing the query string, 355
- qstring query CGI variable
 - function of, 139
- quadtree
 - conceptual view of hierarchical extents that comprise, 294
- queriable layer
 - needed for MapServer to perform a query, 104
- query CGI variables
 - available to MapServer query applications, 138–139
- query examples, 108–129
- query item
 - code for setting in fourth.html file, 148
- query layer
 - for associating a mouse click with a specified data set, 26
 - code for setting in fourth.html file, 148
- QUERY mode
 - maintaining state in, 107
 - point query performed by, 130
 - query example, 109–114
 - using in MapServer, 103–166, 104–105
- query modes
 - functions of, 129–133
 - summary of main features of each, 109
- query parameters
 - saving to a queryfile, 107
- query string
 - code for setting in fourth.html file, 149
- query substitution strings
 - function of in queries, 137–138
- query templates
 - class-level, 135
 - displaying attributes from joined tables in, 137
 - functions of, 105–106, 133–135
 - layer-level, 134
 - levels at which they can be defined, 105–106
 - map-level, 133–134
 - in MapServer Fourth Application, 150–155
 - structure of the for providing complete web page, 106
- query types
 - in MapServer, 104–105
- queryfile [filename] query CGI variable
 - function of, 139
- QUERYFILE CGI variable
 - for specifying a queryfile that's loaded before any other processing, 356
- QUERYMAP keyword
 - for indicating the start of a QUERYMAP object, 313
- QUERYMAP object
 - for determining how query results will be rendered, 341–342
 - function of in query modes, 135–136
 - specifying querymaps in mapfile by, 107
- querymaps
 - defined, 107
- quotation marks. *See* quotes
- quotes
 - enclosing keyword values with in strings, 17

R

- raster layer
 - for rendering a georeferenced image, 26
- record header
 - as shapefile component, 371
- rectObj \$old_extent
 - code for setting values of components of, 179
- REF CGI variable
 - for identifying input tag containing the reference map image, 356
- reference
 - MapServer, 309–368
- REFERENCE keyword
 - for indicating the start of a REFERENCE object, 313
- reference map image
 - code for defining file name and URL for in perlms_third.pl, 175
- reference map object
 - for determining the characteristics of the reference map, 342–344
- reference maps
 - using in MapServer, 87
- REFXY CGI variable
 - function of, 356
- REQUIRES LAYER object
 - function of, 332
- RESOLUTION keyword
 - for specifying the resolution of the output display in pixel per inch, 313
- [rn], and [lrm]
 - function of in queries, 138
- road network: layer 4
 - contents of for second.map application, 41–42
- Roads layer
 - reducing map detail by deselecting, 56–57
- root privileges
 - needed for installing and building MapServer software, 5

S

- savequery [true] [false] query CGI variable
 - function of, 139
- SAVEQUERY CGI variable
 - function of, 356
- scale bars
 - creating, 83–85
- SCALE CGI variable
 - function of, 356
- SCALE keyword
 - setting scale of a map with, 313
- SCALEBAR keyword
 - for indicating the start of a SCALEBAR object, 313
- SCALEBAR object
 - color-related keywords used in, 84
 - for creating scale bars in MapServer, 83–85
 - for defining the scale bar, 344–346
- scalebar utility
 - for reading a mapfile and creating a scale bar image on its contents, 292
- SDE client libraries
 - part of ESRI's Spatial Data Warehouse, 4
- SEARCHMAP CGI variable
 - function of, 356
- second_i.html
 - code file for, 53
- second.html template file
 - code containing the substitution string, 45
 - code file for, 53–54
 - code for defining checkboxes, 46
 - code for producing form for user interaction with MapServer, 45
 - code for providing a virtual mouse click, 47
 - code for providing zoom controls, 45–46
 - contents of for second.map application, 44–50
- second.map application
 - building with navigation controls and layer selection, 35–43
 - coast of northern California as rendered by, 49
 - code file for, 51–53
 - code for defining the WEB object parameters, 36
 - code for layer1: urban areas, 38–39
 - code for layer 2: water features, 40
 - code for layer 4: road network, 42
 - code for setting up the image parameters, 36
 - displaying initialization file for, 47
 - effect of inappropriate detail in, 63–64
 - how defects in relate to MapServer features, 61–66
 - initial display of the second map at full extent, 48

- layer 1: urban areas, 37–39
- layer 2: water features, 39–41
- Midwest and eastern United States as rendered by, 50
- SHADOWCOLOR LABEL object
 - for specifying the shadow color, 326
- SHADOWSIZE LABEL object
 - for specifying the offset of the shadow behind the label text, 326
- shape index
 - setting to the value of the store ID, 255
- shapefile
 - function of, 24
- Shapefile C library
 - routines for writing C programs that can read, write, and update shapefiles, 297–301
- shapefile specification, 369–373
 - attribute information, 370
 - data structures, 371–373
 - file extensions associated with shapefiles, 371
 - file structure and file-naming conventions, 370–371
- shapeindex
 - code for setting in fourth.html file, 149
- SHAPEINDEX CGI variable
 - function of, 356
- shapelib
 - building and installing, 10–11
 - C routines for creating and manipulating shapefiles, 3
- shapelib website
 - for downloads and documentation, 14
- SHAPEPATH keyword
 - using to find the directory containing shapefiles, 25
- shapes file
 - for MapServer to get its spatial data from ESRI shapefiles, 12
- SHAREPATH keyword
 - for specifying the path to shapefiles, 313
- .shp filename extension, 24
- shp2img utility
 - for reading a mapfile and creating a map image on its contents, 291–292
- shpadd utility
 - for adding a single feature to a shapefile, 298
- shpcat utility
 - for appending features of one shapefile to another, 300
- shpcentrd utility
 - function of, 301
- shpcreate utility
 - for creating an empty shapefile, 298
- shpdump utility
 - for dumping contents of a shapefile to STDOUT, 298–299
- shpdxfl utility
 - for creating a DXF graphic file from a shapefile, 301
- shpinfo utility
 - for displaying information about a shapefile, 300
- shpproj utility
 - for re-projecting a shapefile using the Proj.4 library, 301
 - syntax for, 380
 - using to re-project spatial data, 380–381
- shprewind utility
 - function of, 299
- shptree utility
 - function of, 293–294
- shptreevis utility
 - function of, 294–295
- .shx filename extension, 24
- signposts
 - for increasing the utility of a good map, 65–66
- SIZE CLASS object
 - for specifying height of a symbol in pixels, 319
- SIZE keyword
 - setting font point size with, 67
 - setting the size of a symbol with, 79–81
 - for specifying the final map image dimensions, 17
 - for specifying width and height of map images in pixels, 314
 - specifying your image size with, 24
 - values associated with, 84
- SIZE LABEL object
 - for specifying the size of label text, 326
- SIZE QUERYMAP object
 - for specifying size of the querymap image, 342

- SIZE reference map object
 - for specifying the width and height of the reference image, 344
- SIZE SCALEBAR object
 - for specifying the size of the scale bar, 346
- SIZE STYLE object
 - function of, 348
- SIZEUNITS LAYER object
 - for setting the units of the CLASS object SIZE, 332
- slayer (selection layer)
 - value of in Fourth Application query mode, 132
- SLAYER CGI variable
 - containing the name of the select layer for feature query modes, 357
- slayer query CGI variable
 - function of, 139
- Slurp and Burp Restaurants application
 - addressing some design issues, 233–239
 - adding features to a layer, 254–255
 - application in action, 239–247
 - building, 239–265
 - checking the creation of the store table, 237
 - code defining HandleIE() function, 258–259
 - code for the mapfile, 265–273
 - creating and saving map images, 257
 - creating the application user account, 237
 - creating the database and producing a list of tables in, 236–237
 - creating the imagemap, 255–257
 - creating the mapfile for, 247–251
 - cursor is sitting on a cup in IE but tool tip is not displayed, 247
 - describing requirements for, 232–233
 - design issues regarding use of Mozilla vs. IE, 233–234
 - downloading code for, 239
 - importance of limiting the select statement in, 253
 - initial display in IE with navigation arrows, 241
 - initial display in Netscape Navigator, 240
 - invoking the MarkSpot() function in, 255–256
 - JavaScript code creating a table with three each rows and columns, 259
 - multiple query results displayed in IE, 245
 - multiple query results displayed in Netscape Navigator, 244
 - PHP script for (phpms_fifth.php), 273–290
 - a pop-up tool tip displayed in Netscape Navigator, 242
 - a pop-up tool tip displayed in IE, 243
 - retrieving dynamic information, 252–254
 - setting the hot spot in, 255
 - special handling required for browsers to function properly, 233–234
 - tool tip displayed in Netscape Navigator while mode is set to Query, 246
- software
 - building and installing for MapServer and libraries, 5–11
 - required to build MapServer, 4–5
- software licensing
 - for software for building MapServer, 4
- sortshp utility
 - function of, 292–293
- spatial data
 - data sets in used for example in book, 23–24
 - under the shapefile model, 369
 - using shpproj to re-project, 380–381
 - website address for obtaining, 23
- spatial data sets
 - untarring to the data directory, 24
- spatial information
 - in shapefile file structure, 370–371
- spatial queries
 - performed by FEATUREQUERY mode, 131–132
 - performing for Mozilla-like browsers, 260–263
 - provided by MapServer, 103–104
- spatial query results
 - displaying, 263–265
- state
 - maintaining in query mode, 107
- state boundaries: layer 3
 - code for, 41
- stateless protocol
 - MapServer web application based on, 15
- STATUS keyword
 - for specifying whether the map image is created, 314
 - use of in QUERYMAP object, 135
 - using default value with, 18

- values associated with, 84
 - values that can be assumed by, 25
 - STATUS LAYER object
 - function of, 333
 - STATUS LEGEND object
 - function of, 336
 - STATUS QUERYMAP object
 - for specifying whether or not a querymap image will be created, 342
 - STATUS reference map object
 - for specifying if a reference image is created or not created, 344
 - STATUS SCALEBAR object
 - function of, 346
 - stereographic projection
 - schematic representation of, 379
 - syntax for, 378–379
 - structured object
 - example of, 32
 - STYLE 1 scale bar
 - example of, 85
 - STYLE CLASS object
 - for indicating the start of a STYLE object, 319
 - STYLE keyword
 - setting a dash pattern or style with, 80
 - specifying the succession of pixels on and off in a dashed-line style, 383
 - use of in QUERYMAP object, 135
 - values associated with, 84
 - STYLE object
 - elements determine how symbols will be rendered, 347–348
 - function of in CGI MapServer web application, 19
 - parameters contained in, 27
 - STYLE QUERYMAP object
 - for specifying how features will be rendered, 342
 - STYLE SCALEBAR object
 - for specifying the scale bar style, 346
 - STYLEITEM LAYER object
 - function of, 333
 - substitution strings
 - and CGI variables, 137–139
 - in HTML templates, 16
 - importance of enclosing keyword values in quotes, 17
 - used in MapServer, 357–368
 - sym2img utility
 - for reading a symbol file and creating symbol image based on its contents, 293
 - SYMBOL CLASS object
 - for specifying the symbol to use for drawing features, 319
 - symbol definition reference, 381–383
 - SYMBOL keyword, 381
 - for indicating the start of a SYMBOL object, 79, 314
 - SYMBOL STYLE object
 - for specifying the symbol used for drawing features, 348
 - symbol types, 79–81
 - symbols. *See also* map symbols
 - creating and using, 381–385
 - setting antialiasing on or off for, 80
 - using, 79–81
 - SYMBOLSCALE LAYER object
 - for specifying the scale at which a symbol appears at its full size, 333
 - symbolset file
 - for allowing MapServer to create symbols on the fly, 12
 - SYMBOLSET keyword
 - contains symbol definitions, 314, 381
- ## T
- TABLE JOIN object
 - function of, 322
 - TEMPLATE CLASS object
 - function of, 319
 - TEMPLATE JOIN object
 - function of, 322
 - TEMPLATE keyword
 - specifying path to the file for each query result, 135
 - for specifying the name of the HTML template, 18
 - TEMPLATE LAYER object
 - for specifying name of HTML file used to display query results, 333
 - TEMPLATE WEB object
 - function of, 350
 - TEMPLATEPATTERN keyword
 - function of, 314

- TEXT CLASS object
 - for specifying the text used to label features in a class, 319
 - TEXT FEATURE object
 - for specifying the text string used to label a feature, 320
 - TEXT keyword
 - adding label text with, 71
 - for specifying text string used to label a feature, 18–19
 - The TeXbook* (Addison Wesley, 1986)
 - by Donald Knuth, 61
 - third_i.htm
 - code for HTML initialization file, 100
 - third.html
 - code for HTML template, 100–101
 - third.map application
 - fixes for inappropriate labels in second.map application, 61–63
 - legend generated from, 387
 - mapfile code for, 88–99
 - suppressing detail to enhance information content, 64
 - using to build a practical PHP/MapScript application, 212–222
 - tile4ms utility
 - function of, 296
 - TILEINDEX CGI variable
 - function of, 357
 - TILEINDEX LAYER object
 - for specifying the name of the tile index file for a layer, 333
 - TILEITEM LAYER object
 - function of, 333
 - TO JOIN object
 - for specifying the name of the join item in the table to be joined, 323
 - TOLERANCE LAYER object
 - for specifying the search radius or sensitivity for queries, 334
 - TOLERANCE value
 - setting for Countries layer in fourth.html file, 149
 - setting in fourth.html file, 149
 - setting in NQUERY mode, 114–116
 - TOLERANCEUNITS LAYER object
 - for specifying units of TOLERANCE, 334
 - TRANSFORM LAYER object
 - for transforming spatial data from map to image coordinates, 334
 - TRANSPARENCY LAYER object
 - for setting opacity for a layer, 334
 - TRANSPARENT keyword
 - for making background color of the map transparent, 314
 - specifying the color index of transparent color in a GIF pixmap symbol, 383
 - use of for scale bar image, 84
 - TRANSPARENT LEGEND object
 - for making the background color of a legend transparent, 336
 - TRANSPARENT OUTPUTFORMAT object
 - for specifying if transparency is turned on or off for a format, 338
 - TRANSPARENT SCALEBAR object
 - for making the background color of a scale bar image transparent, 346
 - truetype
 - symbol type, 79–81
 - TrueType fonts, 19
 - vs. bitmapped fonts, 66–67
 - Tube Map of London, England
 - website address for, 33
 - TYPE JOIN object
 - for specifying whether join type is single or multiple, 323
 - TYPE keyword
 - determining type of font used to render a label with, 19
 - specifying fonts with, 66–67
 - specifying the layer type with, 25
 - for specifying the symbol type, 383
 - using point value with, 18
 - values associated with, 25–26
 - TYPE LABEL object
 - for rendering labels using either bitmapped or TrueType fonts, 326
 - TYPE LAYER object
 - for specifying the layer type, 334
- U**
- UNITS keyword
 - for specifying map distance units, 315
 - values associated with, 84
 - UNITS SCALEBAR object
 - for specifying the scale bar units, 346

- unnamed CLASS
 - code for defining in fourth.map mapfile, 142
- urban area polygons
 - code for drawing, 72–73
- urban areas: layer 1
 - attributes described in the text file, 37
 - for second.map application, 37–39
- US Census Bureau
 - website address, 37
- utility programs
 - dbfinfo for finding names of attributes, 38
 - included in MapServer source
 - distributions and libraries, 291–307
 - ogrinfo for geographic information and feature values, 38

V

- vector
 - symbol type, 79–81
- vector formats
 - library providing access to reading and writing, 3
- vector symbols
 - creating, 383–385
 - drawing a non-contiguous, 384
- virtual click point
 - code for creating in perlms_third.pl, 174
 - creating for the python_third.py file, 192
- virtual mouse click
 - code for providing in second.html template file, 47

W

- Wall, Larry
 - Perl created by, 167–168
- water features: layer 2
 - feature attributes for second.map application, 39–41
- WEB FOOTER template
 - map-level query template, 151–152
- WEB HEADER template
 - map-level query template, 150–151
- WEB keyword
 - for embedding a MapServer map in a web page, 17–18
 - for indicating the start of a WEB object, 315

- WEB object
 - for determining which HTML templates MapServer will use, 32
 - for specifying the web interface, 348–350
- web page
 - embedding the map created by MapServer in, 17–18
- website address
 - for a comprehensive technical description of the shapefile format, 369
 - for detailed information about CGI, 15
 - for downloading *Cartographic Projection Procedures for the UNIX Environment--A User's Manual*, 379
 - for downloading FreeType, 2
 - for downloading GDAL (Geospatial Data Abstraction Library), 3
 - for downloading GD library, 2
 - for downloading libcurl, 3
 - for downloading libJPEG, 3
 - for downloading libpng, 3
 - for downloading OGR Simple Features Library, 3
 - for downloading overlib distribution, 238
 - for downloading Perl source distribution, 168
 - for downloading Proj.4, 3
 - for downloading shapelib, 3
 - for downloading software for building MapServer, 5
 - for downloading the latest version of MySQL, 236
 - for downloading the PHP distribution, 208
 - for downloading the python_third.py file, 191
 - for downloading the Python source distribution, 188
 - for downloading zlib, 3
 - for example of Tube Map of London, England, 33
 - for information about the OGR utility programs, 301
 - for the "MapFile Reference - MapServer 4.4" document, 34
 - for the MapServer Application Gallery, 232
 - for MapServer documents, 309

- for the MapServer website, 13
- for obtaining spatial data used in book, 23
- for `phpms_hello.php` code distribution, 210
- for pointers to extensive documentation
 - for libraries, 4
- for Unix regular expression syntax man pages, 74
- US Census Bureau, 37
- WMS (web mapping service)
 - support for provided by libraries, 1
- WRAP keyword
 - using to cause label text to wrap to a new line, 70
- WRAP LABEL object
 - for specifying a character that will cause a label to wrap, 327

Z

- zlib
 - building and installing, 6–7
 - data-compression library used by GD, 3
- zlib website
 - for downloads and documentation, 14
- ZOOM CGI variable
 - containing the zoom factor to apply to the new map extent, 357
- ZOOMDIR CGI variable
 - containing the zoom direction, 357
- zoomPoint() method
 - code for calculating the zoom factor to pass to, 179
- ZOOMSIZE CGI variable
 - containing the zoom factor, which is always a positive number, 357

JOIN THE APRESS FORUMS AND BE PART OF OUR COMMUNITY. You'll find discussions that cover topics of interest to IT professionals, programmers, and enthusiasts just like you. If you post a query to one of our forums, you can expect that some of the best minds in the business—especially Apress authors, who all write with *The Expert's Voice*™—will chime in to help you. Why not aim to become one of our most valuable participants (MVPs) and win cool stuff? Here's a sampling of what you'll find:

DATABASES

Data drives everything.

Share information, exchange ideas, and discuss any database programming or administration issues.

PROGRAMMING/BUSINESS

Unfortunately, it is.

Talk about the Apress line of books that cover software methodology, best practices, and how programmers interact with the "suits."

INTERNET TECHNOLOGIES AND NETWORKING

Try living without plumbing (and eventually IPv6).

Talk about networking topics including protocols, design, administration, wireless, wired, storage, backup, certifications, trends, and new technologies.

WEB DEVELOPMENT/DESIGN

Ugly doesn't cut it anymore, and CGI is absurd.

Help is in sight for your site. Find design solutions for your projects and get ideas for building an interactive Web site.

JAVA

We've come a long way from the old Oak tree.

Hang out and discuss Java in whatever flavor you choose: J2SE, J2EE, J2ME, Jakarta, and so on.

SECURITY

Lots of bad guys out there—the good guys need help.

Discuss computer and network security issues here. Just don't let anyone else know the answers!

MAC OS X

All about the Zen of OS X.

OS X is both the present and the future for Mac apps. Make suggestions, offer up ideas, or boast about your new hardware.

TECHNOLOGY IN ACTION

Cool things. Fun things.

It's after hours. It's time to play. Whether you're into LEGO® MINDSTORMS™ or turning an old PC into a DVR, this is where technology turns into fun.

OPEN SOURCE

Source code is good; understanding (open) source is better.

Discuss open source technologies and related topics such as PHP, MySQL, Linux, Perl, Apache, Python, and more.

WINDOWS

No defenestration here.

Ask questions about all aspects of Windows programming, get help on Microsoft technologies covered in Apress books, or provide feedback on any Apress Windows book.

HOW TO PARTICIPATE:

Go to the Apress Forums site at <http://forums.apress.com/>.

Click the New User link.