The Pragmatic Programmers

Desktop GIS

Mapping the Planet with Open Source Tools

Gary E. Sherman

What readers are saying about *Desktop GIS*

Desktop GIS is a comprehensive survey of open source software for GIS users. Everyone from casual mapmakers to seasoned professionals will find a wealth of information from data visualization to advanced spatial analysis techniques. This book is an ideal text for anyone interested in a hands-on approach to learning the latest in open source GIS technology.

► **Matthew Perry**
Senior Staff Scientist, Geosyntec Consultants

Desktop GIS

Mapping the Planet with Open Source Tools

Gary E. Sherman

The Pragmatic Bookshelf
Raleigh, North Carolina Dallas, Texas

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Copyright © 2008 Gary E. Sherman.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in China.

ISBN-10: 1-934356-06-9

ISBN-13: 978-1-934356-06-7

Contents

Preface **9**

How to Use This Book 9
Acknowledgments 10

1 Introduction **12**

1.1 What Is Desktop Mapping? 13
1.2 Desktop vs. Server Mapping 20
1.3 Assembling a Toolkit 21
1.4 Other Mapping Options 22
1.5 What's Ahead? 22

2 Getting Started **23**

2.1 The Three User Classes 23
2.2 Which Are You? 24
2.3 Choosing a Platform 25
2.4 Selecting the Right Toolkit 26
2.5 Acquiring and Installing Software 27
2.6 Integration of Tools 29
2.7 Managing Software Change 30
2.8 Getting Support 31
2.9 Where to Find Data 34
2.10 Next Step 36

3 Working with Vector Data **37**

3.1 Viewing Data 37
3.2 Rendering a Story 42
3.3 Looking at Attribute Data 44
3.4 Advanced Viewing and Rendering 45
3.5 Making Attribute Data Work for You 56

4 Working with Raster Data **67**
4.1 Viewing Raster Data 67
4.2 Improving Rendering with Pyramids 73
4.3 Intelligent Rasters 76

5 Digitizing and Editing Vector Data **81**
5.1 Simple Digitizing 81
5.2 Editing Attribute Data 89
5.3 More Digitizing and Editing 90

6 Data Formats **91**
6.1 Common Formats 91
6.2 Choosing a Standard Format 93
6.3 Conversion Options 96

7 Spatial Databases **98**
7.1 Introduction 98
7.2 Open Source Spatial Databases 99
7.3 Getting Started with PostGIS 101
7.4 Using PostGIS and Quantum GIS 110
7.5 Using PostGIS and uDig 118
7.6 Summing It Up 119

8 Creating Data **120**
8.1 Digitizing 120
8.2 Importing Data 122
8.3 Converting Data 128
8.4 Using GPS Data with QGIS 130
8.5 Georeferencing an Image 135

9 Projections and Coordinate Systems **138**
9.1 Projection Flavors 139
9.2 Working with Projections 140
9.3 The PROJ.4 Projections Library 145
9.4 More Resources 148

10 Geoprocessing **149**
10.1 Projecting Data 150
10.2 Line-of-Sight Analysis 153
10.3 Hydrologic Modeling 156
10.4 Creating Hillshades 159
10.5 Merging Digital Elevation Models 164
10.6 Clipping Features 166

11 Using Command-Line Tools **174**
11.1 GMT . . . 174
11.2 Using GDAL and OGR . . . 186
11.3 Creating a Spatial Index for Shapefiles . . . 201
11.4 PostGIS . . . 203

12 Getting the Most Out of QGIS and GRASS Integration **208**
12.1 Loading and Viewing Data . . . 209
12.2 Editing GRASS Data with QGIS . . . 211
12.3 Using Analysis and Conversion Tools . . . 218
12.4 Summing It Up . . . 233

13 GIS Scripting **235**
13.1 GRASS . . . 235
13.2 QGIS . . . 236
13.3 GDAL and OGR . . . 248
13.4 PostGIS . . . 255

14 Writing Your Own GIS Applications **263**
14.1 Options for Writing Your Application . . . 263
14.2 Examples of Custom Applications . . . 265
14.3 How to Approach Your Own Project . . . 267

A Survey of Desktop Mapping Software **269**
A.1 GUI Applications . . . 270
A.2 Command-Line Applications . . . 283
A.3 Other Tools . . . 289

B Installing Software **290**
B.1 GRASS . . . 290
B.2 OpenJUMP . . . 292
B.3 Quantum GIS . . . 292
B.4 uDig . . . 293
B.5 GMT . . . 293
B.6 GDAL/OGR . . . 295
B.7 FWTools . . . 295

C GRASS Basics **296**
C.1 Location, Location, Location . . . 296
C.2 Getting Some Data . . . 306
C.3 Working with Data . . . 315
C.4 Getting to Know the GUI . . . 319
C.5 Digitizing and Editing . . . 322

D Quantum GIS Basics **330**
D.1 Vector Properties and Symbology Options 330
D.2 Project Properties 336
D.3 Map Navigation and Bookmarks 336
D.4 Plugins 339

Index **343**

Preface

Open source GIS is a rich and rapidly expanding field of endeavor. Take a look at the FreeGIS Project website,[1] and you'll see an impressive list of more than 300 applications. With such a wide array of software available, it's impossible for any one book to cover everything. In *Desktop GIS*, the goal is to introduce you to some of the major open source GIS applications that are in active development today. It's a tough proposition to cover each of these to the extent they deserve. Instead, the approach is to introduce you to tools that will get you started with open source GIS and enable you to reach out and expand on your own.

You might think this book is a beginner's book. Although it's true that it starts out that way, we move quickly into areas that intermediate and advanced users can profit from. Starting from a simple problem and moving through the concepts of using open source, we'll advance to examples of real GIS analysis.

How to Use This Book

If you are new to the concept of GIS, begin at the beginning. For those of you familiar with GIS but new to open source, the introduction is worth reading, but you should definitely take a look at Chapter 2, *Getting Started*, on page 23 for an overview of things to consider.

If you want an overview of what's available in open source GIS, before you proceed take a look at Appendix A, on page 269.

Following the introductory chapters, we delve into working with data, digitizing and creating new data, and then doing analysis using open source GIS applications such as GRASS, QGIS, and uDig. In later chapters, you will find information on scripting and writing your own applications.

1. http://freegis.org

Since this book is not a tutorial, we won't go into all the nuances of each application mentioned. We will show you how to accomplish common tasks using the software, and in those cases you'll find a fair bit of guidance.

The appendixes contain information on installing and using some of the applications mentioned in the book. If you need further assistance getting started, refer to websites for the respective projects where you'll find a wealth of information.

Versions

The dynamic nature of the open source GIS community was readily apparent during the writing of this book with several projects releasing major versions. Fortunately, the differences between the versions don't significantly impact our illustrations and examples. Where there is a difference, it is noted in the text. For software used in the examples, the following versions were used:

GRASS
: For most of the examples, version 6.2.*x* was used. Where the version 6.3 release candidate was used, it is noted in the text.

Quantum GIS
: Most of the examples use version 0.8.1. In later chapters where the Python bindings are discussed, version 0.9.*x* is used. There are some minor differences in the user interface between 0.8.1 and 0.9.*x*, but you should be able to use the later version without much difficulty.

uDig
: For the uDig examples, you can use either the stable (1.0.6) version or the current version 1.1 release candidate.

For GDAL, GMT, PROJ.4, and PostGIS, you can use the latest versions to work through the examples in the book.

Acknowledgments

I want to express my thanks to those who have reviewed all or parts of the book and provided input and encouragement: Markus Neteler, Matthew Perry, Barry Rowlingson, Tyler Mitchell, Frank Warmerdam, Aaron Racicot, Jason Jorgenson, Brent Wood, Dylan Beaudette, Roger Pearson, Martin Dobias, Patti Giuseppe, and Landon Blake.

My family put up with me being "present yet absent" for months on end. I want to thank them for their support, encouragement, and patience during the entire process.

Lastly, I want to dedicate this book to the memory of my father, who passed away during its development. While from another era, he instilled in me the curiosity of how things work and what to do when they don't. He taught me much, and for that I am forever grateful.

Gary Sherman
March 2008
gsherman@mrcc.com

Where there is no vision, the people perish. . .

► Proverbs

Chapter 1

Introduction

Interest in mapping is on the rise, as witnessed by services such as Google Earth, Virtual Earth, MapQuest, and any number of other web mapping mashups. These are all exciting developments, yet there is another realm you should consider—the world of desktop mapping with open source GIS (OSGIS). You may be thinking "Why do I need OSGIS? I have all the web mapping sites and tools I could ever need."

To answer that question, let's consider our friend Harrison. He's coming from the same place as many of us, having played around with the web mapping tools and is now ready to start adding his own data. Harrison quickly discovers he can't add the GPS tracks from his last hike to any of the "conventional" web maps—all he can do is view the data they provide. Next he fires up Google Earth[1] to see whether that will do the trick. He soon finds that with a little digging around on the Internet, he is able to get the tracks off his GPS and import them into Google Earth.[2] With a bit of work, Harrison is now able to display his GPS tracks.

Fresh from his victory in Google Earth, Harrison now embarks on his next project, which is the real reason he is interested in mapping. It turns out that Harrison is an avid bird watcher. Not only did he record his trek, but he also logged waypoints at each bird sighting. With each waypoint, he made a few notes about the species of bird, the number of birds observed, and the weather conditions. Harrison has just moved from simply displaying where he walked to wanting to display his bird sightings and "analyze" his observations. In doing so, he has hit upon the basis of a Geographic Information System (GIS)—linking geographic locations to information.

---

1. Although there is a free version of Google Earth, it is not open source.
2. If Harrison had Google Earth Pro, he could have directly loaded his GPS data. But he opted for the adventurous (in other words, free) route.

Harrison ponders his next move—how to get all that good bird information that's on his trail-weary notebook sheets into a form where he can not only visualize it but even ask some questions (in other words, do analysis). Harrison wants to be able to do the following:

- View the locations where he observed birds
- View only the locations where he saw the yellow-bellied Wonky Finch
- Scale his locations (dots) based on the number of birds seen at each location (more birds = bigger dot)
- See whether there is any relationship to the weather and the number or types of birds he observed

Harrison needs not only a good visualization tool but something he can do analysis with. Harrison needs some GIS tools, and of course we offer up open source desktop GIS as the solution to his mapping needs.

1.1 What Is Desktop Mapping?

Harrison has introduced us to a problem that we can solve with desktop GIS software. So, what exactly is *desktop mapping*? Well, it isn't about drawing a map to find your pencils, pens, stapler, and coffee cup. Desktop mapping is all about using software installed on your computer to visualize and analyze data. Not only can it be used to meet Harrison's bird-mapping needs, you can also create hard-copy maps, create data out of thin air (well almost), and examine the relationships between features.

Although it's true you can do all this with proprietary software, we'll take a journey through the open source GIS landscape to see what we can find. To get started, let's take a look at the kinds of things we can do with open source desktop mapping tools. I've already told you that Harrison's bird project can be handled quite nicely. Everybody likes to "get on the board" quickly rather than learning a bunch of theory and commands. We'll try to do the same here as you start your journey into OSGIS, whether you are a beginner or a battle-scarred GIS geek.

To give you an idea of what we can do, you can see a rather simplistic interpretation of the progression of things we can do with GIS (open source or not) in Figure 1.1, on the following page, in order of increasing complexity. We'll take a closer look at each of these functions to help you get an idea of what's involved with each. In turn, you can decide

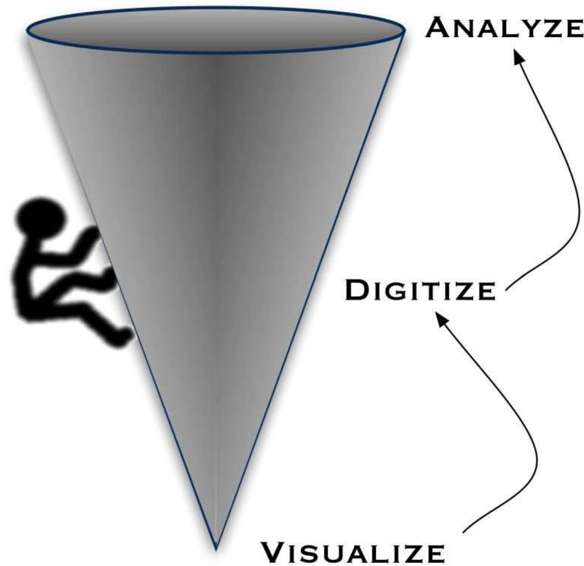


Figure 1.1: GIS functions

how far up you want to climb. You'll notice that our GIS progression is like scaling the outside of an inverted cone. Imagine yourself as a rock climber doing a free climb up the outside of that cone. The higher you go, the more of a workout you're going to get. Learning OSGIS is a bit like climbing that cone. Fortunately, you decide how far to go based on what you want to do. Getting on board is pretty easy. Let's visualize.

Visualize

The dictionary (well, one of them anyway) defines visualize as "make (something) visible to the eye." That definition fits pretty well with what we want to do. We want to *see* our data. This is the entry-level activity in GIS. We get some data, whether from our GPS or by downloading it from the Internet, and we look at it. Remember, that's the first thing Harrison was interested in—looking at his data. That sounds good, but you'll quickly find that just looking at a bunch of black lines on a white background isn't all that exciting. We need a context for our data. Let's return to Harrison for a moment.

Harrison has caught up with us and is staring at a nice collection of seemingly random lines on a snow white background in his desktop GIS viewer.[3] Although immensely proud of his accomplishment, it really isn't much to look at and certainly not very enlightening. Harrison wishes he could display his data over the same topographic (topo) map he took with him on his hike. Using his favorite search engine, he begins the hunt for a topo map. Fortunately for Harrison, he stumbles upon the Libre Map Project[4] that has free topo (DRG[5]) maps for the United States. Harrison quickly finds his part of the world and downloads the appropriate maps. Now he can overlay his GPS data on a background that provides some context.[6] Harrison gets really ambitious now and goes to hunt for some imagery to add to his map. We'll check on him later.

What's the first thing you are going to do when you add your bird locations, fishing holes, or Big Foot sightings to your map? My guess is you'll want to change the color, symbol, size, and any number of other things to control the way it looks. This is another important aspect of visualization—being able to change the way the data looks. We call this *symbolizing* your data. I think it's safe to say that all OSGIS viewers provide this ability. Typically you can change the colors, fill patterns, line styles, and marker symbols to get the effect you want.

Think back a moment to Harrison's requirements for his bird-mapping project. He wants to not only view the locations where he saw the birds but also to change the size of the dot based on how many birds he saw at a location, as in Figure 1.2, on the next page. He also may want to display only a single species. Most OSGIS viewers can easily accomplish these tasks—and more. Harrison hasn't thought of it yet, but he's going to want to symbolize his bird locations by species as well. By using both colors and sizes, he can convey a lot of information about his birding observations. We'll see examples of how to render our own data using these techniques in Section 3.2, *Rendering a Story*, on page 42.

Now that we have explored visualization a bit, let's move on to the next step. From visualize to digitize.

---

3. At the moment, we're talking in generalities; we'll get to some specific OSGIS applications shortly.
4. http://libremap.org
5. A DRG is a scanned USGS topographic map, typically available in TIFF format.
6. The astute observer is asking, what about the projection difference between the GPS data and the DRG? We'll pretend that doesn't exist for the moment.

Figure 1.2: Bird sightings: The bigger the dot, the more birds

Digitize

Let's define what it means to digitize. Breaking out our handy dictionary gives us a definition along the lines of "convert pictures or sound into digital form for processing in a computer." There's nothing mysterious about that definition. See, you may have already done some digitizing when scanning your old photographs or playing with the sound recorder and a microphone on your computer.

When it comes to GIS, digitizing usually means capturing and storing points, lines, or polygons from paper maps. But for the purposes of our general discussion, we'll just equate digitizing with creating data and hope the purists don't catch up with us before we're done.

Harrison has a digitizing project in mind. Looking at the DRG he downloaded to use as a base for his bird visualization project, he finds it shows roads, trails, lakes, contour lines, and other physical features. Unfortunately for Harrison, many of the small lakes on his map are not labeled with their name. In order to make a better-looking display (and

GIS Data Types in a Nutshell

You are about to be exposed to a bunch of new terms as we launch into our discussion of what you can do with OSGIS. Among these are GIS data types. Essentially you can divide GIS data into two types: vector and raster.

Think of vector data as things you would draw with a pencil and paper. We could draw points, lines, and polygons. In GIS, the features have a location in the real world, allowing us to examine their relationship to other features.

Taking it one step further, we can attach *attributes*—information about the feature. Our vector data can have one or more attributes. For example, we might create a polygon that represents the outline of a lake. The attributes for the lake might be name, area, perimeter, and mean depth. Attributes are stored in fields in our dataset, whether they be in a file or a database table.

These two characteristics, location and attributes, are what make GIS different from a simple drawing or paint program.

The other type of GIS data is raster data. In a raster, the information is represented by cells (in some cases, pixels) where the value of each cell represents a quantity or color. Examples are a photograph where the cells represent a color and an image where each cell value represents an elevation.

In GIS, we use both types of data, depending on what we are trying to accomplish. In the simplest case, we might use a raster image—an aerial photograph in this case—showing our neighborhood. We then would overlay our vector data in the form of streets. It not only makes a nice picture to look at, but with attributes attached to the streets, we can also learn the name of each.

ultimately a hard-copy map), Harrison would like to label the lakes. He could just use a paint program and label the lakes, but then he would have to modify the original image. Besides, Harrison tends to flip-flop a bit about what he wants, so maximum flexibility is important. Harrison then discovers the notion of creating his own *vector* data. If you're not familiar with it, vector data is just points, lines, and polygons that represent real features on the ground. Harrison thinks about creating a point near the middle of each lake and labeling it, but that would look a bit goofy, even for him. He then decides to digitize each lake and make a polygon. For each polygon, he'll add an attribute—the name. While Harrison is busy working on his lakes, let's talk briefly about the process of digitizing.

In the simplest sense, digitizing is tracing features with your mouse. In reality, there is a fair bit of skill involved in doing it right. The process goes something like this: you create a new *layer* (think file for now) to store your features in, add some attributes to it (for example, the lake name), and then begin tracing features. As you complete each feature, you enter the attributes. When you are done, you have a layer you can view and label using the attributes you entered. Of course, this is a simple explanation, but all digitizing is really an extension of these concepts.

We've kind of lumped things together under the digitizing category. There are other ways to create new data apart from digitizing. Harrison actually illustrated this for us when he imported his GPS data. Other ways to create data include importing from a text file, scanning images, and even accepting coordinates from a web form. We'll get into more of this later. Let's hope by now both you and Harrison have a good idea of just what it means to create GIS data. Once we have all this good data, it's time to analyze.

Analyze

This is where GIS really shines. Being able to use our data we worked so hard on collecting to answer some what-if questions is what makes GIS exciting. This is also what separates GIS software from being just a "viewer."

Using GIS we can answer all types of questions. Let's get Harrison to help us out with an example. He has a theory that most of his bird sightings are within 200 feet of a lakeshore. With all his hard work, Harrison can view both his bird sightings and the lakes, but he can't

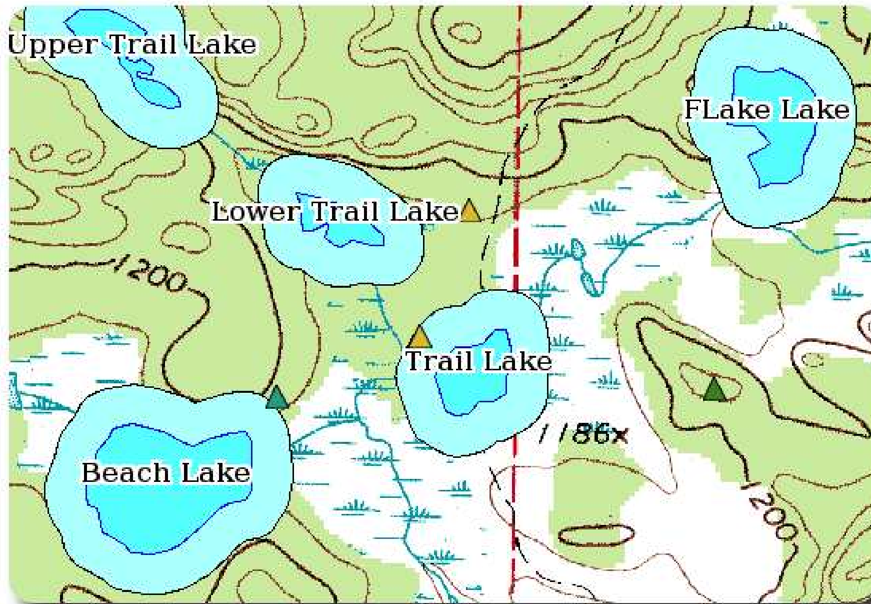


Figure 1.3: A 200-foot buffer around the lakes

really tell how far apart they are. He could use the fine tools provided by his software to measure the distance from each sighting to the nearest lake or lakes. But this is time-consuming and tedious, and the end result can't be visualized. Fortunately, Harrison can use a common GIS operation known as *buffering*.

Harrison proceeds to create a 200-foot buffer around his lakes (see Figure 1.3). This is pretty much a one-stop operation. You indicate what layer you want to buffer (lakes) and enter the distance. The software then calculates the buffer around each lake and creates a new layer containing the result. Harrison now proceeds to set up his display. He adds the new buffer layer to the map, then the lakes, and finally the bird sightings. Any bird sighting falling on the buffer layer is within 200 feet of a lake (or lakes). Harrison can quickly visualize his results and see whether his theory is right. OK, so it turns out he was wrong. It looks like the bird sightings don't necessarily fall within 200 feet of a lake. Harrison decides he can still be right and goes off to create a 500-foot buffer.

This is a simple example of the type of analysis you can do with open source GIS applications available today. You may be thinking that Harrison's analysis is a bit contrived and really not all that significant—and you are probably right. Let's list a few more situations where a buffer analysis might provide meaningful insight:

- Restrict development to a distance at least 500 meters from an active eagle nest.
- Determine where to allow a drinking establishment such that it's at least a quarter mile from any school.
- Develop emergency action plans by identifying all public facilities within a given distance of a hazardous storage site.
- Establish a setback around a creek or stream.

And the list goes on. As you can see, the simple operation of creating a buffer can answer a lot of questions. It's a valuable tool and just one of many that we'll take a look at as we get deeper into specific applications. Of course, there are a lot of other types of analysis we can do with desktop GIS. We'll explore some of these later.

We've now taken a look at three aspects of GIS: visualization, digitizing, and analysis. With that under our belt, we are ready to get into some more specifics. Oh, and about Harrison—he finally proved his point by creating a 5,000-foot buffer around all the lakes. As usual, the tools alone can't provide a meaningful analysis. Before we move on too far, let's take a quick look at the server side of things.

1.2 Desktop vs. Server Mapping

When you think of a server, you probably think of a big machine locked away in an air-conditioned room somewhere. Well, that could be true, but in this case I'm referring to software, not hardware.

The server side of open source GIS provides important capabilities for us on the desktop. For example, we might have *spatial database* that stores our data. Or we might have a spatial server that can pump out data using a number of web standards. We can use all these data sources from the desktop.

We might have resources on a bunch of servers, all accessible from our desktop GIS applications and serving up all the data we need. We're still doing desktop mapping, using the tools installed on our local machine. Let's contrast that for a moment with server-side mapping.

You can see an example of server-side mapping by pointing your web browser at one of the many web mapping applications on the Internet. These range from sites providing maps and driving directions to those serving up massive quantities of data. I'm sure everyone has seen examples of the type of applications I'm talking about, but for Harrison's benefit we'll mention a URL. Take a look at the Geodata.gov website[7] for some sample web mapping applications.

When using server-side mapping, we don't install anything on our local machine, and all we need is a web browser. The good thing about this scenario is that someone else has done all the work in assembling the data and preparing it for display and use. Why would we want to go the desktop route instead of letting someone else do all the work for us? Our friend Harrison discovered some of the reasons in his bird-mapping project. He wanted to view his data, not the data provided by some server somewhere. He wanted to create new data by digitizing the lakes.[8] Harrison also wanted to analyze his data by buffering and storing the results. A lot of these operations can be done with server-side mapping, but the data ends up living on the server. If you're lucky, there may be a way to export it and make it yours.

Am I down on server-side mapping? No—it's an excellent way to *visualize* data and provide it to the masses. In fact, there are projects underway to further enable the server side and extend the capabilities to analysis as well.[9] Using a mix of desktop and server GIS software is a good mode of operations, especially if you are like Harrison and want to be both a data creator and a consumer.

1.3 Assembling a Toolkit

With the preliminaries out of the way, my goal is to help you assemble a loosely coupled toolkit of OSGIS applications. There are good reasons to assemble a toolkit rather than using a single mapping application. Just as you wouldn't use a screwdriver to build an entire house, we'll get better results if we use the right tool for the job at hand. When it comes to OSGIS, I'm a strong proponent of IIWUI—"If It Works Use It."

7. http://geodata.gov
8. Sure, you can create features on some web mapping sites. But where do they reside when it's all said and done? On the server.
9. An example is PyWPS (http://pywps.wald.intevation.org), which allows web access to GRASS GIS.

Not everyone will need or want the same tools in their toolkit. One of the things we hope to accomplish on our journey together is to identify which tools you need and then learn how to assemble them into a system that works for you. Ideally, you should come through the experience with some nicely integrated applications and utilities to serve all your mapping needs on the desktop. As you assemble your toolkit, you'll find that many applications are of the "Swiss Army knife" variety, providing a wide range of capabilities.

1.4 Other Mapping Options

What are your other options? Well, we already mentioned them—web-based applications. Unless you are developing your own web mapping application, you're pretty much at the mercy of the web developer. You must use their interface and work with the layers they provide. For some folks, this is a perfectly good solution, and it's definitely something to consider when you are ready to share your hard work with the rest of the world.

For those of you who need to work with local or distributed datasets to create, edit, and display data, this isn't going to work. You will need tools to create and maintain your data.

A solution that falls in the middle is Google Earth, now available on Mac OS X, Linux, and Windows. With Google Earth you can add and display your own vector data, once you've converted it to the proper format. I find that using my desktop GIS toolkit to create and prep data for Google Earth meets my IIWUI test.

1.5 What's Ahead?

To give you an idea of where we're headed, we're next going to dive into OSGIS and look at the whole notion of using open source for your mapping needs. From that point on, we'll look deeper into concepts, data, and use of the tools at our disposal. Our goal is to get you up to speed on working with OSGIS desktop applications, and there is a lot of ground to cover. Unfortunately, we can't give you an in-depth tutorial for all the applications we'll use. In the appendixes you'll find some additional information for some of them, and we'll point you to additional resources as we go along.

Imagination is more important than knowledge.

► Albert Einstein

Chapter 2

Getting Started

Before you start madly downloading software to assemble your GIS toolkit, let's think a bit about your requirements, including what type of mapping you are interested in doing. You may not know the answer to that question. Most likely if you are starting out, you'll follow the same path as Harrison—moving from visualization to creating your own data to doing analysis. Ultimately, your needs, goals, and requirements will guide you in assembling your toolkit. For example, there is no point in assembling an industrial-strength system to simply view GPS tracks on a map.

As you explore your needs, remember to keep open the possibility for expansion. As you begin your journey into OSGIS, you may end up at a destination you never considered. The good thing is, you can always "upgrade" your toolkit.

2.1 The Three User Classes

If you are already a GIS user, you likely have a good idea of your needs and requirements, but it's always good to reevaluate. Let's consider three classes of GIS users to help you get started. To help us get acquainted, we'll use the names Clive, Irving, and Alyssa.

The Casual User

Clive is a casual user, and what he likes to do is visualize mapping data. His toolkit contains one or more GIS viewer applications and maybe a custom *data store*—a place where data resides—such as a spatial database. In the simplest case, Clive stores his data in files (shapefiles, Tagged Image File Format [TIFF], and so on). He doesn't need big fancy

GIS algorithms to make him happy. Clive may on occasion need to create data by importing GPS tracks or maybe even digitizing some lakes or trails.

Since he doesn't create a lot of data, Clive gets it by downloading from the Internet and sometimes from his GPS, just like Harrison did in the first chapter. The other things Clive uses his GIS software for are printing simple maps and doing some visual analysis by plopping layers on top of each other.

The Intermediate User

Irving is an intermediate user, and he likes to not only visualize but to create data—sometimes lots of it. Irving typically creates data by digitizing and/or converting it from other sources. Sometimes Irving needs to produce cartographic output (a paper map with lots of decorations) to share with his friends and cohorts.

Irving works with a wider range of data formats than Clive. He likes to digitize data from raster maps (just like Harrison), convert data to suit his needs, create subsets of his data to better visualize where things are, and use symbols to help visualize some of the relationships between features.

The Advanced User

Alyssa is an advanced user, and she has mastered the activities and tools used by Clive and Irving. But she has greater needs—Alyssa lives to analyze. Beyond viewing, data creation, and map production, she uses GIS to answer questions based on spatial relationships. She does cell-based analysis and perhaps even routing and geocoding. She also may need to write programs and scripts to accomplish her tasks.

Some of the tasks that Alyssa performs include doing line-of-sight analysis ("Can you see me from here?"), change analysis, buffer analysis, and grid algebra. She needs a high-powered toolkit.

2.2 Which Are You?

What do Casual Clive, Intermediate Irving, and Advanced Alyssa all have in common? They all started at the same place and they each use some of the same tools. You'll also notice that the classes of users bear a strong resemblance to the functions in our GIS Cone in Figure 1.1, on page 14.

Based on our characters, you should be able to determine where you fall in the lineup. Not only should you consider what you are now, but what your needs will be in, say, six months, a year, and beyond. The truth is that each of our users may have the same tools in their toolkit. The difference will be in how they are used and to what extent. As we progress through our exploration of desktop applications and their capabilities, keep your self-assessment in mind. We will provide reminders along the way to indicate which tools are best suited for each class of user.

Lastly, this artificial classification scheme is not hard and fast. It only provides a starting point for you to think about your requirements and help you build up your own open source GIS toolkit.

Although jumping into the deep end of the pool can be an effective, albeit traumatic, way to learn to swim, sometimes it pays to wade in gradually, feeling your way along. The more feature-rich an application is, the more likely it is to have a steeper learning curve. You start with the tools that meet your needs and work your way into the more complex as your appetite for GIS increases.

Determining what kind of user we are wasn't too bad. Now we move into something a bit more difficult and look at some of the challenges in assembling an open source GIS toolkit. Everything you do (including crossing the street) entails some level of risk. Whether you use open source or proprietary (closed source) software, you incur some risk. The rest of this chapter looks at the challenges and risks and provides some insight on dealing with potential pitfalls.

2.3 Choosing a Platform

In ancient times (around twenty to thirty years ago), if you wanted to "do" GIS, you had to buy a certain type of hardware running a specific operating system. As time went on, the choices increased. Today you can pretty much find GIS software to run on your favorite system, assuming it's Linux, *nix, OS X, or Windows. You still can't find much in the way of GIS software for your Commodore 64.

The logical assumption might be that we just get the software for our current platform and forge ahead. But consider this: should you choose the software for the platform or the platform for the software?

There are a number of factors to consider:

- Your comfort level with various operating systems
- The types of applications you need
- Your budget

Typically you will choose the software for your current platform and be on your way. For those of you who are comfortable in two or more operating systems (say Linux, Mac OS X, or Windows), your options are more varied. I would rank my choices pretty much in that order. If you have a choice, Linux or OS X may be a better fit for you. If not, we plan to show you Windows users plenty of options in the coming chapters.

To get the most benefit as an advanced (and to some extent an intermediate) user, you should probably consider Linux or a Unix variant. As your demands increase, you require software that is more readily available on those platforms.

Budget figures in somewhat, with the hardware for some platforms costing more than others. Since your software acquisition costs are going to be low or nonexistent, you can afford to spend a little more on hardware.

2.4 Selecting the Right Toolkit

We mentioned this earlier, but it's worth repeating. Pick the applications for your toolkit based on what you want to do. There is no point in installing every GIS application out there to view Grandma's house and the local latte stand. Those are valid uses, but why make it hard on yourself? On the other hand, you should think ahead a bit and keep your options open. That way we won't end up installing a simple viewer and expect to do volume or fill analysis. To give you a head start, we'll be looking at the applications shown in Figure 2.1, on the next page, which also shows the appropriate user classes. This is really a generalization, but it does give you an idea of the level of experience appropriate for each application. In reality, many of the applications can be used across the spectrum of user classes.

To help you learn more about the software choices available, you can refer to the survey of open source desktop GIS applications and the capabilities of each in Appendix A, on page 269.

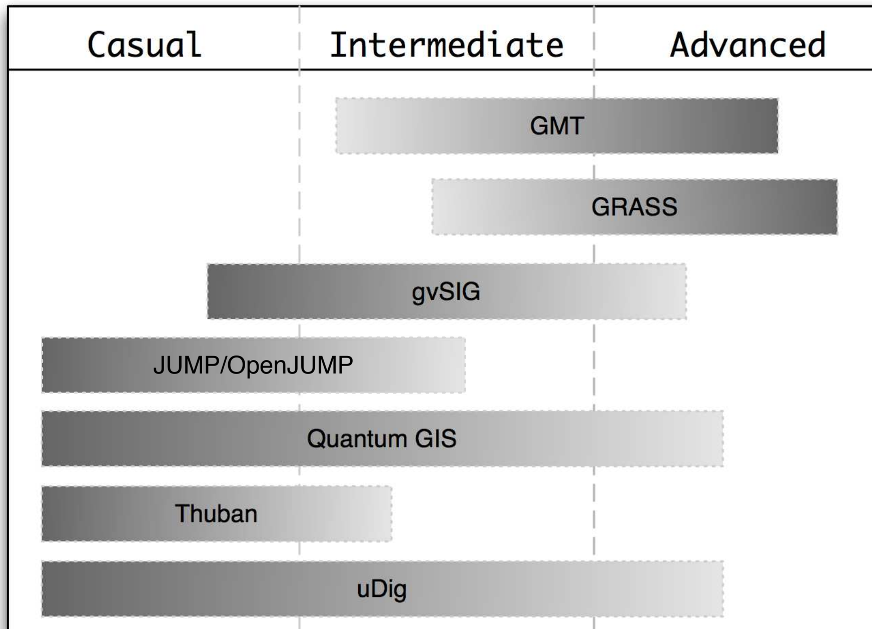


Figure 2.1: Some OSGIS applications in relation to class of user

2.5 Acquiring and Installing Software

Getting an OSGIS package can be a bit different than buying something off the shelf. Although it's true you can purchase Linux distributions off the shelf that include OSGIS software, typically you are going to be downloading a binary package for your platform. If you're in luck, that's the situation you'll encounter as you begin to assemble your toolkit. This route allows you start using the software without worrying about all those nasty things like dependencies, compilers, and libraries. The worst case is you may have to download the source code and compile it yourself.

All the OSGIS software we'll see in this book can be obtained as a binary package or installer, depending on your operating system. This is the easiest way to get started using an application.

Packages, Installers, and Disk Images

Depending on your platform of choice, you may be installing RPMs, DEBs, or .tgz on Linux; .zip or bundled installers on Windows; and disk images (.dmg) or OS X installers on Mac. Most open source GIS projects provide these binary images, and of course it's up to you to determine which to install. As we go along, we'll give you hints on the installation process and mention all three platforms.

Here are some things to be aware of when going the binary route:

- Some packages and/or installers are not provided or maintained by the open source project but by third parties.
- Depending on your operating system, the latest version may not be available.
- The availability of packages for your platform may lag behind the general release of a new version.

Going with a binary package or installer is definitely the way to go when test-driving an application for the first time. This gives you a chance to easily try things without the hassle of gathering dependencies and compiling from source.

In some cases, you have to compile from source because you have no alternative. Here are some reasons why you might want to compile an OSGIS application:

- The binary isn't available for your platform.
- You want the latest and greatest features, but they haven't been released yet.
- You want to *customize* your toolkit.

Compiling a suite of tools from source can be a daunting task for the average user, even for the advanced GIS user. When first starting out, you should consider using binary packages for your platform. This keeps you from becoming frustrated with the process of boot strapping a system from scratch. Once you gain familiarity with the tools and how they interact, you'll be ready to venture into compiling your own system. For now, let's start with the packaged binaries and learn how to use the software rather than get frustrated with the build process out of the gate.

Trying Open Source GIS with a LiveCD

Another option for giving OSGIS a spin is one of the many LiveCDs. These allow you to boot your computer from CD into a Linux system that is preloaded with applications you can use without having to install anything.

You can choose from a number of GIS LiveCDs, but you need to make sure your choice contains fairly recent versions of software. For a LiveCD that attempts to provide the latest versions, check out the Ominiverdi offering.*

*. http://livecd.ominiverdi.org

2.6 Integration of Tools

Rarely will you find one OSGIS application that meets all your needs. In fact, if you do, you're in the minority. An OSGIS toolkit composed of several applications will provide a much more powerful and complete system. Now you're thinking, "Oh, great, I have to learn a whole bunch of new programs to do anything with this stuff." In reality, we'll show you how to get started without a huge learning curve. For those of you already up to speed on GIS and tools, we'll provide that deeper view you're looking for to fill out a complete toolkit.

How Do We Integrate?

The plain fact is that integration is largely up to you. Typically you'll end up with a loosely coupled set of tools, sometimes bound together with scripts or other glue. This shouldn't be interpreted to mean that we are creating a *kludge*, but rather putting our tools neatly in the toolkit and making them play nicely together.

Kludge: A program or system that has been poorly (perhaps sloppily) assembled)

Some tools integrate nicely, and the situation is improving all the time. Consider Quantum GIS (QGIS) and GRASS integration. The GRASS plugin allows you to access a large number of GRASS functions through the QGIS interface.

Another form of integration is using programming language bindings so that you can access the application functionality in Ruby, Python, Perl, and Java programs. We'll talk more about this technique in Chapter 11, *Using Command-Line Tools*, on page 174.

2.7 Managing Software Change

One of the biggest challenges you will face when using OSGIS software is managing change. All systems have an inherent element of risk with regard to change. Computer systems are particularly sensitive to change, meaning if you upgrade one component, you better make sure it doesn't have a negative impact (read: complete meltdown) on the other components. Let's look at an example.

Harrison hears about some really cool features that were just added to SuperMapper. After all, Harrison subscribes to the project's email list and participates on IRC so he can be "in the know." Unfortunately, the new SuperMapper features are in the development version. Undaunted, Harrison proceeds to check out the source code and build the latest, greatest version. And it works great. All the new features are there, and Harrison is one happy mapper—until he goes to run his faithful old workhorse application, MundaneMapper. Turns out that his hacking activities have introduced some library incompatibilities, and now MundaneMapper refuses to start. Harrison has become a victim of BES.

Harrison will glumly tell you that if you want to maintain a stable system, the first thing to avoid is Bleeding-Edge Syndrome (BES). This differs from being an "early adopter." Here is how to tell if you have BES:

- You always download and install the latest beta.
- You find yourself doing CVS and SVN checkouts and building from scratch.
- You subscribe to the CVS/SVN commit mailing lists for several projects and rebuild your toolkit each time a new message comes in.
- You often find yourself with an inoperable system.

Concurrent Versions System (CVS) and Subversion (SVN) are version control systems used when developing software.

Having BES is not so bad if you are a hobbyist or just experimenting and understand the risks. It's not so good if you are trying to do real work and can't afford to break things on a regular basis.

Guidelines for Managing Change

Managing change really refers to keeping your software current, responding to security issues, and keeping things stable so your toolkit can serve you, not the opposite.

Let's look at the three main reasons to upgrade:

- A new version has been released that provides features you need, want, or absolutely can't live without.
- Vulnerabilities in your software.
- A "higher-level" component (like your operating system) requires an upgrade that will render your toolkit applications incompatible.

The first two are a matter of choice; the third may not be if your IT department has any say. If you are lucky, you are master of your own destiny and have control over all aspects of your GIS software, including the operating system. If not, you're going to have to coordinate and cooperate.

Here's a list of some suggestions for managing change in your OSGIS toolkit:

- Proceed with caution. In other words, look before you leap, and make sure you understand all the ramifications of upgrading.
- Identify changes in the latest version of the application(s) that may require extra work on your part.
- Identify changes that remove key functionality you depend on (it sounds strange, but I've seen it happen before).
- Identify dependencies—other packages that will break or things you need to upgrade as part of the process.
- If at all possible, test your upgrades on a nonproduction machine.
- Don't upgrade too quickly after a new release. Monitoring the mailing lists and forums can help identify potential problems that others have already discovered (and oftentimes, solved).

You may be thinking this OSGIS approach is a minefield. In reality, it's no different from managing change with proprietary software. All of the suggestions mentioned here apply equally to both proprietary and open source software, particularly in the GIS realm. Just be smart and never put your data at risk, and you'll be fine.

2.8 Getting Support

Open source software has a unique support system, and OSGIS is no different. Rarely when using a proprietary application can you communicate with the actual developers—with OSGIS you can, often in real time. Most developers are willing to help, assuming you have spent a

bit of time working through things yourself and reading the documentation. Some of the support channels you can use are as follows:

- Mailing lists
- Forums
- IRC (a real-time service that allows you to chat with people across the globe)
- Wikis
- Search engines
- Websites

When using mailing lists for support, you need to be sure to search the archives before posting your question. Quite often the answer to your question will be waiting for you to discover it. In addition to the archives typically maintained by each mailing list, a couple of other independent archives are quite useful: Nabble[1] and Gmane.[2] If the archives don't provide the answer, compose an email to the list, and make sure you include enough information so the group can provide an answer. Keep in mind that most email lists require you to subscribe before you can post a question.

Many people prefer forums for support. Many OSGIS projects have a forum linked to their website. These can be a valuable source of information and are usually searchable. Here you can find users helping users, as well as information from the project members.

Sometimes nothing beats real-time support like you can get on IRC. Many projects maintain a presence on IRC. For example, at any one time on irc.freenode.net you might find the following channels: *#grass*, *#postgis*, *#gdal*, *#mapserver*, and *#qgis*. If you don't know what those projects are, never fear. We cover most of these in our survey of OSGIS applications in Appendix A, on page 269.

IRC has its own unique culture as does each channel. Probably the key thing to remember, apart from doing your homework first, is that people on IRC are almost *always* doing something else at the same time. If you ask a question and nobody answers, it means one of several things. First, nobody is around who knows the answer. People who can't help you often aren't compelled to tell you. Second, the people who do know the answer may be busy at the moment and haven't seen your

1. http://www.nabble.com
2. http://gmane.org

Ask the Right Questions

OK, so you need help, and you're ready to ask for it. Nothing will bring on the silence like a poorly asked question. Remember, the people who know the answer are probably pretty busy and have invested a fair amount of time in collecting the knowledge about the application of interest. You need to do the same. Read the documentation, search the Web, and make your best attempt at discovering the answer yourself. You will learn more from the experience and gain some of that "knowledge."

If you still need help, provide enough information so someone has a reasonable chance of helping you. This typically includes the version of software you are using, your operating system and its version, and exactly what you were trying to do. With most OSGIS applications running on at least three or more platforms, each having its own set of unique issues, this information is pretty important.

Ask the right questions, provide the right information, and you'll get the help you need.

question yet. And lastly, it's possible your question got lost in the rest of the traffic. Just because no one answers doesn't mean they are snobs, arrogant, or hate you. Your best approach is to hang out for a bit on a channel until you figure out the dynamics.

You can also get commercial support for many of the applications discussed in this book. Most OSGIS applications provide information regarding support on their websites. In addition, a list of support providers is available on the Open Source Geospatial Foundation (OSGeo) website.[3]

Although it has been the subject of some heated debates between the closed and open source groups, most people who have needed support for OSGIS are happy with the experience. If you need support, it's out there and readily available.

3. http://www.osgeo.org/search_profile

Free the Data!

There is a lot of data around the world that is either locked up, expensive, or generally unavailable. As we mentioned before, this varies depending on where you are in the world.

There is a movement afoot (actually it's developing on many fronts) to free up data. One example was the "ransoming"* of the U.S. Geological Survey's (USGS) Digital Raster Graphic (DRG) topographic maps. These maps are available online from various sources, some free and some not. To make all the data available in one place free of charge, the maps were purchased and then "held hostage" until contributions equaled the cost. The maps were then given to the Internet Archive to be made available to everyone for free.

Another effort underway in Europe is the Public Geo Data effort.† This effort seeks to liberate publicly collected data and make it available at no charge.

*. http://ransom.redjar.org
†. http://publicgeodata.org

2.9 Where to Find Data

By now you realize (or already knew) that without data, we can't do much with OSGIS. For those of you already deeply entrenched in the GIS world, you pretty much know where to search for data. Feel free to skip ahead. If you are just getting started with GIS, this is a pretty common question. Your desktop GIS toolkit isn't much good without any data to play with.

The availability of free data depends on where you are in the world. If you are lucky, you live in a country that freely provides data collected by the government. If you are not so lucky, you may have to pay, sometimes quite steeply, to get the data. Don't despair—there is a lot of free data available to get you started.

In reality, there are two types of data: base data and "your" data. Base data is just that—you lay it down as a base for the rest of your map. Examples of base data are country boundaries, rivers, towns, and the DRG that Harrison downloaded in our first encounter with him. Your

data is data you have acquired or created for your specific purpose. A simple example is GPS tracks from your latest road trip. You can probably find much of the base data you need for free—let's explore some of the sources of free data.

Clearinghouse Network

One way to find data is to use the Federal Geographic Data Committee's (FGDC) clearinghouse network.[4] The clearinghouse contains nodes (servers) from around the world that contain data and are searchable. Oftentimes you can find the data you need using the clearinghouse search engine.

Geodata.gov

Another source we mentioned previously is geodata.gov.[5] This site was established to be "Your One Stop for Finding and Using Geographic Data." Searching for data on geodata.gov yields a list of results containing links to the metadata or website for each dataset. Some of the data may be available for download. In other cases, you'll find that it's available for viewing only through a web map interface using your web browser.

Other Sources

In the end, the old miner's adage about finding gold applies to geospatial data. Oftentimes the greenhorns would arrive on the gold fields and be clueless. They sought out the sage advice of the old-timers to get them started.

Greenhorn: *Where's the best place to prospect for gold?*

OldTimer: *Gold is where you find it.*

There are a lot of sources for data on the Internet, and a bit of judicious searching can lead to good finds. For additional sources to get you started in your data-prospecting adventure, see the list at desktopgisbook.com.[6]

---

4. http://fgdc.gov/
5. http://geodata.gov
6. http://desktopgisbook.com/data

2.10 Next Step

We've gotten much of the preliminaries out of the way, learned a bit about what OSGIS can do for us, and also looked at some of the things to keep in mind along the way. Now it's time to get into some software and actually do something.

If you want to get the "birds-eye" view of what's available in the open source desktop GIS world, take a look at Appendix A, on page 269.

Now let's get going and view some data.

Chapter 3

Working with Vector Data

In this chapter, we'll start working with vector data (points, lines, polygons) by viewing, editing, and analyzing various datasets. Not only will we view data, but we'll look at tweaking the way data is displayed to make it convey more information at a glance.

3.1 Viewing Data

Viewing data is like the "Hello, World!" application that everyone writes when learning a new programming language. It's the first thing you're going to want to do with any GIS application. Let's start out by seeing what kind of things we can do with vector data using open source GIS software. If you recall Harrison's original project, he first just wanted to view bird locations. We'll take a similar approach and start by viewing some sample vector data.

Viewing data is really more like visualizing the relationships between the features. You can get a lot of information by simply viewing features and applying some special rendering techniques.

When it comes to software, we have a lot of choices for viewing GIS data (see Appendix A, on page 269). As we begin to explore our data, we'll use several different applications to give you a feel for what's available.

Before we can begin, we obviously need some data to work with. If you don't have a shapefile or two handy, you can download[1] a sample dataset and use it to follow along. We will be using this dataset throughout the following chapters when we need to illustrate some basic functions or concepts. The dataset includes world borders, cities, and a nice

1. http://desktopgisbook.com/sample_data

Joe Asks...

What Is a Shapefile?

A *shapefile* stores vector features and their attributes. A given shapefile can contain only one type of feature: points, lines, or polygons.

The term is actually a bit misleading, since a shapefile always consists of at least three separate files. For example, a shapefile named alaska would consist of the following:

- alaska.shp containing the spatial features
- alaska.dbf containing the attributes
- alaska.shx, which is an index file that allows random access to features in the alaska.shp file

In addition to the three main files described here, you might also find alaska.sbx, alaska.sbn, and alaska.qix files. These are additional index files used by some applications. One last file you'll often find associated with a shapefile is a .prj file. This file contains the projection information for the shapefile, including the geodetic datum (for more on datums, see the *Joe Asks...* on page 140).

If you are sharing a shapefile with someone, make sure you include at least the .shp, .dbf, and .shx files; otherwise, it will be unusable.

raster image of the earth (which we'll use in a later chapter when we work with rasters).

Choosing a Viewer

Most of the applications in Appendix A, on page 269, that work with vector data go beyond a viewer. Let's use several of them to look at the sample data. Of course, you don't need to use all of them, but following along will help you decide which is best for you. For help on installing any of the applications, see Appendix B, on page 290.

The truth is that nearly all the OSGIS viewers use a similar user interface. If you can use one, you can figure out the others. Let's start by viewing the world borders data using the User Friendly Desktop Internet GIS, uDig.

Simple Viewing

If you need help installing uDig, take a look at Section B.4, *uDig*, on page 293. OK, let's fire up uDig so we can get a look at that sample data:

- *Linux*: Change to the udig subdirectory, and run udig.
- *Mac OS X*: Double-click the uDig icon in your Applications folder.
- *Windows*: Click the Start button, find the uDig program folder in Program Files, and choose uDig.

When you first start uDig, you are presented with a start-up screen. You can explore the options, but if you are anxious to get busy, click the curved arrow in the upper right of the workspace. This gets us to the business end of uDig.

The uDig workspace isn't much to look at the first time you run it. You'll notice that when uDig starts up, it displays a fairly typical Tip of the Day dialog box. Feel free to click through the tips and see what pearls of wisdom you can find. You can turn off this feature if it bothers you (or you've read them all).

Now let's load the world borders layer to get a feel for how uDig manages layers, as well as the options for symbolizing features. To view the data from our sample dataset, start by clicking the Layer menu and then choosing Add. This opens the Data Sources dialog box, as shown in Figure 3.1, on the next page.

As you can see from the Data Sources dialog box, uDig supports a good selection of formats. Let's start by adding our shapefile of all the countries in the world. Since this is a file-based data store,[2] we choose Files from the Data Sources dialog box and click Next, which opens a file selection dialog box from which we can choose our shapefile. We navigate to the directory containing the shapefile (in this case world_borders.shp) and click Open (or whatever the standard dialog box calls it on your platform). This loads the shapefile into uDig and displays it, as shown in Figure 3.2, on page 41. We've closed the Web Browser tab to maximize the map area and still give you a feel for the entire interface.

If you are feeling adventurous, go ahead and load the cities layer as well, using the same process.

---

2. This is a fancy way to say a GIS data file on your disk drive as opposed to web-accessible or spatial database data.

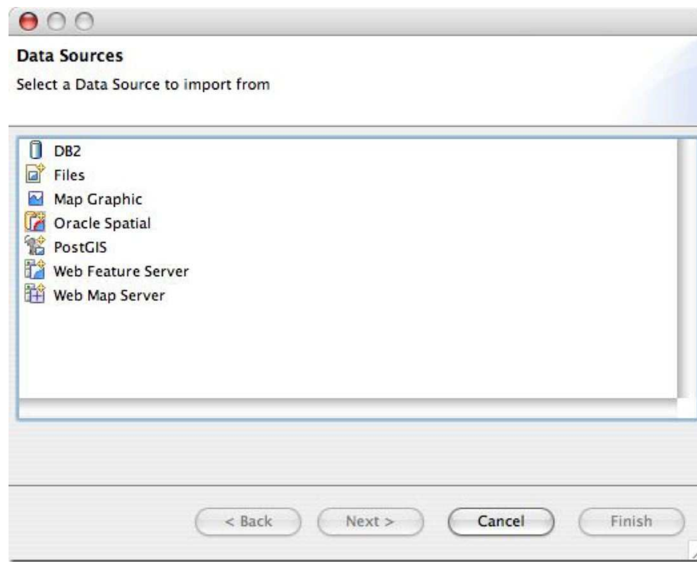


Figure 3.1: uDig Data Sources Dialog

Moving Around

If you're following along, you should be looking at the countries of the world. Take a look at New Zealand—it's pretty small. This is where navigation tools come into play. Every GIS application, whether it be on the Web or the desktop, has a way to navigate around the map. uDig, of course, supports the usual zoom/pan/identify functions common to all applications.

Let's get a closer look at New Zealand. Select the Adjust Current Zoom tool from the toolbar. It's the magnifying glass with the drop-down caret next to it. If you are unsure which tool it is, hover the mouse for a few seconds, and you'll get a tooltip to help you out. Find New Zealand, drag a box around it, and then release the mouse. uDig will zoom the view to cover the region of the box. You now have a better view of New Zealand. You can continue to zoom in as much as you like by dragging boxes with the mouse.

So now that we've zoomed into the gnat's eyebrow, we need to determine how to get back out. There are a couple of ways do it. First we can go back to the full view (extent) by using the Zoom to Layers tool in the

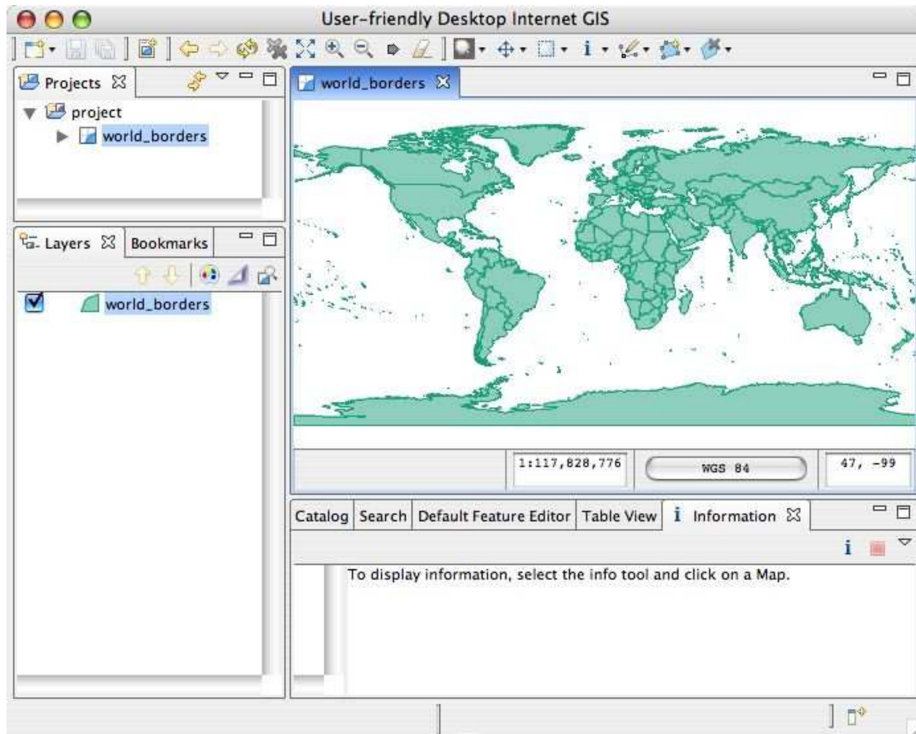


Figure 3.2: uDig displaying world borders

toolbar above the layer list. This will zoom to include all layers on the map. This may not be what we want if we just want to look at the full view of the world_borders layer. To zoom to just its extent, right-click world_borders in the layer list, and choose Zoom to Layer.

We can also zoom out incrementally by using the Zoom Out tool on the main toolbar. Unlike the Adjust Current Zoom tool, this tool is a one-shot affair—you don't interact with the map when using it. With each click, the map is zoomed out by a fixed amount. By now you've probably noticed its cohort, the Zoom In tool. Clicking it zooms the map in by a fixed amount.

One last way we can navigate the map is by panning. To pan the map, select the Pan Map View tool (actually both this and the Adjust Current Zoom buttons are tool groups but contain only one tool) from the main

Joe Asks...

Is It Small or Large Scale?

This can be a constant source of confusion when talking to people about maps, whether they be paper or digital. Let's sort it out now.

A small-scale map covers a large area, whereas a large-scale map covers a smaller area on the ground. The terms *large* and *small* are based on the representative fraction that shows the relationship of one unit on the map to one unit on the ground. A map scale of 1:8,000,000 is smaller than one of 1:24,000 since 1/8,000,000 is a smaller fraction than 1/24,000.

Simple enough. If you ever get confused, just think in terms of fractions, and you'll be able to sort out the small from the large.

toolbar, and drag with the mouse to change the view. You can pan all around the map using this method.

By combining the pan and zoom tools, you can pretty much navigate around the world until your heart's content. You can change the map view in other ways, but we'll leave that for you to discover.

3.2 Rendering a Story

Now it's time to change the way the world looks. This is known as *symbolizing* your data, and you can do it in several ways. Are you happy with the colors uDig chose for the layers? If you are like most people, you have preferences when it comes to these things, and my guess is you're going to want to change the way things look. The simplest, of course, is just a single color for all features, and this is in fact the way all vector layers look when first added to uDig. In uDig you can change the outline color, fill, and marker symbols using the Style Editor. The Style Editor also allows you to turn on labeling and set the maximum and minimum scales at which the layer is displayed.

Our friend Harrison, being the inquisitive sort, quickly decides he wants to be able to tell at a glance where the most populated countries are in

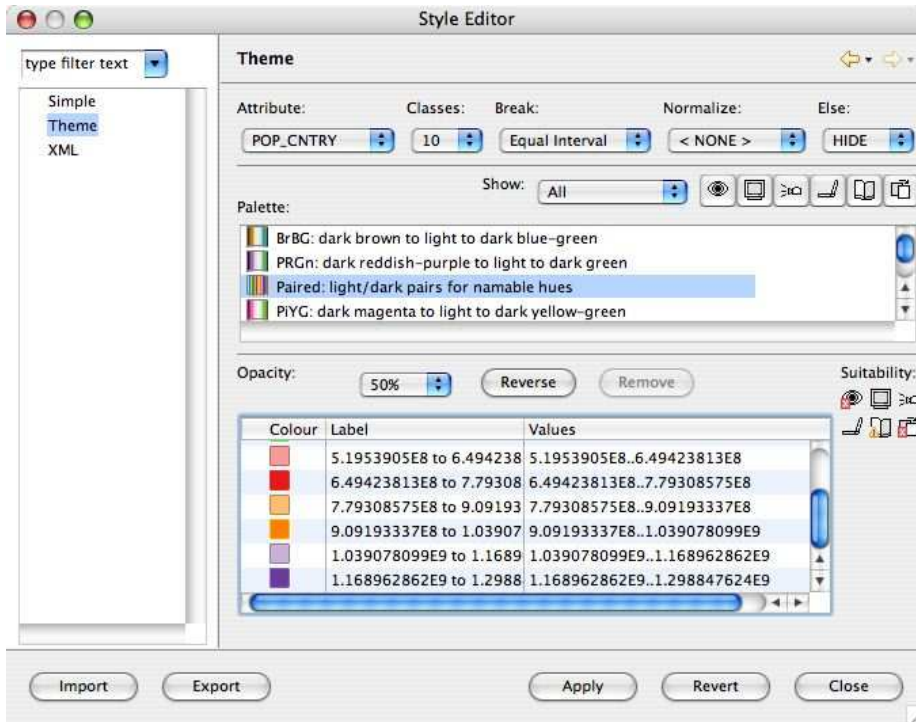


Figure 3.3: Classifying countries by population

the world. Well, we are in luck. Our world_borders layer just happens to have an attribute named POP_CNTRY that contains the population of each country. To make Harrison happy, we can use what's termed a *class break method* to symbolize the data. The Style Editor has a Theme panel to classify the layer based on an attribute. In Figure 3.3, we can see the settings we can use to classify the world boundaries based on population. We set up ten equal interval classes based on the range of population values. The less populated countries will be colored in light blues or greens, while the countries with the largest populations will be rendered in a shade of purple. We could also do a quantile classification, putting roughly the same number of values in each class. But for our purpose, the equal intervals work just fine.

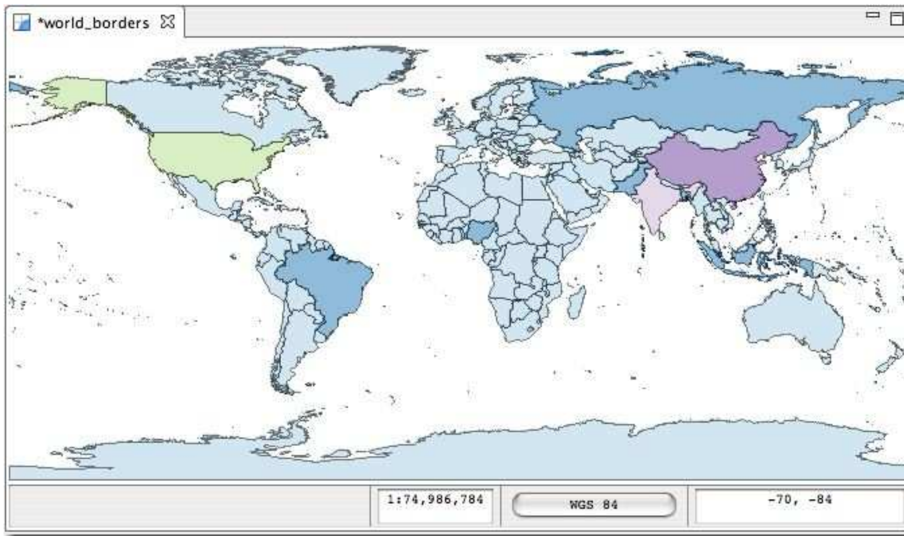


Figure 3.4: Countries classified by population

The results of the classification are shown in Figure 3.4. As you might have suspected, China and India are rendered as the most populous, followed by the United States. We could refine our classification to get a finer-grained view of population by changing the number of classes or the method. This is a common way to render data to make it tell a story. Some of the other OSGIS applications we will look later offer even more ways to symbolize your data.

3.3 Looking at Attribute Data

In the previous section, you might have wondered how we knew about the POP_CNTRY field in the world_borders shapefile. Well, there are a number of ways to examine the attribute data associated with a layer. One of the quickest ways is with ogrinfo, a utility that is part of the GDAL/OGR suite. We'll take a more detailed look at ogrinfo and friends later in Section 11.2, *Using GDAL and OGR*, on page 186. If ogrinfo is not already on your system, see Section B.7, *FWTools*, on page 295 for information on installing FWTools.[3]

3. FWTools is a suite of tools that contains many applications, including ogrinfo.

Then, open a command shell on your system and do the following:

```
$ ogrinfo -so -al world_borders.shp
INFO: Open of `world_borders.shp'
      using driver `ESRI Shapefile' successful.

Layer name: world_borders
Geometry: Polygon
Feature Count: 3784
Extent: (-180.000000, -90.000000) - (180.000000, 83.623596)
Layer SRS WKT:
GEOGCS["WGS 84",
    DATUM["WGS_1984",
        SPHEROID["WGS 84",6378137,298.257223563,
            AUTHORITY["EPSG","7030"]],
        AUTHORITY["EPSG","6326"]],
    PRIMEM["Greenwich",0,
        AUTHORITY["EPSG","8901"]],
    UNIT["degree",0.01745329251994328,
        AUTHORITY["EPSG","9122"]],
    AUTHORITY["EPSG","4326"]]
CAT: Real (16.0)
FIPS_CNTRY: String (80.0)
CNTRY_NAME: String (80.0)
AREA: Real (15.2)
POP_CNTRY: Real (15.2)
```

Near the end of the output you'll find the fields included in the dataset. Note that POP_CNTRY is listed last, and the output indicates that it is a numeric field.

Of course, all the desktop GIS applications provide a way to not only determine which fields are in a dataset but actually view the data itself. In uDig we just click the Table View tab below the map view, and we get a nicely formatted view of the data, as shown in Figure 3.5, on the next page. We'll examine working with data in other applications in a bit.

Viewing the attribute table is good for just browsing around. Let's look at some more advanced ways to view and render our data.

3.4 Advanced Viewing and Rendering

Harrison has some more bird-sighting data he collected in his travels. He wants to view the sightings in a number of ways, including by species and number of birds per site. This will allow him to quickly identify where he saw individual species and large groups of the same species. Fortunately, there are some advanced rendering techniques that can help him out.

Catalog | Web Catalog | Search | Default Feature Editor | Table View | Progress View

Any | search | All

Features Selected: 0

FID	FAT	FIPS_CNTRY	CNTRY_NAME	AREA	POP_CNTRY
world_borders.1037	3	AF	Afghanistan	647500.0	2.8513677E7
world_borders.6	6	AL	Albania	28748.0	3544808.0
world_borders.4	4	AG	Algeria	2381740.0	3.2129324E7
world_borders.15	10	AQ	American Samoa	199.0	57902.0
world_borders.16	10	AQ	American Samoa	199.0	57902.0
world_borders.17	10	AQ	American Samoa	199.0	57902.0
world_borders.18	10	AQ	American Samoa	199.0	57902.0
world_borders.19	10	AQ	American Samoa	199.0	57902.0

Figure 3.5: Viewing attributes in uDig

Let's use QGIS to help Harrison classify his data. If you haven't installed QGIS yet, take a look at Section B.3, *Quantum GIS*, on page 292 for some hints. Then start up QGIS:

- *Linux*: Change to the QGIS install subdirectory, and run QGIS or use the desktop icon if installed on your platform.
- *Mac OS X*: Double-click the QGIS icon in your Applications folder.
- *Windows*: Click the Start button, find the QGIS program folder in Program Files, and choose Quantum GIS.

Once QGIS starts up, you are presented with an empty legend and map canvas. In QGIS, functions are accessible from both the menu and the toolbar. Before we get to helping Harrison, let's explore the interface a bit by loading our world_borders and cities layers. Since these are vector layers, find the tool to load a vector or use Add a Vector Layer from the Layer menu. QGIS has a lot of tools on its many toolbars, so it's best to familiarize yourself with them up front. You can do this by hovering the mouse over each tool to view the tooltip or, better yet, by reading the *User Guide*[4] that comes with QGIS. It contains a summary of the tools and includes pictures of the icons to help you get started.

Once you click the tool or menu option to load a vector layer, the file dialog box is displayed. Navigate to the directory where you placed the sample data, or you can use your own shapefile data if you have some available. Note you can choose more than one layer from the list by using the `Shift` or `Ctrl` key. This allows us to quickly add more than

4. The *User Guide* is distributed with QGIS and accessible from the Help menu.

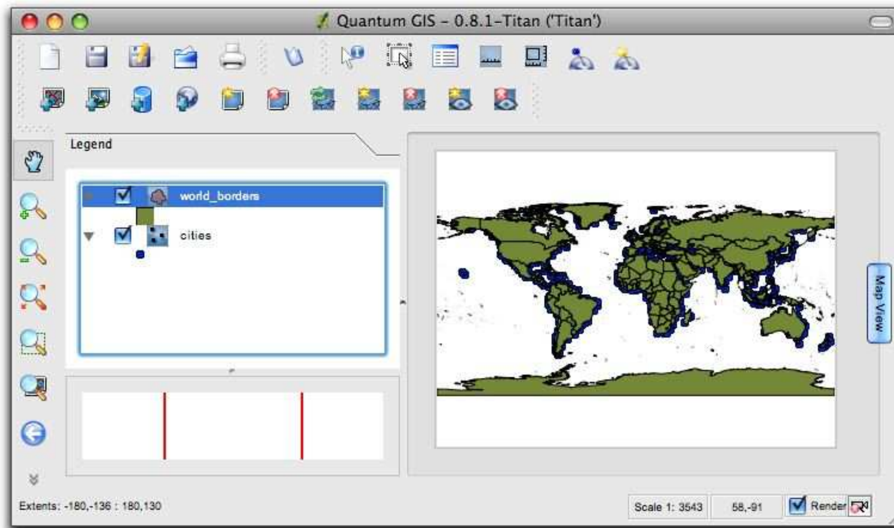


Figure 3.6: QGIS with sample data loaded (OS X)

one layer to the map. Select both the world_borders and cities shapefiles, and click the Open button to load them. If you don't see the shapefiles in the file dialog box, make sure the filter (Files of Type) selector is set to display ESRI Shapefiles.

Once the layers load, you should see something similar to Figure 3.6. You might notice something right off, apart from the atrocious colors QGIS has chosen for our two layers. The cities layer is "underneath" the world borders. This is because QGIS isn't very clever in loading layers and didn't know it should put your point data on top of your polygon data. We can easily fix this by dragging the cities layer to the top of the list in the legend. That solves the ordering problem, but the colors are still bothersome. Fortunately, QGIS has a wide range of options for symbolizing layers.

Fixing the Appearance

To fix the layers, we will use the Layer Properties dialog box. You can access the dialog box by double-clicking a layer or by right-clicking and choosing Properties from the pop-up menu. Let's start by modifying the look of the world. As you can see in Figure 3.7, on the following page, there is the somewhat busy Layer Properties dialog box and the

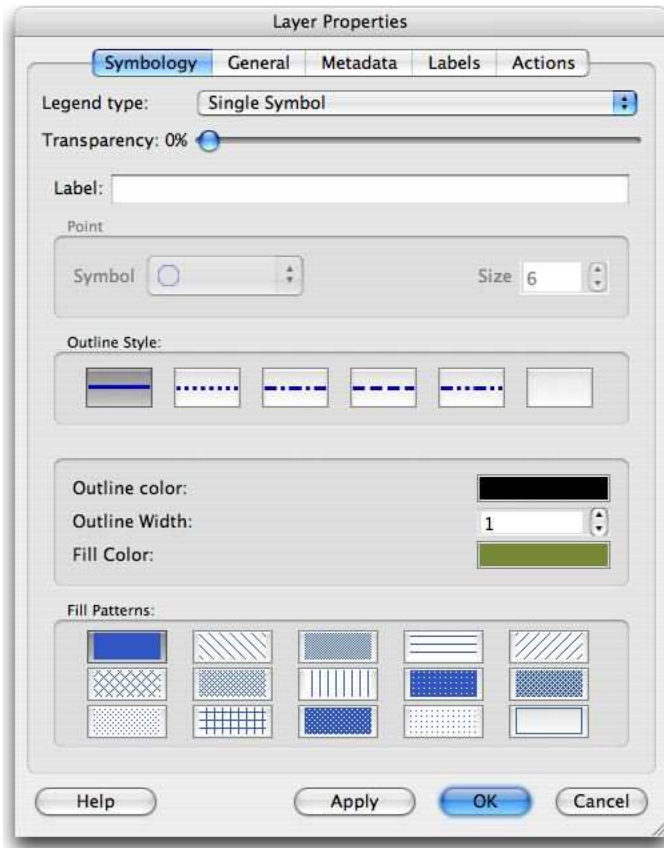


Figure 3.7: QGIS vector layer properties

current settings for the world. Let's take a bit of time to explore the options available. First off note that the ugly green color is quite visible. We'll change that in a moment. The dialog box is organized into tabs, the default being Symbology since that is the most often accessed.

Note that some of the options on the Symbology tab will not apply, depending on the geometry of the layer. By *geometry*, we mean whether it's a point, line, or polygon layer. For more information on each of the options, see Section D.1, *Vector Properties and Symbology Options*, on page 330.

Let's change that ugly green color to something more pleasing. If you don't have the properties dialog box open, double-click the world_borders

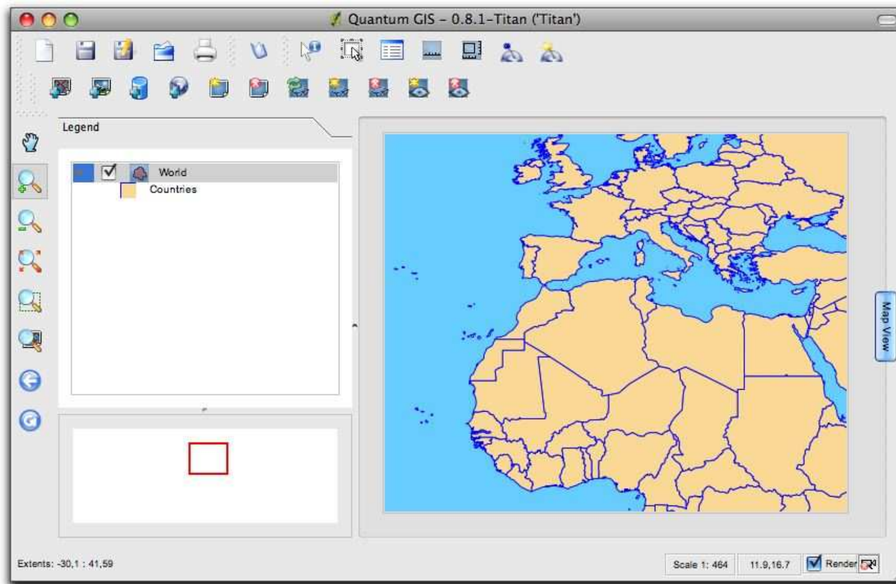


Figure 3.8: Nicely rendered world_borders layer

layer. Let's make the land a light brown color. Click the color box to the right of Fill Color, and choose a color from the selector. We'll do the same with the outline for the countries, in this case choosing a nice blue color. Once you made the selections, you can hit Apply to see the changes or OK to accept the changes and close the dialog box.

One last thing before you see the results. QGIS allows you to set a background color for the map canvas. Often this can be used to improve the appearance of the map, especially when you have large areas of white space. To set a background color, choose Project Properties from the Settings menu. To set the color, activate the General tab, and then click the color box to the right of the Background Color label. Close the dialog box, and refresh the map. The result of all the changes, as shown in Figure 3.8, shows a nice light blue ocean and the countries of the world neatly delineated.

Here are a couple of things to note about the result. We changed the name of the layer from its arcane world_borders to World. We also entered "Countries" in the Label field in the properties dialog box. Note that this label shows up to the right of our symbol box in the legend.

Figure 3.9: QGIS continuous color renderer settings

Now that we are proficient in adding a layer and adjusting its appearance, let's take a look at Harrison's bird data.

Viewing the Bird Data

To answer all of Harrison's questions about his data, we'll use QGIS's Continuous Color, Graduated Symbol, and Unique Value renderers.

Using Continuous Colors

In continuous color rendering, you set a color for the minimum and maximum values in your data, and it automatically assigns colors to each feature. It turns out this is a quick way to render Harrison's bird sightings to get a feel for the relative distribution of birds. The vector Layer Properties dialog box with the continuous color option selected is shown in Figure 3.9.

To set up the renderer, we selected a start and end (minimum and maximum) color. You could use any colors, but we chose a blue to red transition, going from dark blue to dark red. You could just as easily go from an orange to a dark green. The number of birds per site are represented from the fewest (dark blue) to a moderate number (purple) to the most (dark red). The count field from the attribute table contains the number of birds per site and is used to classify the data.

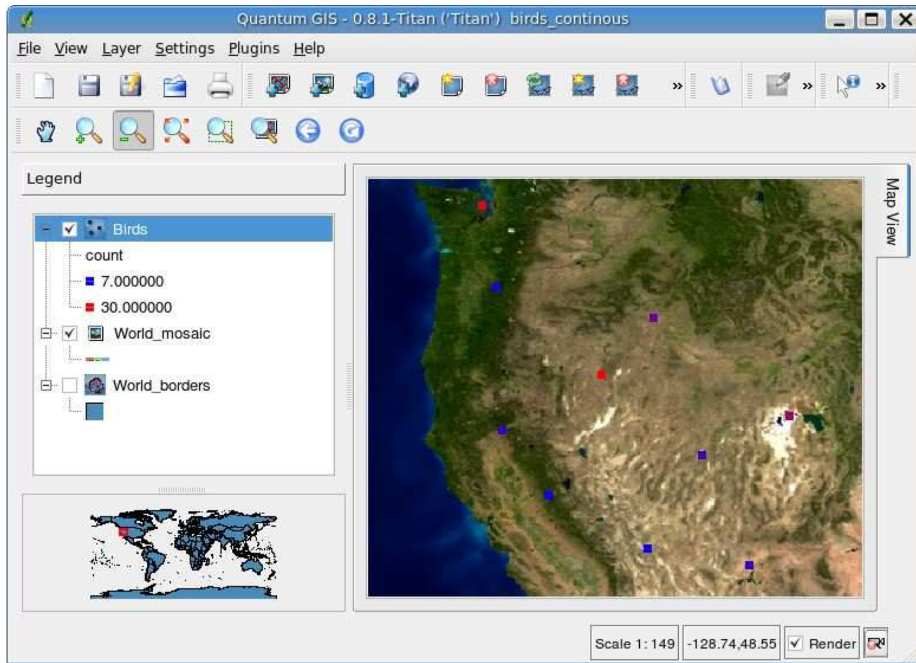


Figure 3.10: QGIS continuous color renderer results

When we apply the renderer, we get a nice display, as shown in Figure 3.10.[5] Notice that we didn't have to specify anything about the data or the individual classes—in fact, there are no options to do that. It gives us a relative view of how the bird counts are distributed, but it is purely qualitative. We can't tell from the legend what a particular color represents in terms of actual number of birds at a given location. Of course, we could use the identify tool (we haven't talked about this yet) to find out.

This is a quick way to render the data and get a feel for how things are distributed. Harrison isn't fully satisfied with the result—he wants more control, and as we'll see, the graduated renderer is better suited to the task.

5. In this and other examples, you'll notice we've added a background image to enhance the display. You'll see how to add rasters in the next chapter.

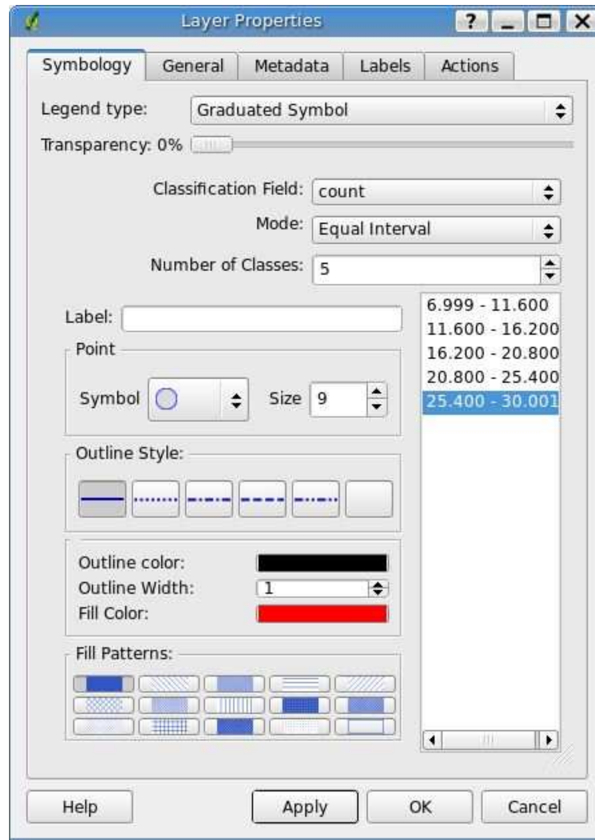


Figure 3.11: QGIS graduated renderer settings

Graduated Symbols

Let's take a quick look at using the graduated symbol render in QGIS. The renderer settings, all ready to go, are shown in Figure 3.11. QGIS currently doesn't support ramped colors or palettes. This means a bit more work when setting up the renderer. When we rendered the world by population with uDig (Figure 3.4, on page 44), it provided us with a bunch of color palettes to choose from. With QGIS we have to specify the color for each class break manually. This means a lot more work if we have a large number of classes. We have to select each one, set the color, and, if applicable, the outline color and fill pattern. Although this gives you more control, it also means more effort if we just want to do a quick render.

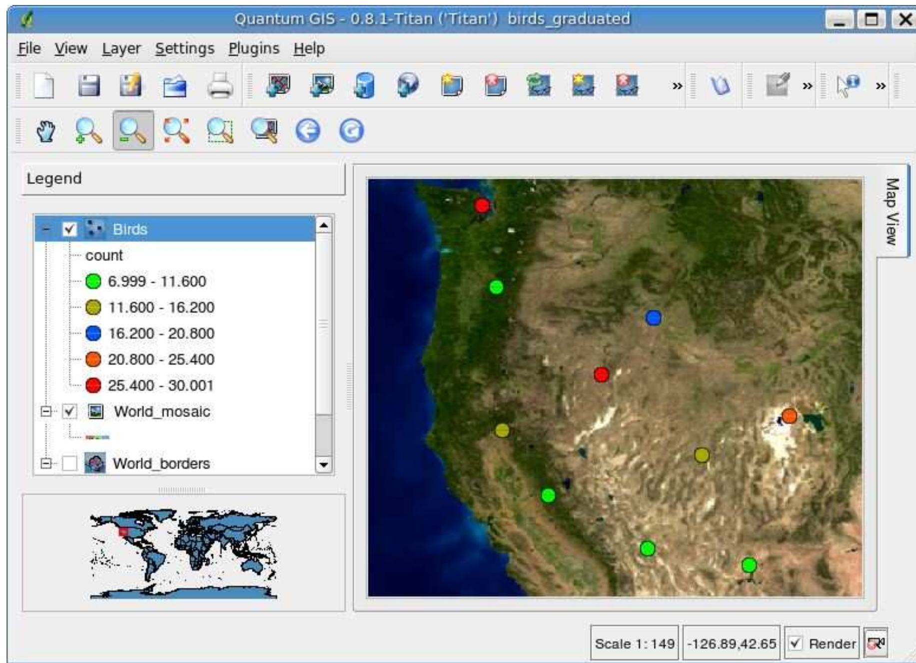


Figure 3.12: QGIS graduated renderer results

In our settings, we specified the sites with the highest bird counts should be a dark red color, while those with the least are rendered a green or tan color. As a result, you can see a lot of green dots in Figure 3.12. This is because we used only five classes. If we wanted a more granular view of the counts, we would need to increase the number of classes. Another way to refine the rendering is to edit the values in the class breaks. QGIS allows you to edit the ranges used for each class by double-clicking the number range in the list of classes (Figure 3.11, on the preceding page). You can adjust the ranges for each class to get the result you want.

The graduated renderer gives us (and Harrison) a quick way to spot the locations with the highest bird count, just by looking at the colors of the dots on the map. We could create an even more effective display by changing the symbol size of the dots for each class, starting with a smaller point size and increasing to a much larger size for the last class. In this way, the size of the dots conveys the relative number of birds at each site, and you can get a very quick idea of the distribution with just a glance.

Unique Values

Harrison can now get an idea of the bird counts at each site by using the continuous color or graduated renderers. But he also wants to view the information by species. This is where the unique value renderer comes into play.

The unique value renderer is useful for visualizing things that are the same. By that I mean rendering features using the same color when they have the same value for an attribute. Some common examples are the following:

- Displaying all the polygons for a land type in the same color, such as state lands vs. city lands
- Displaying volcanoes by their type
- Displaying roads by type: interstate, highway, secondary road, and primitive road
- And of course displaying birds by name

The common thread in that list is: display "xyz" by *type*. That's the purpose of a unique value renderer.

To display the bird sightings by name, we set up the unique value renderer as shown in Figure 3.13, on the next page. We can adjust the colors for each bird by clicking its name in the list and changing the fill color. We can also adjust the style and size of the marker symbols. Once we are happy with the setting and click OK, we get a nice display of our locations by bird name, as shown in Figure 3.14, on page 56.

Now Harrison is happy, and we have gotten a good look at using renderers to help us understand our data. While we are here, let's look at one more example that has nothing to do with birds. So far in this section we have been working with point locations. Let's take an example that is a bit more colorful and is composed of polygons—a geologic map.

A geologic map portrays rocks by type (note that word *type* again), and each type should be rendered in the same color. The unique renderer setup to display our geologic map is shown in Figure 3.15, on page 57. Notice that the rendering is done using the UNIT field. This field contains the abbreviation for the rock types.

As with the graduated renderer, you must set the color and style for each unique value. In the case of this geologic map (which is for the

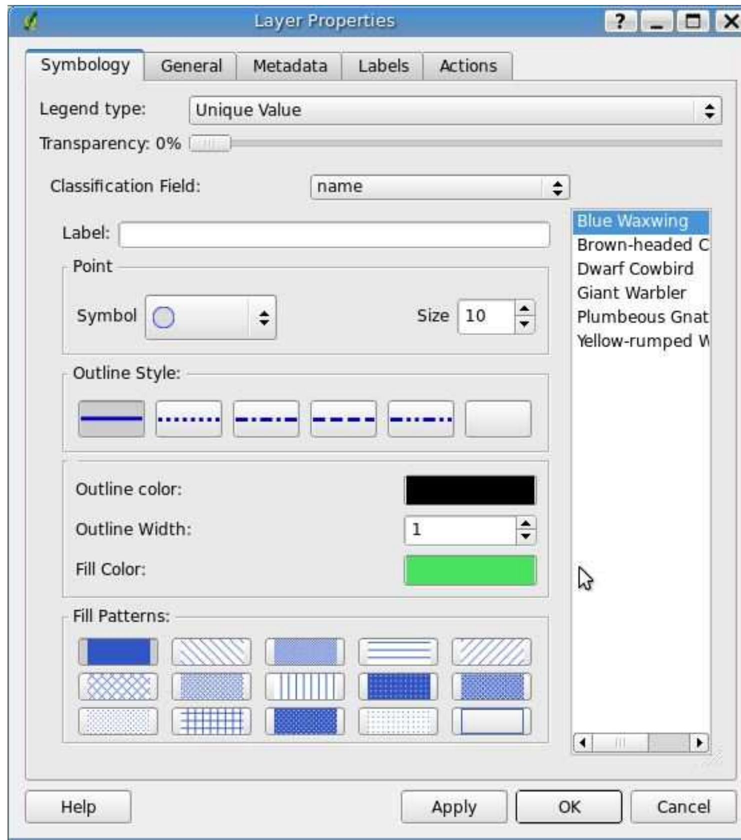


Figure 3.13: Unique value renderer for birds

Livengood quadrangle in Alaska[6]), this is a lot of work. The result of our efforts is shown in Figure 3.16, on page 58. A couple of points about the result. First, the colors don't represent those "standard" for geologic units—so you geologists out there don't get too excited. The second point is, if you wanted to make the colors match the standard, you would have to manually tweak the color for each unit. Let's hope a future version of QGIS will include support for color ramps, palettes, and custom styles to make this a bit easier.

6. You can download this and other geologic maps for Alaska from http://pubs.usgs.gov/of/1998/of98-133-a/arc/covers. The maps are in E00 format and require conversion to use (see Section 8.3, *Importing an E00 Interchange File*, on page 129 for options).

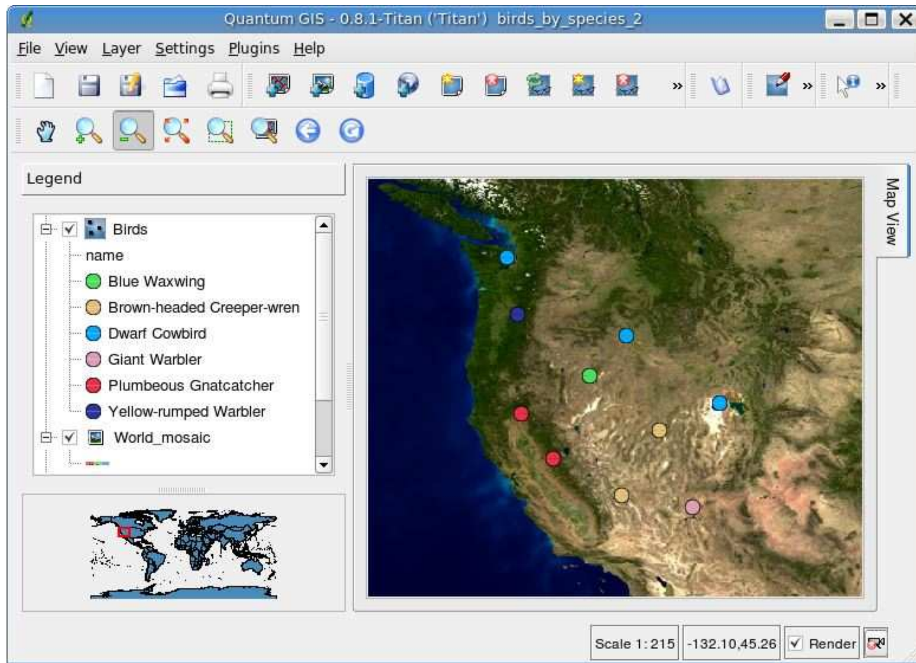


Figure 3.14: Viewing birds by name

Now that we have a good idea of how to load and render vector layers to tell a story, let's look under the hood and see what we can do with the attributes associated with our GIS data.

3.5 Making Attribute Data Work for You

So far, we haven't really dealt much with the attributes associated with our data layers, other than using them with the various renderers. In this section, we'll get into working with our attribute data and making it work for us. We will leave Harrison's birds alone to roost for a while and use the cities layer in the sample dataset to identify features, view the attribute table, select features, and attach actions to attributes. We will be using QGIS to illustrate how to work with attribute data, but you will likely find similar capabilities in the other applications we've mentioned so far.

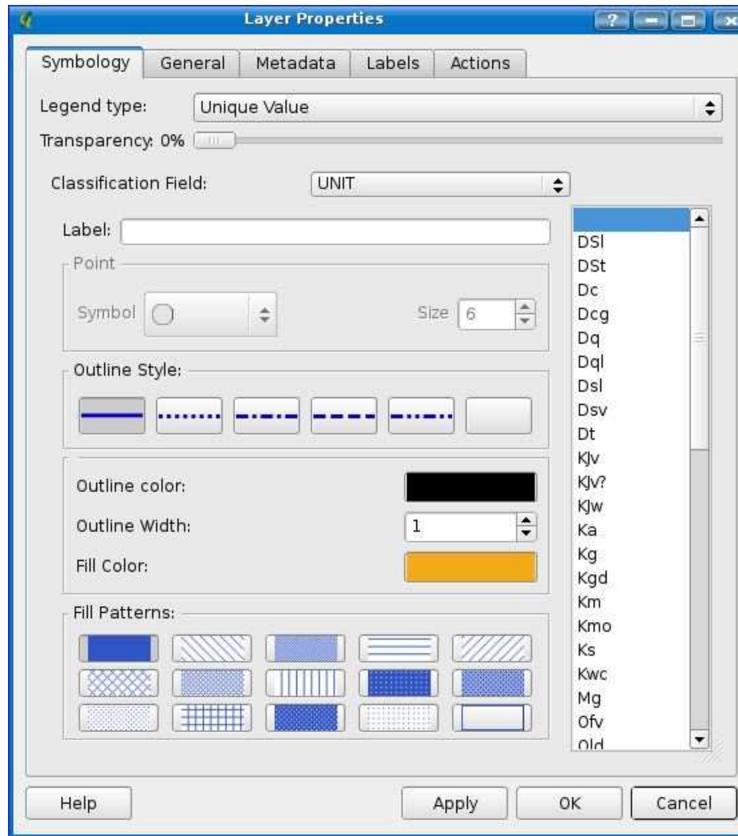


Figure 3.15: QGIS unique renderer settings for a geologic map

Identifying Features

If you want to follow along, open QGIS and load the cities and world_borders layers.

Here is one of the most common questions when working with GIS data—what is that feature, and what can I learn about it? It's also one of the simplest operations we can perform. Once we load the cities layer, we have 606 dots on our map. The next trick is to determine which is which. Sure, we can identify some of them just by looking, assuming we are familiar with the country. This is where the Identify tool comes into play—it's used to query a feature on the map. Simply zoom to an area of interest, activate the layer by clicking its name in the legend, click the Identify tool, and click the feature. QGIS will dig up

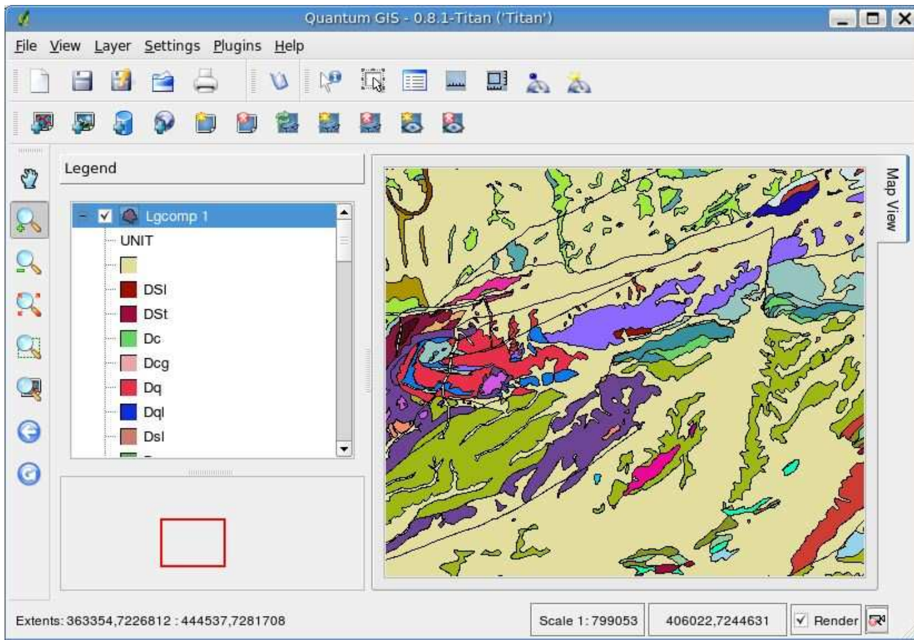


Figure 3.16: QGIS unique renderer result

some information about the feature and open the results dialog box, as shown in Figure 3.17, on the following page.

The results show each field name in the left column and the corresponding value in the right column. You'll have to expand the result nodes to see the field values if more than one feature is returned. In Figure 3.17, on the next page, we have identified Dublin, Ireland, and can see that it is a capital city with a population of 1,140,000.

If you attempt to identify a feature and QGIS tells you it can't find anything at that location—despite that you're sure you clicked it—you'll have to adjust the tolerance used for finding a feature. This setting is on the Map Tools tab of the Options dialog box, accessible from the Settings menu. It's specified as a percentage of the map width. If the default value isn't working, try increasing it. If you increase it too much, you will end up with multiple features returned instead of the one you want. A setting of 0.7% is probably a good starting point for a map with global extent. If you still don't get any results, check to make sure you have activated the layer by clicking its name in the legend.

Figure 3.17: QGIS Identify results

Selecting Features

Selecting features is another common operation when working with attributes. A selection set is used for a number of operations, as we'll see shortly. Here we'll illustrate a simple usage—zooming to the extent of the selection set.

In QGIS there are two ways to select a feature. The first is using the Select Features tool, located next to the Identify button. Click the tool, activate the layer, and then drag a rectangle on the map around the feature(s) you want to select. When you release the mouse button, the features contained in the rectangle will be selected and drawn in a highlighted color (the color is customizable from the Options dialog box).

The other way to select features is to use the attribute table selection tools (see Section 3.5, *Using the Attribute Table*).

Once you have a selection set, you can zoom to it by using the Zoom to Selection tool or by using the menu option found in the View menu.

Using the Attribute Table

By now it's clear that GIS layers have attributes associated with the features.[7] The attribute table not only gives us a view into the data behind the features but in a typical application allows us to edit, select, and search.

In QGIS, as in all desktop GIS applications, you can view the attribute table for a layer. The attribute table for the cities layer is shown in

7. Usually this is true—you could have a layer with no attributes.

Attribute table - Cities

Start editing | Stop editing

	id	NAME	COUNTRY	POPULATION	CAPITAL
1	250	Abidjan	Ivory Coast	1950000	Y
2	587	Abu Zaby	Untd Arab Em	242975	Y
3	414	Acapulco	Mexico	301902	N
4	249	Accra	Ghana	1250000	Y
5	134	Adana	Turkey	777554	N
6	314	Adelaide	Australia	977721	N

Help | Search for: | in NAME | Search | select | Advanced... | Close

Figure 3.18: Attribute table for the cities layer

Figure 3.18. You'll note that there is a row for each feature (city) and columns for each attribute (field) in the layer.

This is a very busy dialog box with lots of buttons and features. We'll start with the basics and come back to the more advanced features later. First, you can sort the table by any field by clicking the column header. Clicking repeatedly toggles between ascending and descending sort order. You can scroll through the table and randomly browse the attributes. This is a good way to introduce yourself to a newly acquired dataset.

You can select features by holding down the left mouse button and dragging down through the rows. You can also select rows by using the `Shift` and `Ctrl` keys, just as you do in a selection box on your operating system.

As you select records in the attribute table, the corresponding features on the map canvas are highlighted.

Quick Search

Say we want to find a particular city—for example, Cuzco in Peru. There are several ways we could do it:

- Using the Identify tool, we could randomly click the cities in Peru until we find it.
- Turn on labels for the cities, zoom to Peru, and look around until we find it. This works pretty well, but if we didn't know Cuzco was in Peru, it would become a bit more difficult.

- We can open up the attribute table, sort it by name, and scroll down until we find Cuzco.
- We can search for it.

These methods are in increasing order of efficiency. For small datasets, browsing the attribute table until you find what you are looking for is reasonable. When you get thousands of records, it can become a chore, and searching becomes the way to go.

You can quickly search the table for a city by entering a search term in the Search For box, selecting the field you want to search from the drop-down list, and clicking the Search button. In our case, we want to enter "Cuzco" as the term to search for, and we want to look in the NAME field. The search will return both full or partial matches. When searching, there are three options you can choose from:

- *Select*: Select the features that match.
- *Select and Bring to Top*: Select the features that match and promote them to the top of the attribute table display.
- *Show Only Matching*: Show just the features that match.

Which option you choose depends on the purpose for selecting the features. If you just want to select them so you can zoom the map to the selection set, then the first option is sufficient. If you want to view the attributes of the selection set, then the second option is the one you want. Of course, you could just scroll through the table and look for highlighted rows, but promoting them to the top makes it quicker and easier to browse the results.

Four tools at the top of the attribute dialog box can be used to manipulate the selection set. Use the mouse to hover over each to learn its function. The tools available are as follows:

- Remove Selection
- Move Selected to Top
- Invert Selection
- Copy Selected Rows to the Clipboard

If you copy the selection to the clipboard and paste into a text editor, you get a comma-delimited list of the attributes in the selected rows, complete with a header row containing the field names. Doing this for Cuzco and pasting it, we get the following:

```
wkt_geom,NAME,COUNTRY,POPULATION,CAPITAL
POINT(-71.860001 -13.600000),Cuzco,Peru,      184550,N
```

The X and Y of Latitude and Longitude

In conversation, most people say "latitude and longitude," not the other way around. In fact, you will most often see it written that way as well. So, it's natural for people to assume the latitude = X and longitude = Y (since we say X,Y), but this isn't so. It's an extremely common mistake, especially for newcomers to the GIS realm. For the record, lines of longitude run vertically and measure units in the X direction. Lines of latitude run horizontally and measure units in the Y direction.

The output can be used for importing into another application for further manipulation or reporting purposes. You might be wondering about the funny-looking POINT notation. It's the Well-Known Text (WKT) representation of a point, consisting of the feature type keyword, in this case POINT, and the coordinates (X and Y) separated by a space. The X and Y values are in the coordinate system of the layer. In the case of our cities layer, it's geographic (longitude, latitude). We also have the name, the country, its population, and whether the city is a capital.

Advanced Search

The attribute table also provides an advanced search query capability where you can really narrow down what you are looking for in the dataset. The quick-search feature allows us to specify only a single search term and field to search. With the advanced query, we can be more specific by using SQL.[8] Don't worry if you aren't a SQL expert or don't even know what it stands for—QGIS makes it easy, as we will see.

Say, for example, we want to find all the cities in the world with more than 2 million people that are also capital cities. We can easily do this with the Search Query Builder. To access the builder, open the attribute table, and click the Advanced button. The query builder populated with the terms needed to find the cities of interest is shown in Figure 3.19, on the following page. Now let's take a look at how this dialog box works.

The fields in the attribute table are listed on the left side of the dialog box. On the right is a box that can be used to display the values for a field. This allows you to get a preview of a field's contents to aid you in building the query.

8. SQL is a standard language for querying and updating a database.

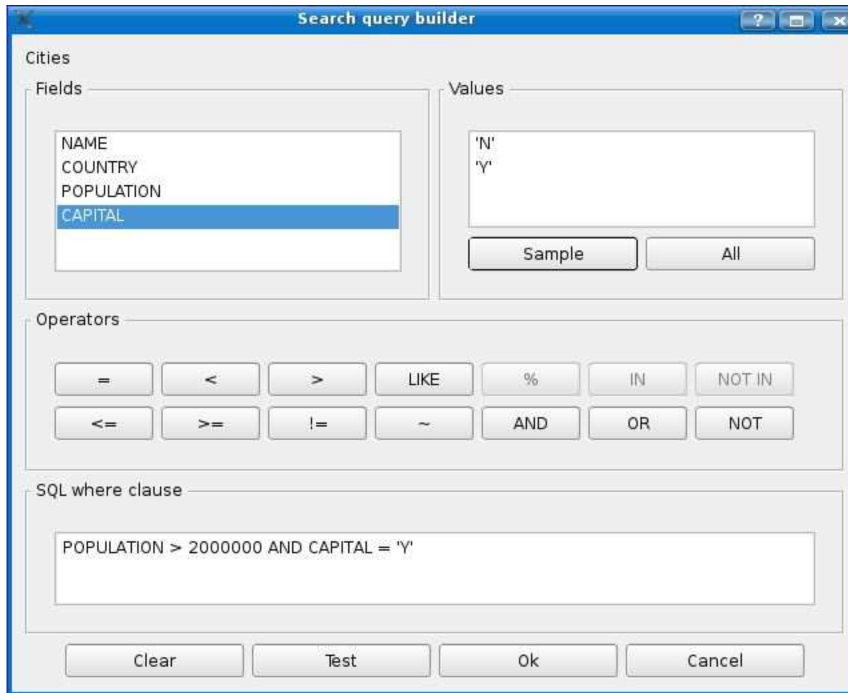


Figure 3.19: Search Query Builder

To preview a sample of the data, use the Sample button. This will pull a subset of the unique values for the field. Viewing a sample is a good method to use when the attribute table is very large because it saves time. If you want to see all possible values for a field, use the All button.

These two boxes also serve another purpose. You can add an item in the box to the query by double-clicking it. To start building our sample query, we double-click the POPULATION field to add it to the SQL where clause. To complete the population part of the query, we click the "greater than" operator button and then type in "2000000." This gives us the following:

```
POPULATION > 2000000
```

When adding field names and operators, QGIS automatically inserts the needed spaces. If we test the query now using the Test button, it reports 110 matching features (cities). To complete the query, we click the CAPITAL field and click the ALL button to get a list of all possible values. It's easy to see there are two possibilities—Y and N. Since we are using a value from the list, we don't have to type anything to complete

the query. First we add the AND operator by clicking the button in the Operators section. To add the final parts, just double-click the CAPITAL field, click the = operator, and double-click Y in the value list. Our query is now complete:

```
POPULATION > 2000000 AND CAPITAL = 'Y'
```

If we test this query, QGIS reports forty-one matching cities. When we click OK, the attribute table will have those forty-one cities selected. We can promote them to the top of the table using the Move Selected to Top button in the Attribute Table toolbar. Since these features are selected, they will be drawn the map using the selection color preference you have set in your QGIS preferences or the project properties.

You can construct very complex queries using the query builder. If you are familiar with SQL, you can bypass the click-and-build routine and enter the where clause manually. In either case, the ability to define custom searches is a powerful tool. In a future chapter, we will see how to use this same concept to create a subset or multiple views of a layer.

Using Attribute Actions

Let's conclude our look at working with attribute data by doing something useful with the attributes in the cities layer. QGIS has the concept of *attribute actions*, in other words, performing some action (think task) using the value of an attribute. With an attribute action, you can call another application and pass the value of the attribute to it. Here are some potential uses for this feature:

- Do a web search based on one or more attribute values.
- Display a photo based on a location stored as a layer attribute.
- Submit values to a URL that creates a report.
- Query a database based on attribute values.

There really is no limit to the way in which you can use actions to integrate QGIS with other applications and tools. Let's take a simple example and do a Google search of a city using the results from the Identify tool. Attribute actions are defined from the Actions tab on the vector Layer Properties dialog box. The steps to create an action are as follows:

1. Determine the attribute field(s) needed.
2. Determine the application that will "drive" the action.
3. Construct the argument string using the attribute value(s).
4. Create the action.

Creating the Action

In our example, we are going to use the name of the city and pass it to Google. The application we need is a web browser. In this case, we will use Firefox. The URL to search for something with Google is http://google.com/search?q=find_me, where find_me is the search term. Attribute actions work by replacing specific strings in our argument list with the values from the attribute table. The format of these strings is simple—it's a percent sign (%) followed by the name of the field we want to use. In our example, we therefore need to use *%NAME* as the replaceable parameter. Putting this altogether gives us the following action:

```
firefox http://google.com/search?q=%NAME
```

One important thing to note here—the browser must be in our path. If not, the action will fail. The alternative and perhaps the safe way is to fully specify the path to the browser:

```
/usr/bin/firefox http://google.com/search?q=%NAME
```

If you have spaces or other oddities you need to specify in the action, use double quotes around the entire thing. They will be ignored when the action is executed but safely allow you to specify the path and other exotic parameters.

So now that we have the text for the action figured out, let's put it all together to create and use the action. First open the properties dialog box for the cities layer and click the Actions tab. Give the action a name, "Google search" will work, and then enter the text of the action. If you are following along, make sure to adjust the path for your browser. Click the Insert Action button to add the action to the list. We could go ahead and create other named actions in the same way. If you make a mistake, you can click the action in the list and edit it.

When you are finished, click the Update action button to save your changes. Notice we didn't use the Insert Field button and its associated drop-down list of field names. That's because we determined beforehand we were using *%NAME*. Also notice the browse button to the right of the action text box. If you click this button, it lets you browse to the location of the application you want to use to execute the action. In our example, we could have used it to browse to the location of Firefox (/usr/bin/firefox). This is useful if you aren't sure of the full path for the application needed to execute the action.

To complete the creation of the action, click OK. It's now ready to use.

Figure 3.20: Attribute action enabled in QGIS

Using the Action

Now that we have it, let's see how to use it. The results of identifying a city on the map are shown in Figure 3.20. If you compare this with Figure 3.17, on page 59, you will see we now have a new entry at the bottom of the results list. This is our action, appropriately labeled "Google search." To execute an action, you can click it or right-click and choose the action from the pop-up list (if we had defined more than one action, we would choose it from this list). When clicked, QGIS will launch Firefox and execute the Google search for New Orleans. You can identify another city and use the action to perform a Google search on it. Depending on your operating system and browser, it may reuse the current browser window or open a new one.

Now it should be clear where attribute actions could come in handy. There is one last trick you can use when defining actions. If you use the special parameter %% instead of a field name, QGIS will replace it with the value of the currently highlighted field in the identify results list. In the case of our example, this would allow us to do a search on any field value in the layer. Most of our fields in the cities layer aren't well suited for Google searching, but the COUNTRY field would return useful results. Being able to specify values in this way, as well as the ability to define multiple actions gives us a lot of flexibility.

What you do with attribute actions is now limited only by your imagination and cleverness.

Now that we've exercised our vector muscles, let's move on and work with some raster data.

Where there is no vision, the people perish. . . .
► Proverbs

Chapter 4

Working with Raster Data

Raster data is everywhere in the GIS world. You can use it as a background layer for your vector data or do full-blown analysis with it. In this chapter, our goal is to get you up and running with raster data. In later chapters, we'll delve into some analysis and manipulation.

Nearly every OSGIS desktop application can display at least some raster formats—and some more than others. In particular, those applications that are based on the GDAL library can support an impressive range of raster data. You can find a partial list of formats that GDAL supports in Section A.2, *GDAL/OGR*, on page 283. Both QGIS and GRASS use the GDAL library for reading and writing raster data.

4.1 Viewing Raster Data

We'll start with something simple in our endeavor and load a TIFF image. A fairly common thing you might want to do is view a topographic map of your area. You might recall that this is what Harrison used as a background for his bird data. In the United States, many of these rasters can be downloaded from the U.S. Geological Survey website. Using a topographic map as a base is useful when you want to view your vector data (for example, GPS tracks and waypoints) over it.

Let's download a TIFF from the Internet Archive of USGS Maps.[1] You can pick any state you like—for our example, we'll grab a random image from Montana.[2] You can pick one for your area by browsing the archive by state. We'll need both the .tif and .tfw files. Once you have your raster and the world file, you can view it in QGIS.

1. http://www.archive.org/details/maps_usgs
2. In case you're interested, we grabbed http://www.archive.org/details/usgs_drg_mt_47113_g1.

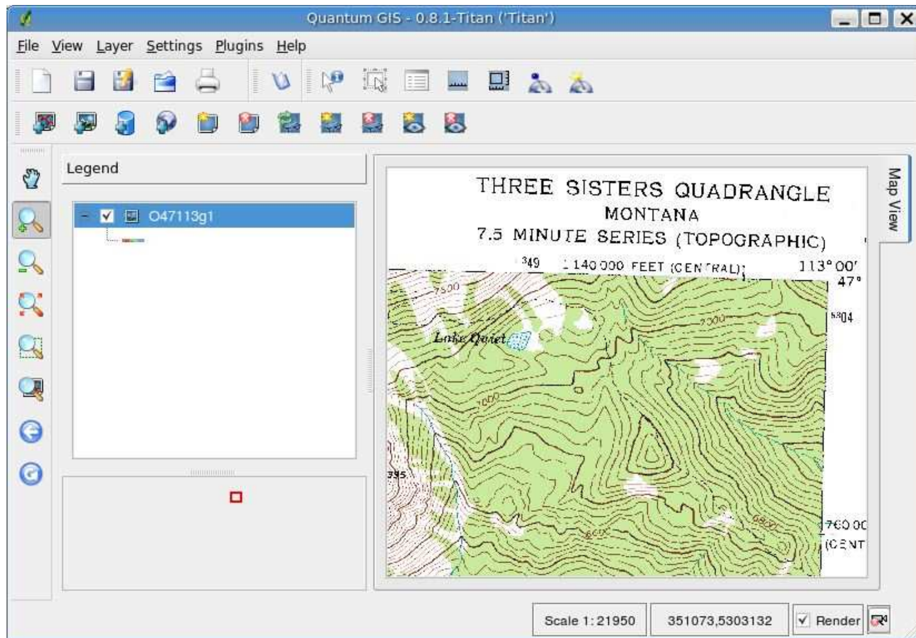


Figure 4.1: Montana topographic map in QGIS

The QGIS toolbar contains a button for loading rasters (it's right next to the Vector button), or you can choose Add a Raster Layer from the Layer menu. Rasters in QGIS are loaded using the standard file dialog box—of course this will vary in appearance depending on your operating system. To select a TIFF, we need to make sure the filter is set properly. The QGIS raster dialog box has filters for a number of data types, including GeoTIFF, ERDAS Imagine, and USGS Digital Elevation Models. To get started, just make sure the GeoTIFF is selected in the Files of Type drop-down box, and navigate to the location of the TIFF file. Select it and click the Open button. QGIS opens and displays the image at full extent. In Figure 4.1, we have loaded the raster and zoomed to the northeast corner of the Montana DRG.

The first thing you're likely to notice is that the raster has a bunch of text annotations around the border. We call this the *collar*, and although it has good information, it's a bit distracting when we're working with our data. This becomes readily apparent when you download the DRG adjacent to it and want to display them in a seamless fashion.

Joe Asks...

What Is a Georeferenced TIFF?

It's a TIFF image that has metadata (information) about its coordinate system. This information can be associated with the image in a couple of ways. In a GeoTIFF, the coordinate system information is "embedded" in the file itself—in other words, it is self-contained.

The other way to specify coordinate information is with a world file. A world file contains information about the map units per pixel in the image, as well as the real-world coordinates of the upper-left corner. For a TIFF, the world file usually has a .tfw extension. World files are used with other georeferenced image formats, including JPEG (.jgw) and PNG (.pgw). For software that employs the GDAL library for raster access, the .wld can be used with TIFF, JPEG, PNG, and other supported formats.

We'll take up this issue in a bit and show you how to create a seamless raster from several TIFF images in Section 10.6, *Clipping Rasters with GRASS*, on page 167.

With your topographic map loaded, feel free to zoom around and explore the countryside. If you take the Identify tool and click the raster, you'll find it doesn't yield much in the way of information. In fact, the only thing it will tell you is the palette index of the pixel where you click. This is because rasters are composed of cells (a pixel is a cell) and contain only one value. In the case of a DRG, that's the palette index number. For each index, there is a corresponding color value. So for the Montana DRG, if we click a lake or stream, we find that the palette index is 2. This isn't all that useful, but when we get to Section 4.3, *Intelligent Rasters*, on page 76 you'll see other rasters where the cell values convey significant information.

The last thing we need to mention is the coordinate system for this raster. If you open the Raster properties dialog box (just double-click the raster name in the legend) and click the Metadata tab, you'll find that my Montana raster is in UTM Zone 12, NAD27 datum. You can glean that information from the Layer Spatial Reference System section of the dialog box.

Another way to get the same information is with gdalinfo, a utility that comes with GDAL and is included in FWTools:[3]

```
$ gdalinfo o47113g1.tif
Driver: GTiff/GeoTIFF
Size is 4769, 6920
Coordinate System is:
PROJCS["NAD27 / UTM zone 12N",
    GEOGCS["NAD27",
        DATUM["North_American_Datum_1927",
            SPHEROID["Clarke 1866",6378206.4,294.9786982139006,
                AUTHORITY["EPSG","7008"]],
            AUTHORITY["EPSG","6267"]],
        PRIMEM["Greenwich",0],
        UNIT["degree",0.0174532925199433],
        AUTHORITY["EPSG","4267"]],
    PROJECTION["Transverse_Mercator"],
    PARAMETER["latitude_of_origin",0],
    PARAMETER["central_meridian",-111],
    PARAMETER["scale_factor",0.9996],
    PARAMETER["false_easting",500000],
    PARAMETER["false_northing",0],
    UNIT["metre",1,
        AUTHORITY["EPSG","9001"]],
    AUTHORITY["EPSG","26712"]]
Origin = (339751.702040000003763,5305307.515394000336528)
Pixel Size = (2.438400000000000,-2.438400000000000)
...
```

We didn't include all the output from gdalinfo; we included just enough for you to see the projection information. If you recall earlier, I told you that when downloading, we needed the .tfw file, which is actually the world file for the raster. The fact that the projection information is reported by gdalinfo means that it is a GeoTIFF and contains not only the coordinate information needed to properly display it but also the projection information. You don't need the world file for a GeoTIFF. The fact that the DRG was a GeoTIFF wasn't readily apparent from the website, so we played it safe and downloaded the world file as well. Fortunately, we didn't waste much bandwidth downloading it since world files are only a few hundred bytes in size.

Let's return to our global theme now and view a raster mosaic, courtesy of NASA Visible Earth.[4] If you want to follow along, you can fetch

3. If it's not already on your system, see Section B.7, *FWTools*, on page 295 for information on installing FWTools.

4. The ev11612_land_ocean_ice_8192 image is owned and provided by NASA. The image was obtained from the Visible Earth (http://visibleearth.nasa.gov) and developed by the Earth

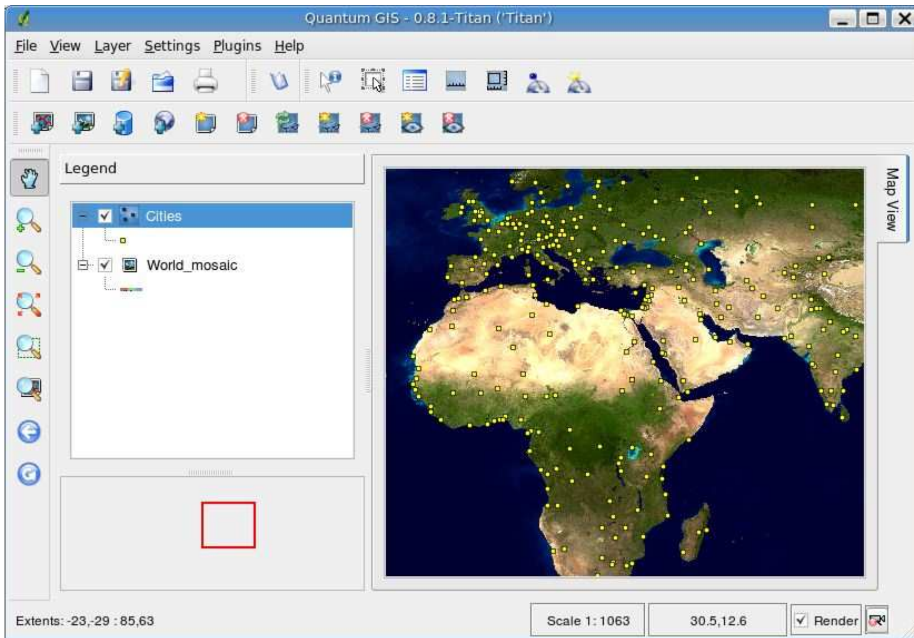


Figure 4.2: NASA world mosaic viewed in QGIS

the raster from http://desktopgisbook.com/sample_data or the NASA site. Loading it into QGIS gives us a world mosaic, as shown in Figure 4.2.

The NASA raster is a georeferenced image in geographic coordinates, meaning it can be used in conjunction with our world vector layers. If you look carefully at Figure 4.2, you'll notice we've added the cities on top of the raster, just to prove they line up. Fortunately, both the raster and vector data have geographic coordinates in the same *datum*. In case you're wondering, a datum is a model of the shape of the earth used to measure positions. In this case, the coordinates are in WGS 84, the same datum commonly used in modern GPS units. We'll take a further look at datums and projections in Chapter 9, *Projections and Coordinate Systems*, on page 138.

Observatory team (http://earthobservatory.nasa.gov)

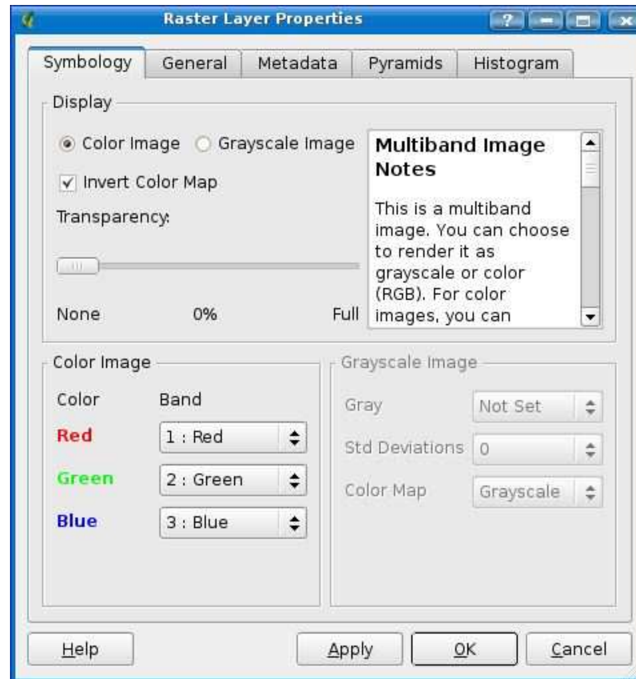


Figure 4.3: QGIS raster properties dialog box

Raster Properties

Now we'll take a brief look at some of the properties associated with a raster. Just like vector layers, in QGIS there is a properties dialog box that allows us to adjust the appearance of the image, as well as get some information about it. You can access the properties dialog box by double-clicking the layer name or right-clicking it and choosing Properties. If this sounds familiar, you are correct. It's the same method used for accessing the vector properties dialog box. In Figure 4.3, you can see the raster properties dialog box.

As you can see, the raster properties dialog box bears a bit of resemblance to the vector properties dialog box. They both have Symbology, General, and Metadata tabs. Rather than look at each of these in detail, we'll focus on a couple of things you need to know to effectively use images in QGIS. On the Symbology tab, you have the choice to display the image as either color or grayscale. By default, the appropriate display mode is chosen for an image when you load it.

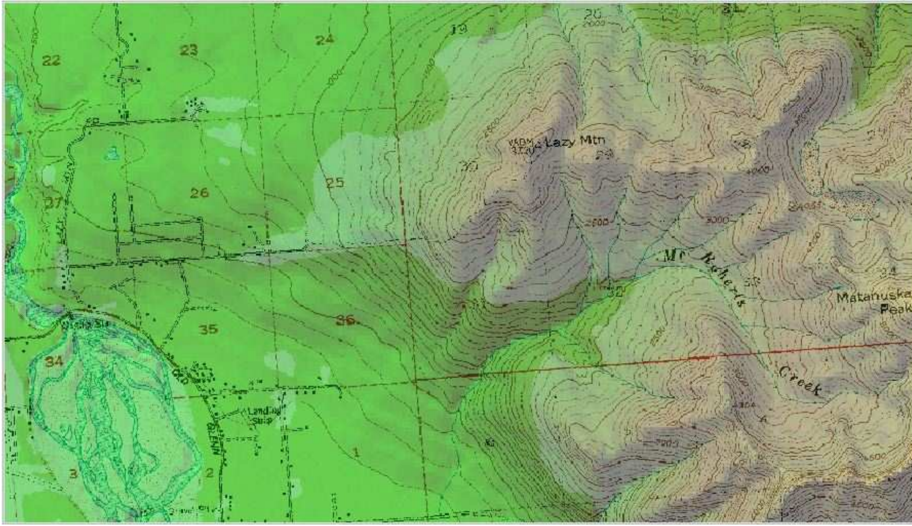


Figure 4.4: Semitransparent digital elevation model draped over a DRG

In the case of our example, we have a multiband image consisting of three bands: red, green, and blue. We can do a number of things with the bands, including changing which band is used for which color. In other words, we can swap the mapping of color to band and observe the effects. We can also invert the colors, which swaps the light and dark colors. The transparency of a raster can be set using the slider, allowing the layers underneath to become visible. This can be used to get some interesting effects, such as those in Figure 4.4.

If you are working with a grayscale image, you can adjust the color map and standard deviation used to display the raster. See the QGIS user guide to learn more about these options. One other thing to note is you can display a color image as grayscale by selecting the Grayscale radio button and then setting the band in the Gray drop-down box. So, for example, we can display our color image as grayscale using the green color band (band 2) to determine the appearance of the image. This might be used to bring out features or characteristics not visible when the image is displayed in color.

4.2 Improving Rendering with Pyramids

Pyramids are essentially multiple views of a raster at reduced resolutions. Using pyramids means that the software, in our case QGIS, doesn't have to draw every detail of the image to get it on the screen.

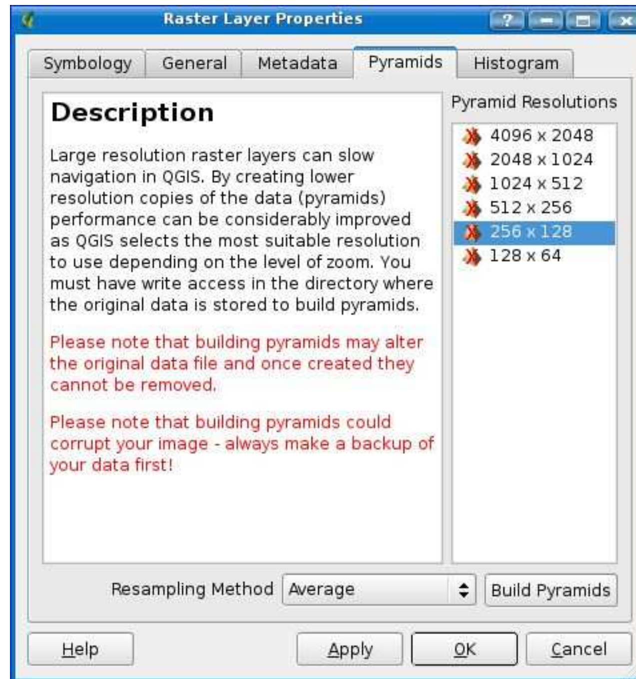


Figure 4.5: QGIS raster pyramids dialog box

This is appropriate because all that detail is lost on you at small scales. QGIS supports building and using pyramids through the GDAL library. Some software stores pyramids in external files, but GDAL builds and stores them by default within the image itself. This means your original image will be altered and in fact grow in size. You may want to make a copy of the image before creating pyramids because the process is not reversible. GDAL supports the creation of external pyramid files for some raster drivers. In our case, we will use the default and add the pyramids to our TIFF image.

To create pyramids, first open the raster properties dialog box, and select the Pyramids tab. Note the warning about altering and possibly corrupting the image, and make the backup copy before you proceed. QGIS populates the Pyramid Resolutions list with a set of resolutions appropriate for your image. The dimensions of the world mosaic image are 8,192 by 4,096 pixels. In Figure 4.5, you can see that QGIS offers to build a range of pyramids based on the dimensions of the image. These levels are calculated by dividing the width and height down to

some minimum level. Notice the small X over the pyramid icon for each level. This indicates that pyramids do not exist for that level. This of course changes once we build pyramids for the raster. You can choose a sampling method for calculating the pyramids. If you are unsure what to do, just take the default value because it will give you acceptable results. Select the levels you want to build by clicking each one. The more you select, the longer it will take to build, and more important, the larger your image will grow. Click the Build Pyramids button, and wait while QGIS generates the pyramids for each selected level in the list. This can take a while, especially for large images. Once complete, the small X will disappear from each level for which pyramids were built.

You should now see improved performance when drawing the image at smaller scales. You might also notice some degradation in the appearance of the image; however, at large scales (when zoomed in), the full quality of the image is preserved. You may have to experiment a bit to determine how many and which levels you want to build. For this reason, be sure to keep a copy of the original image in a safe place.

You can also create pyramids using the GDAL utility gdaladdo. As with QGIS, the pyramids are by default added to the original raster. If you want to create pyramids for a lot or rasters, using gdaladdo is the way to go. You can write a small script (shell, Python, Ruby, or Perl) to process each file in a directory and add the pyramids. Here is an example of using gdaladdo to create pyramids for our Montana DRG:

```
$ ls -lh o47113g1.tif
-rw-r--r-- 1 gsherman gsherman 6.8M 2007-07-09 17:24 o47113g1.tif
$ gdaladdo -r average o47113g1.tif 2 4 8 16
0...10...20...30...40...50...60...70...80...90...100 - done.
$ ls -lh o47113g1.tif
-rw-r--r-- 1 gsherman gsherman 11M 2007-07-11 19:49 o47113g1.tif
```

Here we created four levels of pyramids using the “average” resampling algorithm. We listed the size before and after the operation. Notice that building pyramids increased the raster from 6.8 megabytes to 11. For more information on gdaladdo and its options, see the documentation.[5]

Creating pyramids for your raster data can give you a huge performance gain when rendering at various scales. You may want to make a backup copy and then experiment with the various levels and sampling methods to see which provide the best results for your data.

5. http://www.gdal.org/gdaladdo.html

4.3 Intelligent Rasters

All rasters are intelligent—at least in the sense that they convey information. So far we have looked at rasters (DRGs and our world mosaic) that are useful for general viewing or as a background layer. Typically these rasters have cell values that don't really mean anything—they are just a value used to determine how each pixel should be colored. Cell values in a raster (or *grid* as these are often called) can be either integer or floating point, depending on the capabilities of your software. You might be wondering what sort of data can be represented by a raster. The answer is anything you can count, measure, or identify for a defined area on the ground. Some examples include the following:

- Rock or soil types, specified by a unique numeric code for each
- Vegetation types
- Elevations
- Quantity of an element such as gold or silver, determined by sampling and analysis

Why not use a vector layer rather than raster to delineate data by "type"? Oftentimes it's appropriate to use a vector layer, as we did for our geologic map in Figure 3.16, on page 58. Here each rock type is represented by a polygon and rendered by value. It really depends on what you need to accomplish with the data. In the case of a grid containing quantities that we want to use in an analysis, the raster model is the right choice.

Let's look at a couple of examples of what we might term a "smarter raster," in this case a Digital Elevation Model (DEM) and a grid containing measured quantities of silver.

Digital Elevation Models

The cells in a DEM contain an elevation value. For any location on the DEM, we can determine the elevation by examining the value of the cell. The smaller the size of the cells, the more resolution you will get from the DEM. We can do a number of interesting things with a DEM, including the following:

- Display it as is for a backdrop.
- Create a shaded relief to display terrain.
- Perform arithmetic operations on the individual cells to create new values.
- Create contours from the elevations.

We'll look at DEMs in more depth in Chapter 10, *Geoprocessing*, on page 149, where we'll see how to use them in analyzing, creating contours, and generating shaded reliefs—also known as *hillshades*.

The Grid of Silver

Let's see how we can use a grid to do some qualitative analysis. Harrison's interests are diverse—this time he's off to do some prospecting for silver. In terms of the methodology, he could just as easily be examining the concentration of hazardous materials or four-leaf clovers.

Harrison has found the results of a soil sampling survey that was done over a regular grid. Each sample was analyzed for a number of elements, but he is interested only in silver. Fortunately for Harrison, along with the results is a raster grid in Arc/Info Binary Grid format. Harrison fires up QGIS and finds he can easily load the grid by choosing the Add a Raster Layer tool from the toolbar and changing the file type to "AIG GRASS and all other files (*)." The grid consists of a directory containing several files. The one we (and Harrison) are interested in is the .adf file. This contains the grid data, and QGIS knows how to display this format. In Figure 4.6, on the following page, you can see what the grid looks like loaded into QGIS and underlain by the DRG for the region. In order to see the underlying DRG, we've made the grid partially transparent using the transparency slider on the raster properties dialog box.

Notice that the grid is "tilted." That's because it has been transformed from the original coordinate system, a simple X-Y grid, to one that places it where it belongs in the real world. From looking at it, we can conclude that whoever laid out the grid didn't do it using cardinal directions of north-south and east-west. Transforming it to the proper coordinate system also gives it a ragged appearance along the edges since it's not possible to approximate a straight line given the cell size in this grid.

We can use the Identify tool just as we did when working with our vector data (Section 3.5, *Identifying Features*, on page 57). Unlike vector data where we might have several attributes for a feature, what we're identifying here is the value for a single cell.

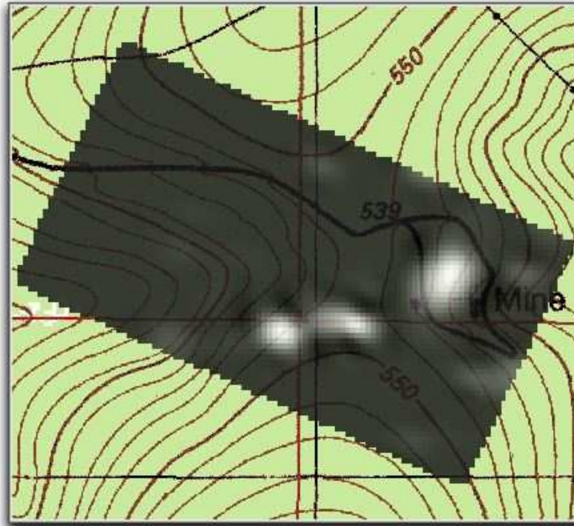


Figure 4.6: Grid of silver values

You'll also notice that although for a vector layer we can use the select tools and open the attribute table, we can't do it for the grid. The only information we can gain is the cell value.

To help Harrison with his qualitative analysis, let's identify a few cells to get a feel for the data. The dark cells to the west (left side of the map) are low values,[6] typically in the neighborhood of 0.2. If we look at the brighter areas of the grid, we find values up to 3.65 or greater. So, just by looking at the grid we can get a feel for where the higher values are, once we know the pattern.

Using the Metadata tab in the raster properties dialog box we can get some interesting information about the grid. If you scroll to the bottom of the dialog box, you'll find some statistics of interest:

Property	Value
Band	Undefined
Band No	1
Min Val	-0.2643643320
Max Val	3.6971910000

6. The units on this grid are in parts per million (ppm). Nobody is going to get rich off the silver in this grid.

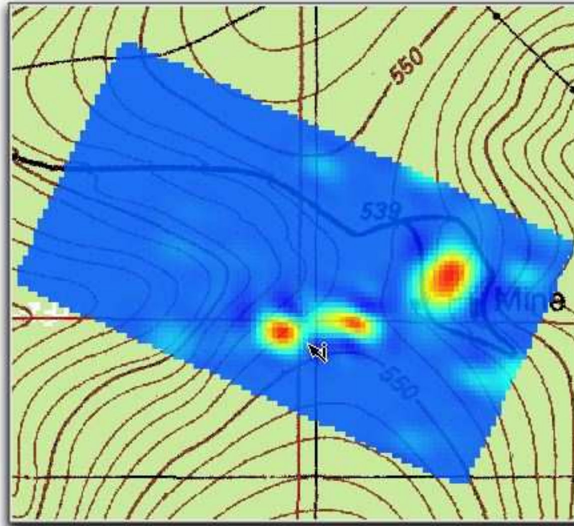


Figure 4.7: Grid of silver values in pseudocolor

Range	3.9615553319
Mean	0.3253868880
Sum of Squares	991.8232408759
Standard Deviation	0.4478514579
Sum of All Cells	1609.3635482636
Cell Count	4946

This tells us the minimum and maximum values in the grid, the number of cells, the range, and some other statistics. This is probably a better way to get a quick overview of the distribution of a grid, as opposed to randomly identifying cells.

To make the display a bit more dramatic, we can open the raster properties dialog box and change the color map from grayscale to pseudocolor. When we apply this change, we get the result shown in Figure 4.7. Now the high value areas are red and the low values are blue, making it even easier to visually analyze the distribution of the values.

We could take our analysis further by using GRASS to create a contour map of the cell values or by generating a hillshade to enhance the appearance of the map.

We'll delve into contouring and creating hillshades, as well as some other raster manipulation, when we get to Chapter 10, *Geoprocessing*, on page 149.

Now that we've learned a bit about rasters, let's take a look at digitizing some vector data using our OSGIS software.

Chapter 5

Digitizing and Editing Vector Data

One of the strengths of desktop GIS is the ability to create new data. Although your favorite desktop application can be a data consumer, it can also be a creator. In this chapter, we'll look at creating vector data and some of the reasons why you might want to do so.

5.1 Simple Digitizing

If you remember Harrison's original bird project, one of the things he wanted to do was create a new vector layer for the lakes in one of his birding areas (see Figure 1.3, on page 19). Once he had the lakes in a new vector layer, he could do some more advanced GIS processing to create a buffer and test his hypothesis regarding birds and nearness to lakes. This is a pretty simple digitizing project, one that we'll do for Harrison.

Picking a Tool

As you've gathered, we could use a bunch of tools to digitize the lakes. Since we're going for simple here, either uDig or QGIS is a good choice. You could use OpenJUMP or GRASS as well. Since we are going to do some geoprocessing with this layer (a fancy way of saying create a buffer), it makes sense to create it in a flexible format that we can import into whatever application we choose. The obvious choice is a shapefile, although we could just as easily have chosen PostGIS—but since we won't talk about that until Chapter 7, *Spatial Databases*, on page 98, we'll keep it simple.

To build Harrison's layer, we'll use QGIS and create a shapefile containing lakes as polygons.

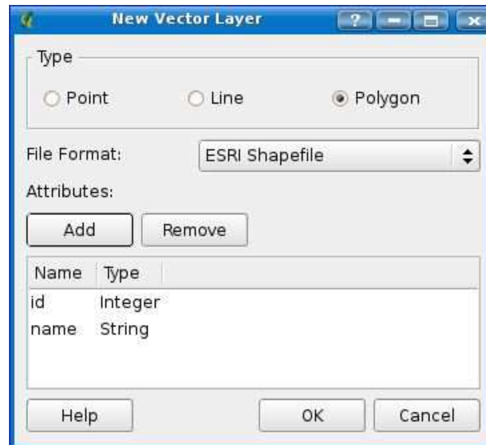


Figure 5.1: Creating a new shapefile in QGIS

Digitizing the Lakes

The first step is to fire up QGIS and add the raster we want to digitize from to create the lakes. We're using o48092d8.tiff, a DRG from the Daley Bay Quadrangle in Minnesota.[1] Once we have the raster loaded, we need to create a new vector layer for the lakes.

As of version 0.9, QGIS supports the creation of shapefiles only for editing, although you could create a new PostgreSQL layer using SQL and edit it. For now we'll create a shapefile with an id field and a name field. To do this, choose New Vector Layer from the Layer menu. In Figure 5.1, you can see the completed layer information with the fields defined. Even though there is a drop-down for file format, there is only one choice, as we mentioned before. Once we click the OK button, QGIS opens the dialog box to save the file. This allows you to navigate to the directory where you want the shapefile to live and give it an appropriate name. Having done that, our shapefile is created and displayed in QGIS as shown in Figure 5.2, on the next page. Of course, there is nothing in it yet.

Now we are ready to digitize. First we need to allow editing on the new layer by right-clicking it in the legend and choosing Allow Editing from

1. Available at http://www.archive.org/download/usgs_drg_mn_48092_d8/o48092d8.tif.

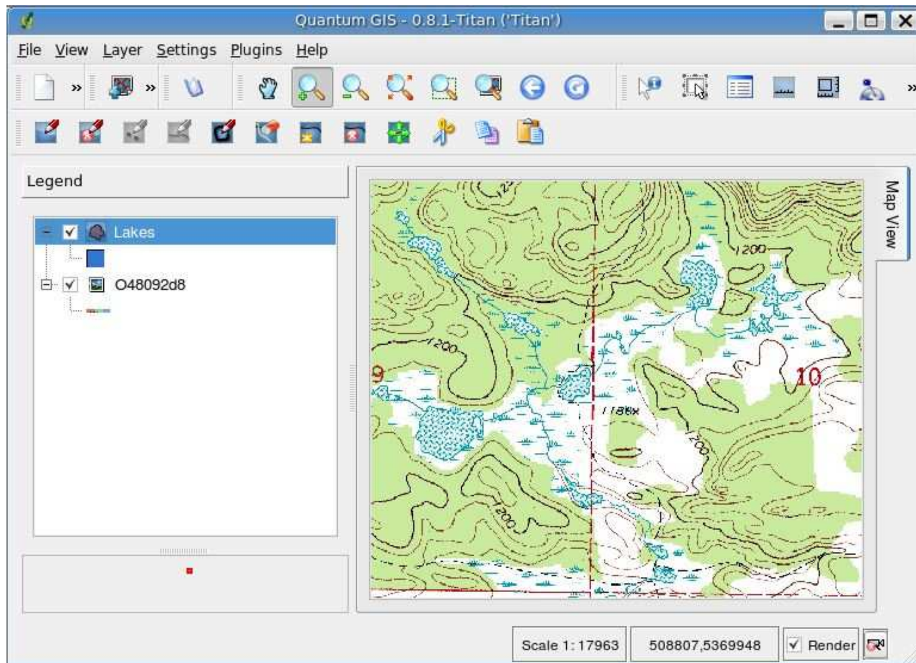


Figure 5.2: QGIS with new layer ready to edit

the pop-up menu.[2] A small pencil with a little scribble trailing from it will appear over the layer's icon in the legend to indicate that we are now in edit mode. Before we start editing, we need to make sure the digitizing toolbar is visible. If not, right-click in the toolbar area of QGIS, and choose Digitizing from the pop-up menu. In Figure 5.2, the digitizing toolbar is visible (it's on the second row of the toolbar area). You'll notice that some of the buttons are disabled (grayed out), in particular the point and line buttons. This is because QGIS knows we are editing a polygon layer and won't allow you to create the wrong feature type. In addition to creating polygons, there are tools for starting and stopping editing and editing existing features. Let's start digitizing by clicking the Capture Polygon tool.

First we will digitize the boundary of largest lake in our map view (it's the big one to the west). An important thing to remember when digitizing is to choose an appropriate scale for creating features. To get a

2. In QGIS 0.9.*x*, the Allow Editing menu option has been renamed Toggle Editing.

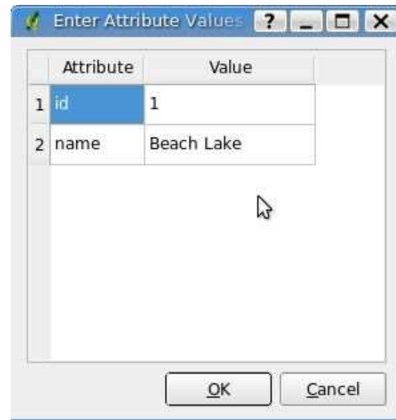


Figure 5.3: Entering attributes for a feature

gross approximation of the lake, we can just zoom in until we see only the lake. If we want a detailed representation of the lake, we need to zoom in much closer and pan around the map as we digitize. We'll take a middle-of-the-road approach here and zoom in somewhat to illustrate navigating the map while digitizing.

To start, we zoom in until the lake pretty much fills the map view. To digitize the lake, click with the left mouse button and begin moving along the shoreline, clicking at each location where there is a change in direction. If you get to the edge of the map canvas and need to pan to continue digitizing, don't use the pan tool; instead hold down the Spacebar and move the mouse to pan. If you are zoomed in really close, this technique allows you to work your way around the lake. If you find out that you need to change the zoom level to effectively digitize, you can zoom in and out using the mouse wheel. Both of these navigation techniques are discussed in Section D.3, *Map Navigation and Bookmarks*, on page 336.

You'll notice as you digitize around the lake, the display is updated with the polygon as you create it. To complete the lake, right-click at the final point. This opens the attribute dialog box where we enter the attribute values for the feature. In Figure 5.3, we have assigned an ID of 1 and entered "Beach Lake" as the name. When we click OK, these values will be saved for storage in the .dbf of our shapefile. That completes Beach Lake; however, the feature hasn't actually been saved yet. To

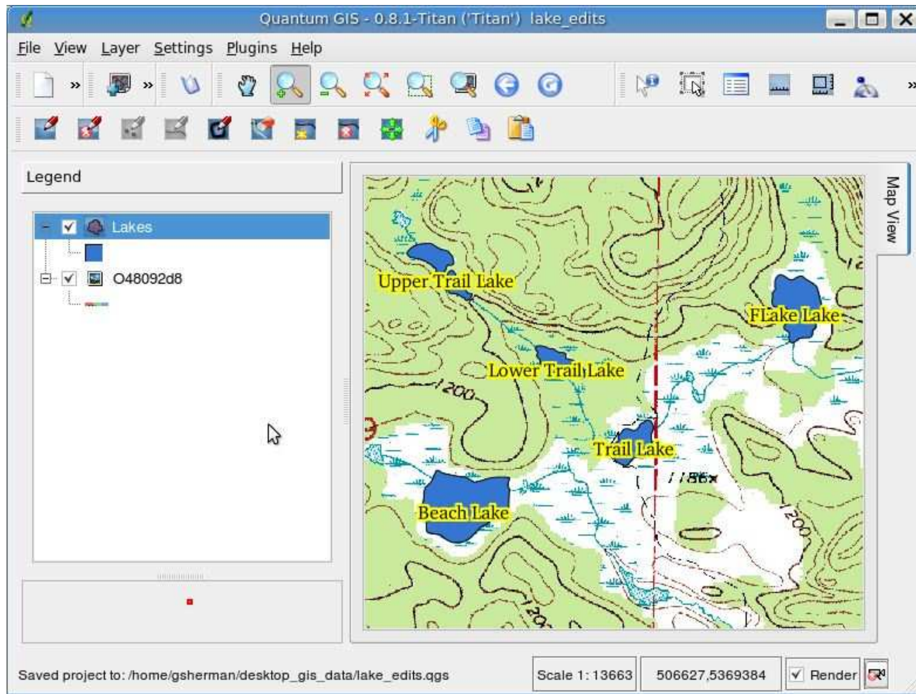


Figure 5.4: Results of digitizing lakes in QGIS

actually write our changes out, we need to stop editing by clicking the tool in the toolbar or right-clicking the layer and choosing Allow Editing. A confirmation dialog box appears, and we have to choose Yes to save the edits. Once we do that, the lake appears like a regular polygon feature, properly colored, and we can both identify it and view the attribute table. In Figure 5.4, you can see the results of our digitizing effort, with the lakes labeled using the name field in the attribute table.

Fixing Mistakes

As you digitize, you are bound to make mistakes. You might find that you didn't follow the shore quite right, made a line too straight when it should have been curved, or just totally bungled the boundary. Typically the best approach is to continue and then correct the problems after you have completed the feature.

Keeping Your Data Safe

When editing your data with any GIS application, it's a good idea to make sure you have a backup copy. Let's face it, disaster can strike, whether it be a program crash, power outage, or beverage incident. Keeping a current backup of your critical data is just good practice.

All our OSGIS desktop applications that support editing allow you to make adjustments to features by moving, deleting, and inserting vertices. Once you've completed a feature, the vertices are displayed (in QGIS they look like X's). Using the vertex tools on the editing toolbar, you can adjust the boundary to correct any errors.

Harrison decides he also needs the streams connecting his lakes digitized. Along the way, he makes a few mistakes, which we'll help him fix. In Figure 5.5, on the following page, you can see part of Harrison's fist attempt at connecting the lakes using QGIS and a line shapefile for the streams.

If you look closely at the streams (we've made them red so they stand out), you'll notice a number of problems. For one, we have a stream that doesn't connect with its neighbor. In another case, the line segment overshoots the intersection. These undershoots and overshoots are called *dangles*. Lastly, we have a stream that runs too far into the lake on the east. The other thing you'll notice is that Harrison was a bit sloppy in following the stream course, especially approaching the lake to the east.

Let's use the vertex-editing tools to clean things up a bit. All the errors can be easily corrected just by inserting new vertices where needed and moving existing vertices to intersect where they should. To do this, we first need to set the snapping tolerance. This controls how QGIS snaps to existing vertices when editing. By setting a reasonable tolerance, we can make QGIS "jump" to the closest vertex, thereby making our job easier as well as making sure the line segments actually touch. It's harder than you may think to manually move a vertex and get it exactly on the line.

To determine a proper snapping tolerance, you need to take a look at your data and get an idea of distance between vertices in your lines.

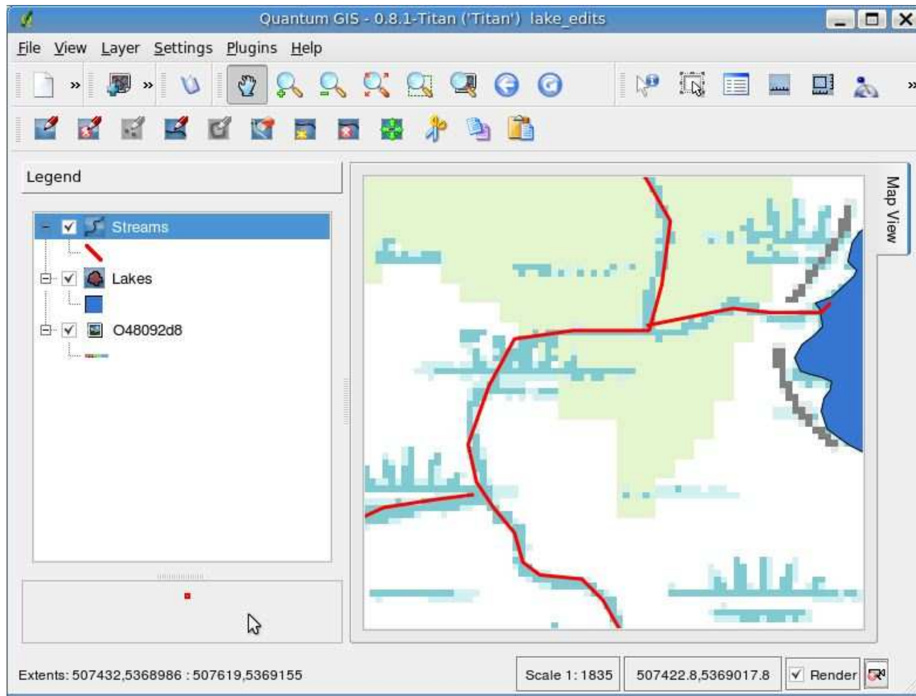


Figure 5.5: Digitized Streams

In QGIS the tolerance is set in map units, so you may find you need to experiment to get it set right. If you specify too big of a tolerance, QGIS may snap to the wrong vertex, especially if you are dealing with a large number of vertices in close proximity. Set it too small, and it won't find anything; it will pop up an annoying warning to that effect. To set the snap tolerance, open the Project Properties dialog box, click the General tab. Remember the tolerance is in map units. If you want to make sure, use the Measure Line tool to examine the distance between vertices to make an educated guess at a proper value. Since our data is in meters, a value of 2.0 seems to work well. This of course depends not only on the map units but also the scale at which you are digitizing. Fortunately, you can tweak the tolerance to get something that works.

Once the tolerance is set, we can move the vertices to fix the dangles. With the Move Vertex tool selected, place the cursor over the vertex to be moved and drag it to the new location. When you release the mouse, the vertex is moved, and the shape of the feature changes. You'll notice

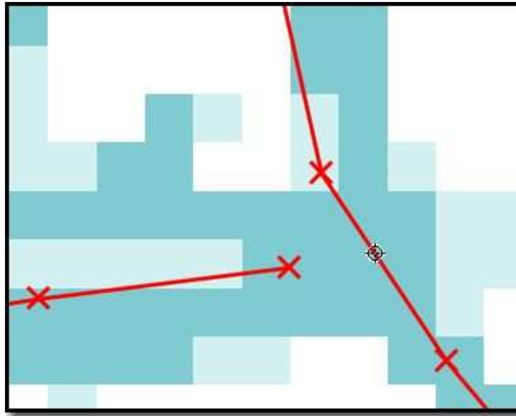


Figure 5.6: Problems with the digitized streams

that when you get close to a vertex, the snapping will kick in and pop the vertex you are moving right on top of the other. You may find that there is no vertex at the location you where your lines should meet. This is the case in Figure 5.6. To get the stream to join the other line segment, we need a new vertex at the location shown by the black, circular crosshair. We could do it without a new vertex, but then we won't get the benefit of snapping, and we'll probably just move the problem from one location to the other. In QGIS, use the Add Vertex tool to add the new vertex. Then use the Move Vertex tool to connect the dangling stream to the new vertex. You can use this technique for both overshoots and undershoots.

To clean up the sloppy work and make the streams match properly, we can add more vertices to solve the problem. First use the Add Vertex tool to add the vertices where needed by clicking the line segment that needs to be modified. Don't worry if you don't get them exact. Once you've added the vertices you need, use the Move Vertex tool to adjust the positions to reflect the path of the stream. If you didn't add enough vertices to accurately portray the feature—no problem—just add some more and move them around until the job is done.

You may find that you have too many vertices or just plain put one where it doesn't belong. In that case, use the Delete Vertex tool to get rid of the ones you don't need. When you delete a vertex, the feature reshapes itself automatically. In Figure 5.7, on the following page, you

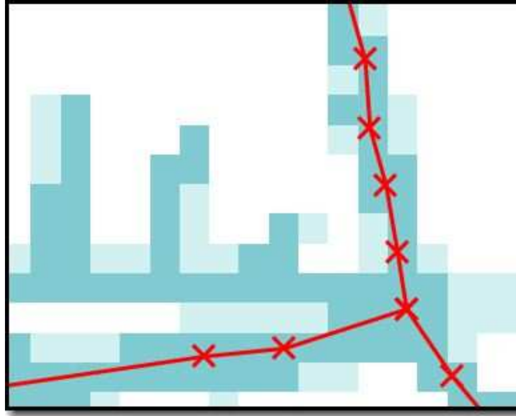


Figure 5.7: Digitized streams with corrections

can see the result of fixing a dangle and reshaping the stream to better match with the DRG. As you can see, it's pretty simple to correct mistakes in your data as you go and alter it to make it more precise when your requirements dictate.

5.2 Editing Attribute Data

Now that we have digitized the lakes and streams, everything is in good shape—except we have some problems with attributes associated with the features. If you look at Figure 5.4, on page 85, you'll notice the lake in the southeast corner of the map is named FLake Lake. Not only is there a capital *L* in "FLake," the actual name is supposed to be Fluke Lake. We can correct this error using the editing capabilities built in to the attribute table.

To fix the name of the lake, we simply open the attribute table for our lakes layer and click the Start Editing button. Once we do that, any of the items in the attribute table can be modified by clicking them and changing the value. We change the name of the lake to Fluke Lake like it's supposed to be and hit `Enter` to make the change stick. Once we're done editing the table, click the Stop Editing button to save our changes.

It was pretty easy to find the record that needed editing, since we had only five lakes in our layer. What happens if we have 5,000? In that

case, you have two options. You can use the search tool, as described in Section 3.5, *Using the Attribute Table*, on page 59, to find the desired record, or you can select the lake on the map canvas using the select tool and then float it to the top of the table using the Move Selected to Top button.

If you find that you need to make changes to a lot (or all) records in a table, then you should consider using something other than a shapefile. A spatial database may be more suited to your needs. We'll take a look at them in Chapter 7, *Spatial Databases*, on page 98.

5.3 More Digitizing and Editing

So far we've used simple digitizing and editing techniques to capture some data from Harrison's background DRG. Needless to say, there are more advanced means of digitizing and editing available to us, particularly with GRASS.

We'll look at some additional digitizing tasks in Chapter 8, *Creating Data*, on page 120. Nowadays it's often best to spend a little bit of time looking around for the data you need before you dive into a digitizing project. Oftentimes the data already exists, and you can save a lot of time by just grabbing it from the Internet and moving forward. But at times you are faced with having to create your own data. The methods we have looked at so far form a good foundation for you to launch your own projects.

Speaking of data, let's move on to a look at data formats and what you need to know when working with OSGIS desktop software.

Chapter 6

Data Formats

One of the challenges in working with GIS software, whether it be proprietary or open source, is making sense of the many data formats you encounter. Let's take a look at some of the common formats you will encounter so you can get an idea of what's out there. We'll also look at where these data formats come from, some of the conversion options, and lastly how to choose a standard data format for your mapping projects.

6.1 Common Formats

So far, we've indirectly discussed a number of formats, including shapefiles, GeoTIFF, grids, PostGIS, and GRASS vector and raster. If you are a casual or intermediate user of OSGIS software, odds are you are going to be using only a few data formats on a regular basis. Typically this means working with the following:

- Shapefiles (.shp)
- GeoTIFF or TIFF with world files (.tif, .tfw)
- JPEG with world files (.jpg, .jpw)
- GPS data (.gpx)

In fact, these are pretty common, and you can accomplish a lot with just these formats. Other vector formats you might run across during your search for data include the following:

- ArcInfo Binary Coverage
- ArcInfo Interchange File (.e00)
- MapInfo (..tab, .mid, .mif)

- SDTS, a data transfer standard for both vector and raster data[1]
- Topologically Integrated Geographic Encoding and Referencing (TIGER) data, used and distributed by the U.S. Census Bureau
- Digital Line Graphics (DLG)

There are a lot of raster formats you might encounter, including the following:

- ERMapper Compressed Wavelets (.ecw)
- Erdas Imagine (.img)
- Digital Elevation Models (.dem)
- JPEG 2000 (.jp2, .j2k)
- Multi-Resolution Seamless Image Database (MrSID) (.sid)
- GTOPO30, a global digital elevation model (DEM) derived from a number of raster and vector sources

Some of the formats are open, meaning that they have a published specification you can use to write applications and utilities that work with the format. Others are closed, requiring you to use the vendor-provided API. Of course, this is a concern only if you want to write your own applications and utilities. If you're content with using the OSGIS applications available, someone else has done the hard work for you.

Although it's not important to understand these formats to use them, it does help to know a bit about them so you can determine whether your favorite OSGIS software supports the format. In case you haven't realized it yet, the GDAL/OGR library supports a huge range of vector and raster formats—see Section A.2, *GDAL/OGR*, on page 283 for lists of additional vector and raster formats you might encounter.

The good news is if you are using OSGIS software that uses GDAL/OGR for accessing raster and vector data (such as GRASS or QGIS), then you have access to most, if not all, of the formats listed.

In Section 11.2, *Using GDAL and OGR*, on page 186, we'll look at using the GDAL/OGR utilities to get information about our data as well as convert and transform both raster and vector layers.

Web-Deliverable Data

Another "format" you'll encounter is data deliverable over the Web. This category of data is often referred to as W*S. The moniker W*S is

1. See http://mcmcweb.er.usgs.gov/sdts

attached to standards for delivering geospatial data over the Web and includes Web Mapping Service (WMS), Web Features Service (WFS), and Web Coverage Service (WCS). A good chunk of the web mapping applications you might use in your browser get some or all of their data from a W*S service.

Many of our desktop applications include support for at least WMS. This allows us to include data from across the Internet in our mapping projects. The good thing is you don't have to understand the standard or how it works; you just use it and get good data for free.

If you want to get real technical information on WMS, WFS, and WCS, you can find standards on the Open Geospatial Consortium website.[2]

6.2 Choosing a Standard Format

You might be wondering about a standard format for all your projects. This isn't strictly necessary, although it might make your life easier. Assuming your OSGIS software can handle a multitude of formats, there may be no reason to convert.

Reasons to Standardize

There are some valid reasons to standardize on a data format for your raster and vector data. In reality, you'll probably still have some data that's not in your standard format. Let's take a look at a some reasons why you might want to convert to a standard format.

Data Management

GIS data can have several unsavory characteristics—and we're not talking about accuracy or quality. As you begin to work with data, transform it, analyze it, and so forth, you'll find the following:

- It propagates rapidly.
- It grows and hides in places you never expect.
- Unchecked, it rapidly becomes unmanageable.

Some (OK, us) have gone so far as to call GIS data an illicit drug.[3]

You may be wondering how converting to a standard format will improve data management. Well, it's not a silver bullet, but it can aid in

2. http://www.opengeospatial.org
3. http://spatialgalaxy.net/2006/03/29/gis-data-is-an-illicit-drug/

creating a logical structure for storing your data. For example, if all your vector data is in shapefiles and you can create a nicely organized directory structure, be it by theme or by project, then your ability to find and use the data you need increases.

Some find that data management is improved by using a spatial database, although there are perhaps better reasons to use one. By storing your vector and/or raster data in a spatial database, you provide one point of entry for all your data needs. There is no question of which server or directory you need to search to find the data you need. We'll look at spatial databases in more detail in Chapter 7, *Spatial Databases*, on page 98.

Another example is GRASS, which uses its own format for storing both raster and vector data. Data in GRASS is organized by location and mapset, making it easy to structure your data collection in a way that can be more easily managed.

Is improved data management alone a reason for converting to a particular format? Probably not, especially if you're talking about converting between file-based formats such as shapefiles and GeoTIFFs. The chief considerations are making the data discoverable, accessible, and usable. If your formats of choice include file-based data, you should create a structured logical directory layout and naming convention and adhere to it. This will make managing your data much easier and prevent the multiplication of duplicate data sets. Another important management tool is metadata that documents each of your datasets. If you want to take the formal approach and create metadata in a format that others will understand, use the standard.[4] At the very least, you should include a text file describing the data, its origin, and the processing steps used to create it. Fortunately, the metadata standard includes all these components, so you may find it's worth using.

Improved Functionality

If you want to do more than display and edit spatial data, then conversion to gain improved functionality is certainly a worthwhile consideration. For example, although QGIS is great at displaying a wide range of raster and vector data and is even able to digitize and edit features, it lacks the ability to perform common GIS operations. Harrison's simple analysis of bird sightings using a buffer (Figure 1.3, on page 19)

4. http://fgdc.gov/metadata

requires something with geoprocessing capability. In this case, Harrison could have converted his digitized lakes (which likely began life as a shapefile) into a PostGIS or GRASS layer. Both of these give him the functionality he needs to create a buffer. In the case of PostGIS, it's done with a fairly simple SQL query. In GRASS, he can use the v.buffer command or the buffer module in the QGIS-GRASS toolbox.

PostGIS is a good example of a reason to convert. Not only does it improve data management by giving you a "portal" to your data, but it has been certified as OGC compliant and provides the spatial functions specified in the Simple Features Specification for SQL. This means that not only can we display and edit PostGIS data in an application-like QGIS, we also get a whole batch of spatial functions that we can use to query the relationships between features, transform between projections, and create new features. If you find that your work requires more than just simple viewing and editing, then conversion is worth considering.

When software is certified as OGC compliant, you can be assured that it adheres to established standards and can interoperate with other compliant software.

Enhanced GIS Capabilities

The other reason to convert is to gain enhanced GIS functionality. You're probably asking what the difference is between this and the improved functionality aspect we just covered. You can view it as a progression to more powerful and perhaps complex GIS operations. We're pretty much talking about GRASS and its bountiful assortment of vector- and raster-processing tools. Since GRASS stores data in its own format, we need to convert our existing data in order to take advantage of the tools.

Examples of the type of capabilities we're talking about include the following:

- Line-of-sight analysis
- Union and intersection of layers to create a new layer
- Merging raster data
- Mathematical operations on grids
- Contouring

We'll dive into some of these topics later in Chapter 10, *Geoprocessing*, on page 149.

6.3 Conversion Options

So, you decide to convert some or all your data to a new format. The next question is, what tools are available to do the job? Fortunately, in the OSGIS world, conversion between formats is not only commonplace but easy as well.

If you choose to migrate all your data to PostGIS or GRASS, it's not a problem. Both provide the routines to import your data and export it should the need arise.

GRASS Conversion

GRASS provides both vector and raster import/export functions for a nice range of formats. To give you an idea of the capabilities, here is a partial list of the import commands and formats supported by GRASS:

r.in.arc
: Converts an ESRI ARC/INFO ASCII raster file (GRID) into a (binary) raster map layer

r.in.ascii
: Converts ASCII raster file to binary raster map layer

r.in.aster
: Imports, georeferences, and rectifies an ASTER image

r.in.gdal
: Imports a GDAL-supported raster file into a binary raster map layer

r.in.srtm
: Imports Shuttle Radar Topography Mission (SRTM) .hgt files into GRASS

r.in.wms
: Downloads and imports data from WMS servers

v.in.ascii
: Creates a vector map from ASCII points file or ASCII vector file

v.in.db
: Creates new vector map (point layer) from database table containing coordinates

v.in.dxf
: Converts AutoCad DXF files to GRASS format

v.in.e00
: Imports an ArcInfo export file .e00 to GRASS format

v.in.garmin
: Downloads waypoints, routes, and tracks from a Garmin GPS receiver into a vector map

v.in.gpsbabel
: Downloads waypoints, routes, and tracks from a GPS receiver or a GPS ASCII file into a vector map using formats supported by gpsbabel

v.in.ogr
: Converts OGR-supported formats into a GRASS vector map

You can see from the list of commands that there are a lot of options for getting your data into GRASS. In fact, we didn't list all of them for you, just some of the major ones. For exporting data out of GRASS, there are also a lot of options. We won't list them here, but in case you're curious, the commands are all of the form r.out.* for rasters and v.out.* for vectors.

PostGIS Conversion

If you choose PostGIS, it supports the loading of data using SQL and the importing/exporting of shapefiles using shp2pgsql and pgsql2shp. We'll take a look at these two utilities in Section 11.4, *PostGIS*, on page 203. If your source data isn't in shapefile format, you can still import it—you just need a little extra power. In this case, you need to use the Swiss Army knife of conversion tools.

The Swiss Army Knife

Just as you wouldn't go out into the wilderness without your Swiss Army knife (or maybe bug spray), venturing into the world of data conversion without the GDAL/OGR utilities is not advised. These utilities provide conversion between file-based vector and raster formats, as well as spatial databases.

We take an in-depth look at these tools in Chapter 11, *Using Command-Line Tools*, on page 174. For now just keep these two commands in the back of your mind: ogr2ogr and gdal_translate.

Chapter 7

Spatial Databases

7.1 Introduction

In this chapter, we take a look at spatial databases. A spatial database allows us to store features, display them, or perform geoprocessing and analysis through a rich set of spatial functions. Some of the advantages of storing data in a spatial database are as follows:

- Attributes and geometry of features are stored together.
- Spatial indexing makes drawing faster at larger scales.
- Spatial queries provide the ability to explore features and their relationships.
- You get better data management.

Structure of a Spatial Database

A spatial database is nothing more than a regular database with support for geometry data types. It typically contains functions to manipulate the geometries and perform spatial queries.

In a spatial database, a table represents a layer, a row is a feature, and a spatial column contains the geometry of the feature.

Joe Asks...

What Is a Spatial Query?

A spatial query is one that involves features and their relationship to one another. For example, assuming we had the appropriate data in our database, we might ask "Give me the names of all the coffee shops within 10 kilometers of my house." A spatial database is well suited to that type of query and can easily answer that question. Another simple example is finding all the eagle nests within a drainage basin. Of course, you can do much more complex things with spatial queries including transforming and creating new data, as well as projecting data on the fly.

7.2 Open Source Spatial Databases

In the OSGIS world there are currently two options for spatially enabled databases: PostgreSQL[1] with PostGIS[2] and MySQL.[3] Of the two, PostgreSQL/PostGIS is the most mature and feature rich. MySQL has just recently added basic support for geometries. Although the MySQL implementation contains many of the OGC spatial functions, not all of them are implemented according to the specification. If you just want to store spatial features, MySQL may be the database for you. If you want to use the database to do spatial processing and queries, PostgreSQL with PostGIS is the best choice. Support for displaying features stored in MySQL is just now emerging, so your options may be limited in that regard as well.

There are other efforts in the spatial database realm, but none is to the point that it can be used by the GIS enthusiast.

Comparison of Open Source Spatial Databases

Both PostGIS and MySQL implement the Open GIS Consortium's OpenGIS Simple Features Specification for SQL (OGC). You can find the specification on the OGC website at http://www.opengeospatial.org.

1. http://postgresql.org
2. http://postgis.refractions.net
3. http://mysql.org

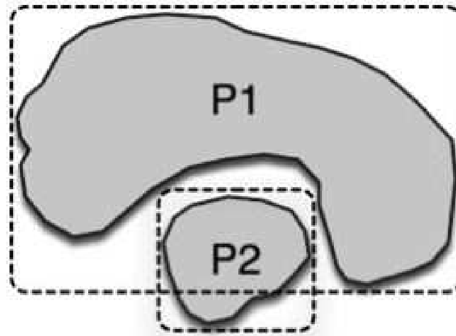


Figure 7.1: Polygons with overlapping MBRs

The standard is more fully implemented in PostgreSQL/PostGIS than MySQL. In fact, the PostGIS implementation has been certified by the OGC as compliant with version 1.1 of the standard. What does this mean to you? It means that PostGIS provides a complete and robust implementation of the standard, along with additional features not in the specification. Since PostGIS has been around longer than the MySQL spatial implementation, more desktop and web mapping clients/ servers support it.

As we said before, MySQL doesn't fully implement the OGC specification, especially when it comes to spatial functions. This means that although you can use a function, the result will be less than accurate. For example, if you want to determine whether two polygons overlap, MySQL uses a simple bounding box comparison. Depending on the shape of the polygons, they may or may not overlap even though their bounding boxes do.

In Figure 7.1, you can see two polygons that do not overlap. For the sake of example, let's assume they are lakes. In the figure you can easily see that polygon P1 and polygon P2 are spatially distinct. The dashed rectangle around each polygon is the minimum bounding rectangle (MBR).[4] Note that the MBRs do overlap. Since MySQL does only MBR comparisons, it would say that P1 and P2 overlap.

4. You'll see MBR also referred to as *extent*, *bounding box*, or *BBOX*. Some software prefers one term over another. We use MBR here because it is the term used in the MySQL documentation.

Another impact of using MBRs to compare features takes place when doing a selection in our favorite desktop GIS application. Suppose we want to select P2 in Figure 7.1, on the previous page. We would typically choose the Select tool and either point at or draw a small rectangle inside P2 to select it. You can guess what will happen if MBRs are used to determine what should be selected. Since our selection rectangle may fall within the MBR of both P1 and P2, we could end up with both selected (and usually highlighted). This isn't really desirable and can often be confusing. Although MySQL provides storage and MBR functions, it isn't really suited for applications that depend on the correct functioning of spatial functions for selecting and identifying features.

To be fair, the MySQL developers freely admit the nature of their implementation and also imply that they may support the OGC specification fully in future releases. Support for MySQL spatial data is already available in OGR and will likely show up in other OSGIS applications soon.

7.3 Getting Started with PostGIS

In this section, we'll look at how to enable PostGIS in your PostgreSQL database and load some data from shapefiles and perhaps from other sources. We assume you already have a working PostgreSQL install. If not, refer to the installation section of the manual[5] where you will find detailed instructions for getting PostgreSQL up and running on your platform. If you're lucky, PostgreSQL is already installed, and you are ready to proceed with getting PostGIS set up.

Getting PostGIS can be easy if you are running the right platform. You may find a binary version available for download.[6] Otherwise, you will have to build from source. If you are running Linux, be sure to check for a PostGIS binary using your package management tool. Many distributions include PostgreSQL and PostGIS, making it easy to get started. If you are running Windows, the latest PostgreSQL installers include an option to install PostGIS. It's not selected by default, so make sure to you include it when choosing options during the install. Once you have the software in place, you're ready to set up a database and add the PostGIS extension and tables.

5. http://postgresql.org
6. http://postgis.refractions.net

Creating a PostGIS-Enabled Database

Since PostGIS is an extension to PostgreSQL, you have to add it to a database in order to use the geometry types and functions. Let's look at a session that creates a new database and enables PostGIS:

```
$ createdb -E UTF8 desktop_data
CREATE DATABASE
$ createlang plpgsql desktop_data
$ psql desktop_data
Welcome to psql 8.1.3, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
\h for help with SQL commands
\? for help with psql commands
\g or terminate with semicolon to execute query
\q to quit

desktop_data=#  \i ./lwpostgis.sql
BEGIN
psql:./lwpostgis.sql:39: NOTICE:  type "histogram2d" is not yet defined
DETAIL:  Creating a shell type definition.
CREATE FUNCTION
...
CREATE FUNCTION
COMMIT
desktop_data=#  \i spatial_ref_sys.sql
INSERT 0 1
...
INSERT 0 1
COMMIT
VACUUM
desktop_data=# \d
              List of relations
 Schema |       Name        | Type  |  Owner
--------+-------------------+-------+----------
 public | geometry_columns  | table | gsherman
 public | spatial_ref_sys   | table | gsherman
(2 rows)

desktop_data=#
```

Let's breakdown what we did and explain the steps. First we created a database using the createdb command, specifying an encoding type of UTF8 (Unicode) using the -E switch. Using Unicode for the database encoding provides us with the most flexible solution, especially when storing non-ASCII data. Once the database is created, we add the PostgreSQL procedural language (PL/pgSQL) to the database using the createlang command and the plpgsql keyword. PostGIS needs PL/pgSQL in order to implement its spatial types and functions.

PostGIS and Templates

There is an easier way to create additional PostGIS-enabled databases. When PostgreSQL creates a database, it does it by copying an existing database. Usually this is the standard system database template1. Anything in the template database ends up in your newly created database. We can use this capability to create a PostGIS template database that can be used when creating a new PostGIS-enabled database.

To create a template, simply create an empty database from the standard template using createdb -E UTF8 postgis_template from the command line. Then follow the example in this chapter to load the lwpostgis.sql and spatial_ref_sys.sql scripts. Once you have the template, use createdb -E UTF8 -T postgis_template myNewDb to create a new PostGIS-enabled database.

If you installed PostgreSQL on Windows with the PostGIS option, it should have created a postgis_template database for you. In this case, you are ready to start creating your own PostGIS-enabled databases.

Now that we have a database set up and properly configured, the next step is to load the PostGIS extension into our database. The commands to do this are provided with PostGIS in the lwpostgis.sql file. We simply execute this SQL in our newly created database. There are a number of ways to do this (for example, from a database client tool such as PgAdminIII); however, we chose to use the PostgreSQL interactive terminal psql. In psql, we use the \i command to read the file from disk and execute the SQL statements. In our example, we had changed to the directory containing lwpostgis.sql. If we hadn't, the full path to lwpostgis.sql would be required. This creates the types and functions. At this point we have a PostGIS-enabled database, but we aren't done yet.

The final step is to create the spatial references table that contains more than 2,600 coordinate systems. To do this, we executed the spatial_ref_sys.sql file, also provided with PostGIS. Our database is now ready to use for PostGIS data. Using the \d command in psql gives us a list of the tables in our new database. In addition to the spatial_ref_sys table, you'll notice the geometry_columns table. Let's look at it in a bit more detail.

The geometry_columns Table

The geometry_columns table describes the spatially enabled tables in your database. Many application programs rely on the records in geometry_columns to determine which tables are spatial tables. This table, as well as spatial_ref_sys, is described in the OpenGIS Simple Features Specification for SQL.[7] We can view the structure of the geometry_columns table using the \d command in the psql interactive terminal:

```
desktop_data=# \d geometry_columns
            Table "public.geometry_columns"
      Column       |          Type          | Modifiers
-------------------+------------------------+-----------
 f_table_catalog   | character varying(256) | not null
 f_table_schema    | character varying(256) | not null
 f_table_name      | character varying(256) | not null
 f_geometry_column | character varying(256) | not null
 coord_dimension   | integer                | not null
 srid              | integer                | not null
 type              | character varying(30)  | not null
Indexes:
  "geometry_columns_pk" PRIMARY KEY, btree (f_table_catalog, f_table_schema,
      f_table_name, f_geometry_column)
```

The first three columns provide a fully qualified name for a table. Some databases may use "catalog," and since the geometry_columns table is a standard, it is included in every OGC-compliant implementation. PostgreSQL doesn't use the concept of a "catalog," so this column will always be blank. By default, all tables in PostgreSQL are placed in the "public" schema. So for a PostGIS-enabled table, we will find the f_table_schema and f_table_name populated with "public" and the table name, respectively.

The f_geometry_column contains the name of the column in your spatial table that contains the geometry. The coord_dimension contains the dimension of the features (2, 3, or 4). The srid is the spatial reference ID column and contains a number that is related to the srid column in the spatial_ref_sys table. This defines the coordinate system for the table. The type field contains information about the feature type contained in the table and contains a keyword such as POINT, LINESTRING, POLYGON, and the MULTI forms of each feature.

With this information, a client application (in other words, your desktop GIS) can quickly determine which spatially enabled tables are available

7. http://www.opengeospatial.org

and also collect the information needed to load, project, and display the features in a table. How does the geometry_columns table get populated? There are several ways:

- When loading data using shp2pgsql, QGIS, or ogr2ogr, a record is inserted into the geometry_columns table.
- Using the AddGeometryColumn function on a table that does not already have a spatial column. This creates the column in the table and also inserts a record into geometry_columns.
- Manually inserting a record using a SQL insert statement.

Typically you use the AddGeometryColumn function when you create a new table and want to add a geometry column. In this case, you create the table using SQL *without* the geometry column and then use the function to both add the column and create an entry in the geometry_columns table.

```
desktop_data=# CREATE TABLE lakes(LAKE_ID int4, LAKE_NAME varchar(32),
  LAKE_DEPTH float);
CREATE TABLE
desktop_data=# SELECT AddGeometryColumn('public', 'lakes', 'the_geom', 4326,
  'POLYGON', 2 );
                    addgeometrycolumn
-----------------------------------------------------
 public.lakes.the_geom SRID:4326 TYPE:POLYGON DIMS:2

(1 row)
```

A final note on the geometry_columns table: Some software such as QGIS can search your PostgreSQL database and determine which tables are spatially enabled. To maintain maximum flexibility in your database, you should probably ensure that each spatial table has an entry in the geometry_columns table.

Spatial Index

When working with PostGIS, it's important to make sure you have a spatial index for each layer. Having an index improves both spatial query and rendering performance.

PostGIS provides a Generalized Search Tree (GiST) index for spatial features. Depending on how you created your layers, the index may already exist. You can easily check to see whether an index exists for a layer using psql to examine the properties of a layer.

```
$ psql gis_data
Welcome to psql 8.2.4, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

gis_data=# \d parklands
           Table "public.parklands"
  Column  |         Type          | Modifiers
----------+-----------------------+-----------
 id       | integer               | not null
 name     | character varying(10) |
 the_geom | geometry              |
Indexes:
    "parklands_pkey" PRIMARY KEY, btree (id)
    "sidx_parklands" gist (the_geom)
Check constraints:
    "enforce_dims_the_geom" CHECK (ndims(the_geom) = 2)
    "enforce_geotype_the_geom" CHECK (geometrytype(the_geom) =
       'POLYGON'::text OR the_geom IS NULL)
    "enforce_srid_the_geom" CHECK (srid(the_geom) = 4326)
```

We use the \d command to list the properties of the parklands table. Notice under the indexes heading there is a primary key on the id field and, under that, a GiST index on the geometry column the_geom. If you don't see an entry for a GiST index in the list, you should create one for your table. To create a GiST index for your table, use the following SQL:

```
CREATE INDEX sidx_parklands on parklands USING GIST (the_geom GIST_GEOMETRY_OPS);
```

If you import shapefiles using SPIT, a GiST index will not be created. Using shp2pgsql allows you specify the creation of a GiST index during the import of the shapefile. See Section 11.4, *Importing Shapefiles*, on page 203 for an example.

Loading PostGIS Data

There are a number of ways to load data into a PostGIS-enabled database. In Chapter 11, *Using Command-Line Tools*, on page 174, you will see how to load data using both the OGR and PostGIS command-line utilities. For now we'll look at two other methods for loading data: SQL and QGIS.

Using SQL to Load Data

To load spatial data using SQL, use the GeomFromText function. This function is part of the OGC specification and as an argument takes the Well-Known Text (WKT) representation of a feature. WKT is a simple way to specify a feature type and its coordinates. Some examples of WKT representations are as follows:

- POINT(-151.5 61.5)
- LINESTRING(-151.5 61.5, -151.5 62.5, -152.25 63.0)
- POLYGON((-155.82 57.31,-155.94 61.18,-152.82 61.18,-152.78 57.31,-155.82 57.31))

To create a polygon in the lakes table created earlier, you would use an insert statement as follows:

```
insert into lakes values(1, 'Big Lake', 127.6,
  GeomFromText('POLYGON((-155.82 57.31,-155.94 61.18,
    -152.82 61.18,-152.78 57.31,-155.82 57.31))', 4326));
```

This creates a lake (a pretty rectangular one) named Big Lake with an ID of 1 and a depth of 127.6. The feature type is a polygon, and the spatial reference ID is 4326.

It makes sense to use this method of loading data when you want to import data using a script from a text file or in application code where you are taking input from a user or device. In normal practice, it can become quite cumbersome to manually enter WKT to build up a SQL statement. At least now you are aware of the capability, but you don't need to know this technique to load and use PostGIS data in your desktop GIS application.

Using QGIS to Load Data

QGIS comes with a plugin called SPIT, which stands for Shapefile to PostGIS Import Tool. This plugin allows you to import shapefiles into PostGIS from within QGIS. To use SPIT, make sure it's loaded from the Plugins menu. Once loaded, it will appear as a blue elephant icon on the Plugins toolbar. Before you can use SPIT, you need to have already created a connection to PostGIS (this isn't strictly true, but it makes things easier). We haven't talked about creating connections yet, but you can learn how by jumping ahead to Section 7.4, *Using PostGIS and Quantum GIS*, on page 110. Assuming you have a working connection, just click the SPIT icon to open the tool. In Figure 7.2, on the next page, you can see the tool with a few shapefiles ready to be loaded into PostGIS.

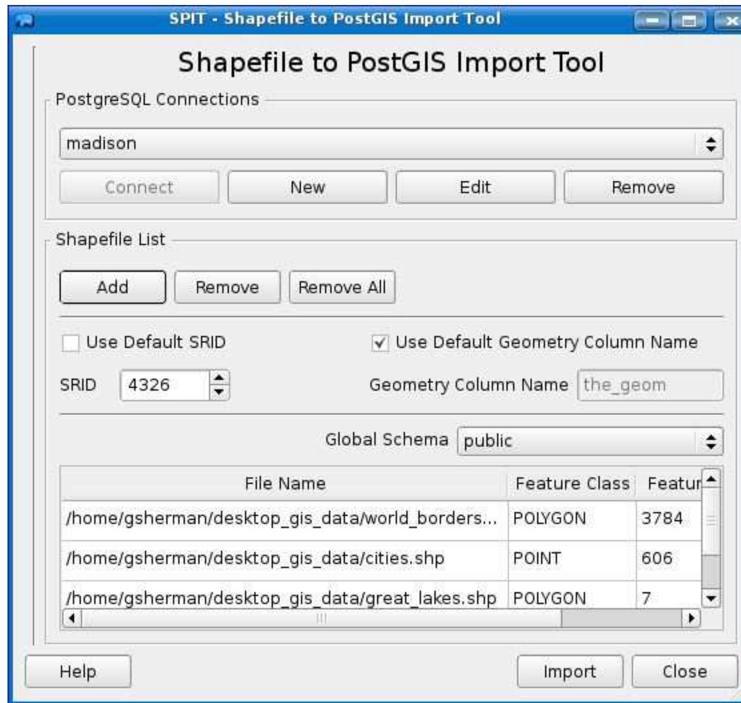


Figure 7.2: Loading shapefiles into PostGIS using SPIT

Let's take a look at each of the items SPIT requires:

PostgreSQL Connections
: The drop-down box lists all the connections you have defined. You can also create a new connection if you don't have one already. You can also edit the connection selected in the drop-down if things aren't quite right. Note that you don't explicitly have to connect. When SPIT starts up, it automatically connects to the database in the drop-down list.

Shapefile List
: This panel provides three buttons to manage the list of shapefiles to be loaded. The list is displayed at the bottom of the dialog box. You can add one or more shapefiles by clicking the Add button and selecting the file(s) from the file dialog box. You can remove a single shapefile from the list by clicking it to select it and then using the Remove button. The Remove All button does as it says and empties the list of shapefiles.

Use Default SRID

Click this checkbox to use the default SRID of -1. This is usually a bad thing because none of your geometries will be associated with a coordinate system as defined in the spatial_ref_sys table. It is better to uncheck the box and enter the spatial reference ID in the box.

Use Default Geometry Column Name

If this box is checked, the default geometry column name of "the_geom" will be used when creating tables. If you want to use a different name, uncheck the box, and enter the name you desire.

Global Schema

This drop-down box lists all schemas in your database. If you want to create your new spatial tables in a schema other than "public," select it from the list.

File List

The file list contains all the shapefiles you have selected for loading. You can edit the feature class and schema for each shapefile entry by clicking the text or choosing the schema from the drop-down list. Editing the feature class type can cause your import to fail but may be needed in some circumstances. The file list also shows the number of features in the shapefile and the name that will be used to create the table.

Once you click the Import button, SPIT proceeds to process each file. A progress bar displays the status as the import proceeds. As the files are processed, they are removed from the list.

Although SPIT is a handy tool, it is also somewhat picky. You may find that, depending on the feature type, some shapefiles can't be loaded. For the fail-safe loading of shapefiles into PostGIS, use one of the methods described in Chapter 11, *Using Command-Line Tools*, on page 174.

Spatial Queries

Let's look at one last feature of spatial databases before we move on to viewing data stored in PostGIS. One of the strengths of an OGC-compliant database is the ability to do spatial queries. PostGIS provides a wealth of both OGC and custom functions to perform queries based on spatial relationships. Using SQL, we can find features that overlap, intersect, touch, or are contained in/by another feature. We can also transform coordinates on the fly, reprojecting them from one spatial reference system to another.

Let's look at one simple example to illustrate. Suppose someone says "I live at latitude 18N, longitude 77W." We want to know where that is—what are our options?

We can start up our desktop GIS, load up a world country layer, and move our mouse around to find the location. Or if we have the data in PostGIS, we can quickly do a spatial query to determine the location:

```
desktop_data=# select cntry_name, pop_cntry from world_borders
  where GeomFromText('POINT(-77 18)',4326) && the_geom;

 cntry_name | pop_cntry
------------+-----------
 Jamaica    |   2713130
(1 row)
```

The query uses the OGC function GeomFromText to create a temporary point object to use in the search. We use the && operator to test whether the bounding boxes of the features (our point and all polygons in the world) intersect. The query returns the results, in this case Jamaica. This is a simple example of the power of queries using a spatial database. The output isn't a map, and we didn't even use a GUI to answer the question.

For details on using spatial functions and geometry constructors, see the nicely detailed PostGIS manual.[8] Although you may think that using these functions isn't a "desktop GIS" activity, it is an important part of data preparation, conversion, and analysis, so it pays to check out the features and capabilities.

7.4 Using PostGIS and Quantum GIS

QGIS and PostGIS have a long history—well, at least from the QGIS side. The first working version of QGIS supported only one data type—PostGIS. So, support for PostGIS has been included in QGIS from day one. This means that the implementation is fairly complete and an important part of the development and maintenance process.

8. http://postgis.refractions.net/documentation

Figure 7.3: Creating a new PostGIS connection in QGIS

Typical use of PostGIS layers goes something like this:

1. Open the PostGIS dialog box by clicking the Add a PostGIS Layer tool.
2. Select the connection to use—if one doesn't already exist, create a PostGIS connection to your database.
3. Connect to the database.
4. Select the layer(s) you want to add to the map.
5. Optionally specify a query to limit the features returned.
6. Optionally set the encoding.
7. Click the Add button to add the layer(s) to the map canvas.

We'll go through these steps one by one. Of course, by this point, we assume you have loaded some data into a PostGIS database. If not, see Section 7.3, *Loading PostGIS Data*, on page 106 for information on loading up your spatial database. To begin, let's open the PostGIS dialog box and create a new connection. Click the New button to open the Create a New PostGIS Connection dialog box. In Figure 7.3, you can see the completed dialog box for creating a new connection. Let's take a look at the required fields.

Name
: A descriptive name for the connection. This should be unique enough so you can recall at a glance the database it uses.

Host
: The host name where the database resides. This can just be localhost if you are running QGIS on the same machines as PostgreSQL/PostGIS.

Database
: The name of the PostgreSQL database to which you want to connect.

Port
: The port number on which the database listens. This is filled in with the default value when you open the dialog box, and you do not need to change it unless your database is listening on a different port.

Username
: The username used to connect to the database.

Password
: The password for the database user. QGIS supports a blank password if the PostgreSQL database is configured to support trusted connections from your machine.

Save Password
: This saves the password along with the rest of the connection information. Depending on your computing environment, this may be a security risk. If you don't store the password, QGIS will prompt you for it at connection time.

Only Look in the geometry_columns Table
: Clicking this prevents QGIS from looking through all your tables to see whether they contain a geometry column. This can speed up displaying the list of layers to choose from if you always have an entry in geometry_columns for every spatial layer.

Only Look in the 'public' Schema
: This constrains QGIS to only look in the public schema when searching for spatially enabled tables.

In Figure 7.3, on the previous page, we are running QGIS on the same machine as the database, so we specified "localhost." Our database is named "gis_data," and we are using the standard PostgreSQL port. Once you have filled in the connection information, you can use the Test Connect button to test the connection. If it fails, check the parameters again. If they are correct, you may have to check the PostgreSQL

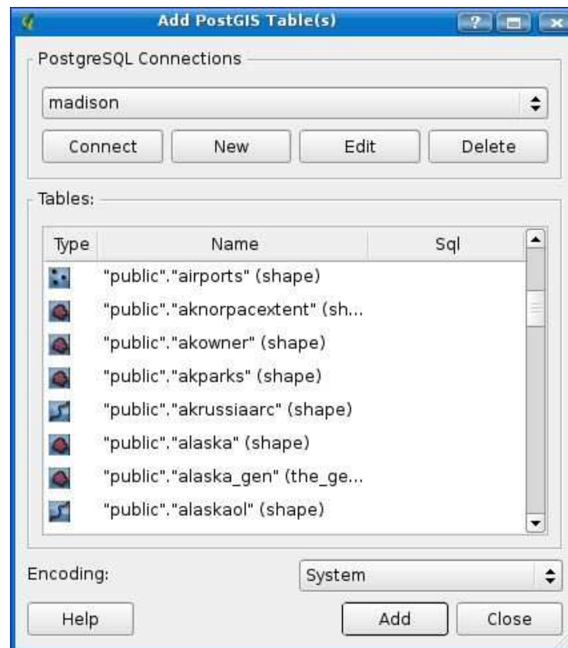


Figure 7.4: List of available PostGIS layers

database access configuration to make sure you have privileges to connect. Once you can connect, just click OK to save the connection. This takes you back to the Add PostGIS Table(s) dialog box.

We are now ready to connect to the database—with our new connection selected in the drop-down list, just click the Connect button. Once you do this, the list of available layers is populated as shown in Figure 7.4.

If we look at the list of tables, we see a representative collection of PostGIS data layers in our database. Under the Type column, you will notice an icon that indicates the feature type stored in the table. These can be point, multipoint, linestring, multilinestring, polygon, or multipolygon. You can't distinguish from the icon whether a given feature is a regular or "multi" type feature.

The Name column shows the name of the layer in the following format:

```
schema.table (geometry column name)
```

So, for example, the alaska layer is a polygon layer in the "public" schema and has its geometry stored in a column named shape. You'll

Joe Asks...

What's the Difference Between a Table and a Layer?

In our discussion, a PostGIS layer is a table in PostgreSQL that has a geometry column. It may or may not have a record describing it in the geometry_columns table.

So, layers are also tables, and you may find us referring to them in both ways. A PostgreSQL table without a geometry column is just that—a table in the database.

notice there is a "Sql" column in the layer list. This will be initially be empty, and we'll see its purpose in just a minute.

Loading a layer from the list is easy. Just select one or more by clicking them (they will be highlighted as you click so you know they are selected); then click the Add button. The layers will be added to the QGIS map canvas and drawn using a random color. Once loaded, you can modify the colors and rendering using the symbology options we discussed in Section 3.4, *Advanced Viewing and Rendering*, on page 45.

Suppose we have a PostGIS layer with 10 million features. As you can imagine, it would take a while to draw, moving all the data across the network. Or consider a layer that contains thousands of features, but we are interested only in some of them based on one of their attributes. This is where the ability to limit the features in a PostGIS layer comes in handy. You could think of these as "virtual layers" since they are defined by a query at the time you add them to QGIS. Let's look at an example using the Geographic Names Information System (GNIS) available from the U.S. Geological Survey.[9]

GNIS contains information about geographic features, including the "official" name. For example, the data includes lakes, streams, islands, glaciers, towns, and schools. All the features are represented by a single point. We'll use the GNIS data for Alaska in our example and add several "virtual layers" based on queries against the gnis table.

9. http://geonames.usgs.gov

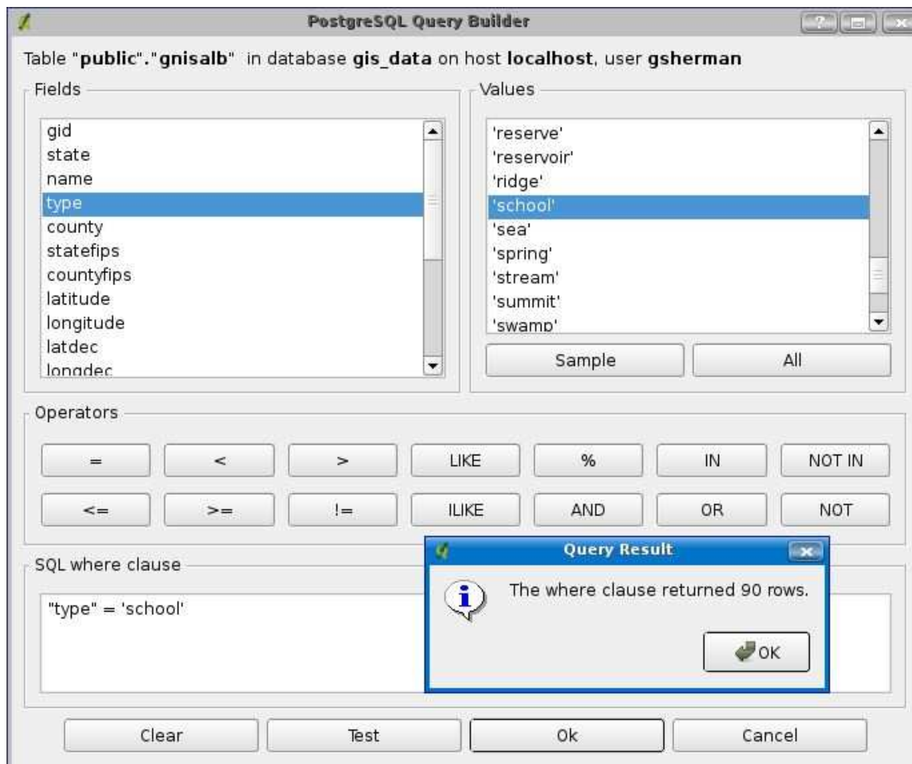


Figure 7.5: PostGIS query builder in QGIS

First we open the Add PostGIS Table(s) dialog box, connect to our database, and scroll through the list of layers until we find the gnis layer. Instead of clicking it and adding it to the map, we double-click to open the PostgreSQL Query Builder. You'll notice a strong similarity between this and the query builder we used in Section 3.5, *Advanced Search*, on page 62. In fact, they share common roots and function in essentially the same fashion. There is a slight difference in the operators available, but otherwise once you know how to use one, you can easily navigate the other. The difference of course is that now we are querying a real database instead of a shapefile. In Figure 7.5, you can see the query builder populated with the parameters for our first layer (schools) and the results of clicking the Test button.

The query we executed to create schools returned 90 rows. Once we click OK in the query builder, we are returned to the Add PostGIS Table(s) dialog box.

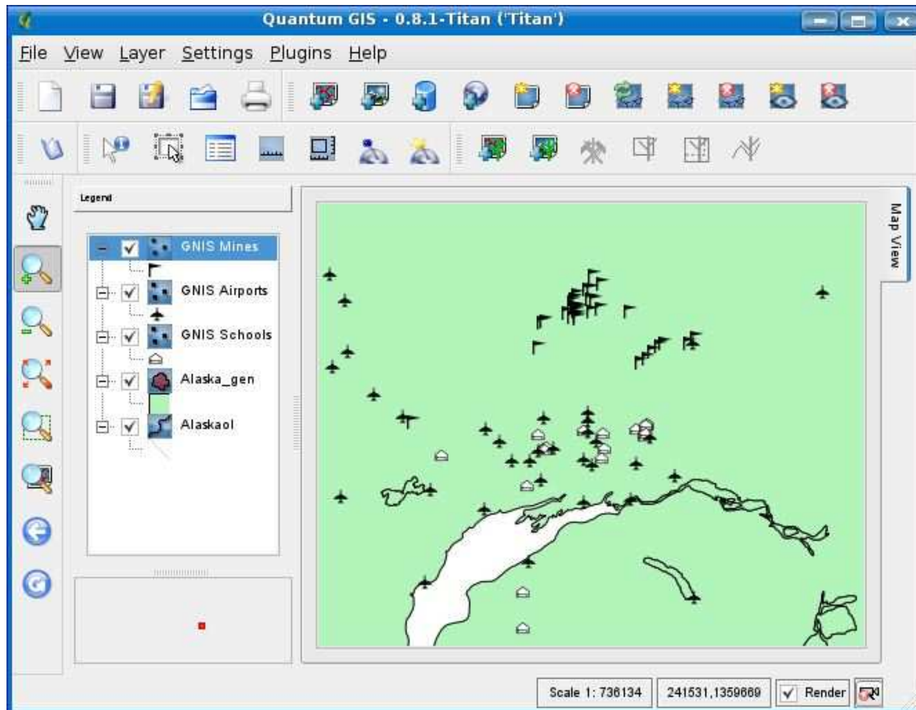


Figure 7.6: PostGIS layers created with the query builder

Note that now there is something in the "Sql" column next to the gnis layer. This is just the contents of the query box but serves to remind us what we are adding in the event that we set up queries and add more than one layer at a time. With the gnis layer selected, we click the Add button to add it to the map. In Figure 7.6, you can see that in addition to the school layer, we added layers for airports and mines. QGIS doesn't provide a very pleasing name in the legend when adding layers in this way, so we took the liberty of renaming each of the GNIS layers to something sane. So, now we have a map with three separate layers, all derived from the gnis layer in our database.

Now maybe you are asking yourself, why not just add the gnis layer and symbolize it based on type? We could do that, and it might work assuming the following:

- Our data is not too dense.
- We want to see all types, not just schools, airports, and mines.
- Our layer isn't so large that it causes performance problems.

In the case of the GNIS data, symbolizing all of it by type would result in a blob of dots and colors, with lots of overlap. Using the same PostGIS layer to create our "virtual" layers turns out to be an efficient way to get just the data we want out of a large dataset. Though we didn't demonstrate it, your queries to create a layer can be more complex than just a simple this='that' query by using operators such as **AND** and **OR** to select rows on more than one condition.

Before we leave this topic, we should mention one more thing. Once you have created a layer using a PostGIS query, you can change it using the Query Builder button found on the General tab of the vector Layer Properties dialog box. Clicking the button opens the query builder, allowing to modify (or completely change) the query that defines the layer.

Creating a Spatial View

If you are a SQL wizard (or wizard-in-training), you can accomplish the same effect as our "virtual layers" using database views. Whether you choose to do this depends on how frequently you need to access the filtered data. If you always use a certain subset of a given layer, creating a spatial view is a good solution. For example, to make our schools layer always available, we can create a view using psql, the PostgreSQL interactive terminal (of course, you could use any tool that can access PostgreSQL and execute queries).

```
$ psql gis_data
Welcome to psql 8.1.3, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

gis_data=# create view school_view as select * from gnis where type = 'school';
CREATE VIEW
gis_data=#
```

This creates a view for us that includes all the columns from the gnis table but includes only those features that are schools. When you fire up QGIS and connect to the database, you'll find the school_view in the list of available PostGIS layers.

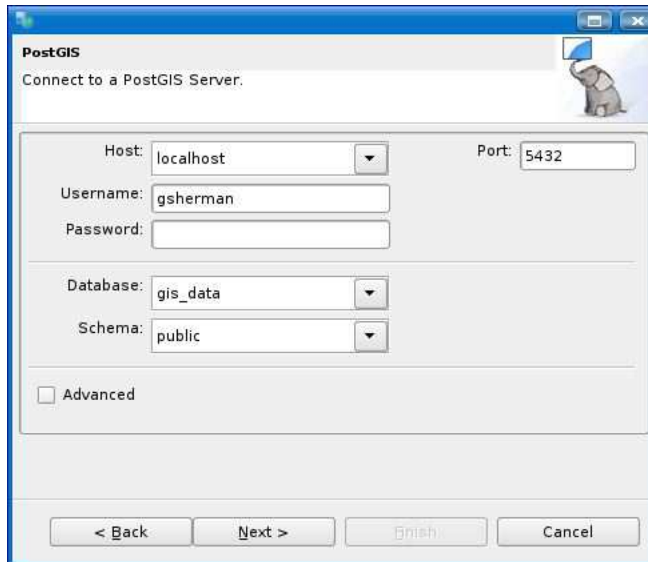


Figure 7.7: PostGIS connect dialog box in uDig

7.5 Using PostGIS and uDig

You can use uDig to display PostGIS layers—which is no surprise since both come out of Refractions Research.[10] If you look back to Figure 3.1, on page 40, you'll recall that PostGIS was one of the choices when adding data to the map.

Adding a PostGIS layer is pretty easy—you just have to know the particulars of your database location and connection parameters, just as we did with QGIS. In Figure 7.7, you can see the uDig PostGIS connection dialog box with our connection parameters filled in. Once we click Next, we are presented with a list of layers in the database that can be added to the map. uDig doesn't currently support the filtering of PostGIS layers, so we can't create a "virtual" layer. Once the layer is added to the map, you can symbolize it just like we discussed in Section 3.2, *Rendering a Story*, on page 42, including the use of color palettes.

Once you've made a connection, uDig keeps it available in the catalog, accessible at the bottom of catalog, accessible at the bottom of the

10. http://refractions.net

workspace. When you click the Catalog tab, you'll get a list of the data stores available to you, one of which will be your PostGIS connection(s). If you expand the PostGIS node, you'll see a list of all the layers for a given connection. To add one of the layers, simply right-click it and choose Add to New Map or Add to Current Map. You'll note this is also a quick way to create a new map and get some data on it. If you choose Add to New Map, a new map tab is created and named the same name as the layer you chose.

7.6 Summing It Up

You now have been exposed to the power and flexibility of a spatially enabled database. Should you use a spatial database or stick to file-based data like shapefiles? That depends on your needs and goals. If you have large datasets that you want to create "virtual" layers from using views or definition queries, a spatial database is the way to go. Another good reason is to create a centrally located, shared data source for multiple users.

A spatial database adds a bit of complexity in terms of getting started, but it's worth the effort when managing large datasets and many layers. If you are a casual user, you may find it's not for you—again, it depends on your goals and needs.

Lastly, you may be wondering why we are talking about server software in a desktop book. If you've gotten this far, you realize that the "back end" is just as important as the front. Using a spatial database provides a data store that we can use on the desktop, as well as for web mapping applications. From that perspective, it's a good choice as a central repository for all our data.

Chapter 8

Creating Data

Using existing data is fine and gives us a lot of capability—until we want to display data specific to our area of interest. Sometimes we luck out and find the data; other times we have to create or convert it. At some point in your OSGIS career, you are going to need to do some creation or conversion of data to get what you need. This is where you move on from the hunter-gatherer stage in your GIS data usage.

Ways to create data suitable for our use include the following:

- Digitizing
- Importing from text files or other sources
- Converting data
- Importing GPS data
- Georeferencing an image

In this chapter, we'll explore some of the ways in which we can torture data (whether raw or cooked) into submission and make it usable.

8.1 Digitizing

We've already seen examples of digitizing in the previous chapters. In its nonglamorous form, digitizing is just tracing features, whether it be on a digitizer tablet or the screen. This is a tried-and-true method of generating new vector data from paper or a scanned image. If you digitize from a scanned image displayed on your screen (as we did in Chapter 5, *Digitizing and Editing Vector Data*, on page 81), it's called *heads-up digitizing*—you've got to keep your head up and focused on the monitor to do it. This method of digitizing has become quite prevalent

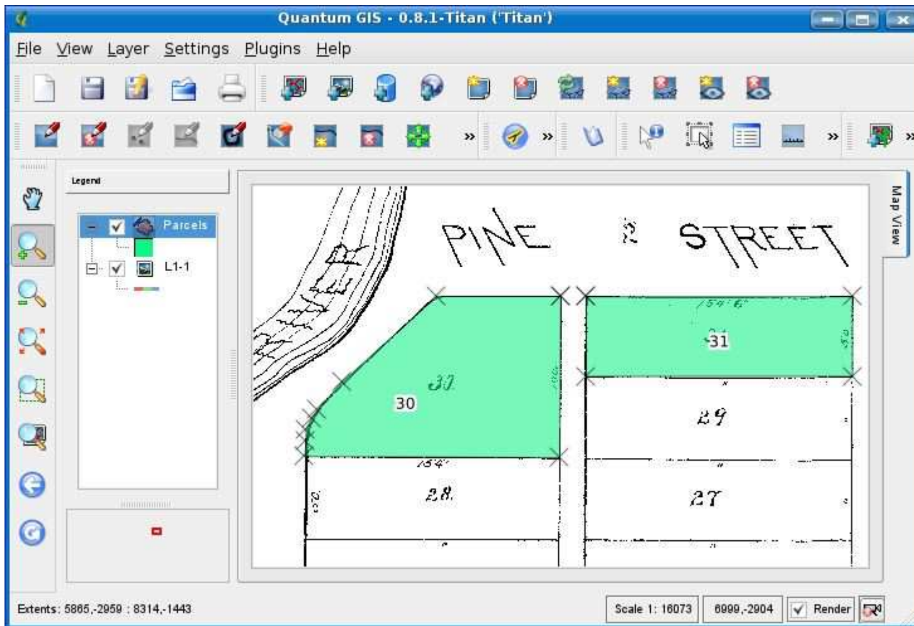


Figure 8.1: Digitizing a subdivision plat

with the availability of imagery and the ability to scan large documents and maps into a format suitable for onscreen display.

As an example, we went out on the Internet and dug up an 1882 subdivision plat from Wichita, Kansas. Our task is to create a vector layer from the plat for use in historical archiving or some other creative purpose. Of course, digitizing plats is an ongoing activity for government entities. Our plat is a TIFF image and is not georeferenced. Since we don't have any reference points to register it, we'll pretend it's in the proper coordinate system. If we were doing this for real and wanted the vector layer to overlay other city features, we would need to get it georeferenced first. In Figure 8.1, you can see our work partially complete, with the completed parcels shown in green. The plat itself is a black-and-white scan of an original paper plat. We created a new shapefile using QGIS and began digitizing the plat, storing the lot number for each parcel in the attribute table as we go. When complete, this gives us a new vector layer that contains the parcel number for each lot. This in turn can be displayed with other vector layers in the same coordinate system or linked by parcel ID to additional data in a database.

An alternative to digitizing the raster is to use the GRASS r.to.vect command. This will create a vector layer from the raster. The results depend on the quality of the raster. In the case of our plat, we end up with a huge vector layer containing 70,607 polygons. When using this method, everything on the image gets converted, including the text. On our example plat, the quality of the lines around the individual lots is such that we end up with an inner and outer polygon, the outer one being as wide as the line on the image. In addition, the process creates a polygon for the entire image boundary. You could spend as much time cleaning up the result as digitizing the polygons from scratch. To aid in cleaning up the vectors, you can use v.clean with the tool=rmarea tool to remove small areas. The other option may be to preprocess the image using a graphics program to remove some of the noise and unwanted information. In any case, you may find that r.to.vect is an effective solution when you need to vectorize a raster.

Digitizing is an activity that you'll find insanely boring, tedious, interesting, or therapeutic, depending on your outlook. It remains an important means to create vector data from raster.

8.2 Importing Data

Another important way to get data into your GIS realm is by importing it from text or other source files. Depending on the format of the data, you may find there is a ready-made solution for importing it. A prime example of this is delimited text that can be easily imported by both QGIS and GRASS. QGIS supports the import of points only, while GRASS can accommodate all feature types in the GRASS vector model.

Quite often you find yourself with some text data that contains coordinates and other attribute information that's begging to go into your GIS. Let's start with a simple example to get started. We want to create a data layer of all the volcanoes in the world. Using a search engine, we find a website[1] that provides a means to search for volcanoes and their locations. By not entering any search parameters, we are presented with a web page containing a table of all volcanoes and their locations.

Since the website doesn't provide a download of the data, the first thing we need to do is copy and paste the results into a text editor. Doing

1. http://www.ngdc.noaa.gov/seg/hazard/vol_srch.shtml

this provides us with a text file containing a header row with the field names, followed by a row for each volcano:

```
Number  Volcano Name  Region  Latitude  Longitude Elev  Type  Status  Last Known
Eruption
0803-001  Abu Honshu-Japan  34.5  131.6 571 Shield volcano  Holocene  Unknown
0103-004  Acigol-Nevsehir Turkey  38.57 34.52 1689  Maar  Holocene  U
1505-017  Acotango  Chile-N -18.37  -69.05  6052  Stratovolcano Holocene  U
1101-112  Adagdak Aleutian Is 51.98 -176.6  645 Stratovolcano Holocene  U
```

The file looks a bit scrambled up with no clear spacing or delimiter. Looking at the text file in our editor, we discover that the columns of the table are separated by a tab character. We can use tab as our delimiter to import the data. The only change we need to make is to clean up the header row (the first line of the file). We can modify the field names to shorten them and make them more appropriate for import. The other change is to delete the second line of the file, since the Last Known Eruption field name is broken across two lines. Our changed header now looks like this:

```
Number  Name  Region  Latitude  Longitude Elev  Type  Status Last_Eruption
```

When making the changes, make sure that each field name is separated by a tab character. Otherwise, the import won't work properly.

Importing Data with QGIS

With the header row fixed, we are ready to import the data. First we will use the QGIS Delimited Text plugin to load and view the data. From the QGIS Plugin Manager, load the plugin to add the tool to the Plugins toolbar.[2] Click the Delimited Text tool to begin the import. In Figure 8.2, on the next page, you can see the delimited text plugin dialog box, populated with the parameters for our input file.

We used the browse button to populate the delimited text file box with our prepared file. The plugin is actually pretty smart. When you enter the filename, the remainder of the parameters for the import are populated using an educated guess. This includes the layer name (based on the input filename) and the delimiter. If you use \t (notation for a tab character) for the delimiter, the plugin parses the input file and makes a guess as to the x and y fields—in this case longitude and latitude. Basically it's ready to go—all we have to do is click the Add Layer button. If the delimiter is not a \t, you'll have to enter the proper delimiter

2. For an overview of plugins in QGIS, see Section D.4, *Plugins*, on page 339.

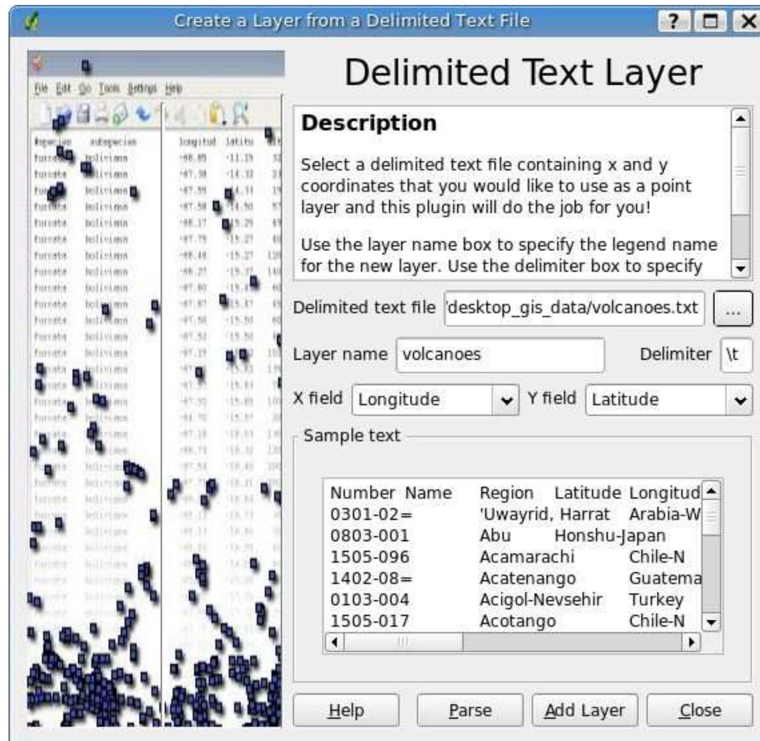


Figure 8.2: The QGIS Delimited Text plugin

and then use the Parse button. You can then choose the x and y fields from the drop-down boxes.

Once you've added the text file to the map, you can use it just like any other layer, including identifying features and viewing the attributes. So far we haven't really imported anything. The Delimited Text plugin includes a *data provider* that allows QGIS to treat the text file like a true layer. Essentially a data provider acts as a translator between QGIS and the data store, whether it be a text file, OGR layer, or PostGIS layer. If you are happy with the text file, you can save it as a shapefile by right-clicking the layer in the legend and choosing Save as shapefile. The next time you want to use the data, just load the shapefile rather than going through the text import process.

Keep in mind that you can use anything for a delimiter—it doesn't have to be a tab character. QGIS also supports the use of regular expressions when defining the delimiter, allowing you to import text data that is not entirely uniform.

Preprocessing Text for Import

Sometimes the data you find may not be ready for import. When this happens, you're faced with preprocessing; either in an editor or with a script. Let's look at an example that illustrates this point.

Say we want to plot historic earthquakes in Alaska. To get started, we can download earthquake data for Alaska from 1898 through 2006 at http://www.aeic.alaska.edu. Upon the examination of the data, we find that it is a fixed-length format. Some columns have blanks in some rows and values in others. This means we can't just split the record up on the whitespace to get the fields.

```
Mo/Dy/Year  Hr:Mn:Sec     Latitude   Longitude    Depth     mb      ML      MS
                                                   (km)
12/23/1906  17:21:11.7   56.8500 N  153.9000 W     0.0              7.3     7.3
08/22/1907  22:24:00.0   57.0000 N  161.0000 W   120.0     6.5     6.5
05/15/1908  08:31:36.0   59.0000 N  141.0000 W    25.0              7.0
```

Note that the "mb" column has a value in the second row but not the first and third.

To properly parse the records and get the fields we want, we resort to writing a small Ruby script to prep the data. In addition to breaking it out by fields, we also check for longitudes in the western hemisphere and set them to negative to make sure they plot where they should in the world.

Download prep_earthquakes.rb

```
#!/usr/local/bin/ruby
# Prep a text file of earthquake events with fixed length records to be
# imported as delimited text. The "|" is used as the delimiter.
#
f = File.open("db_search2291")
# Skip the first two header records
2.times {f.gets}
# print the delimited header row containing the fields we are interested in
print "event_date|event_time|latitude|longitude|depth|magnitude\n"
# process the earthquake records
while not f.eof
  record = f.gets
  # use a fixed length approach to get the fields we want since
  # splitting on white space isn't feasible
```

```
  event_date = record[1..10]
  event_time = record[13..22]
  latitude = record[26..32]
  longitude = record[37..44]
  longitude_direction = record[46..46]
  depth = record[50..54]
  magnitude = record[66..69]
  # if the longitude is in the western hemisphere, it must be
  # negative
  if longitude_direction == 'W'
    longitude = -1 * longitude.to_f
  end
  # print a delimited record
  STDOUT << event_date << "|" << event_time << "|" \
    << latitude << "|" << longitude << "|" \
    << depth.strip << "|" << magnitude.strip << "|\n"
end
# close the input file
f.close
```

When we run this script, we get a nicely formatted file, delimited with | and containing only the fields in which we are interested.

```
event_date|event_time|latitude|longitude|depth|magnitude
06/29/1898|18:36:00.0|52.0000|172.0000|0.0|7.6
10/11/1898|16:37:32.7|50.7100|-179.5|0.0|6.9
07/14/1899|13:32:00.0|60.0000|-150.0|0.0|7.2
```

You can now import the data using the Delimited Text plugin in QGIS, using the same method as we used with the volcano data. The only difference is this time we are using | as a delimiter.

If you are lucky, you won't have to go through a big preparation process before importing your data. Oftentimes you can patch up the text file using a text editor and global search/replace to get it formatted for import. When you can't just write a quick Ruby (or Python or Perl) script to do the job.

Importing with GRASS

Once we have the delimited text file formatted, importing into GRASS is even quicker than using QGIS. From the GRASS shell, we can use the v.in.ascii command to import the data. If you don't specify column names, GRASS assigns default names that may not be too meaningful. We want meaningful names, so for the columns option, we specify the name for each in SQL style.

```
> v.in.ascii input=earthquakes_delim.txt output=earthquakes skip=2 \
      x=4 y=3 cat=0 columns="event_date date, event_time varchar(10), \
      lat double precision, lon double precision, depth double precision, \
      magnitude double precision"
Maximum input row length: 57
Maximum number of columns: 6
Minimum number of columns: 6
column: 1  type: string  length: 10
column: 2  type: string  length: 10
column: 3  type: double
column: 4  type: double
column: 5  type: double
column: 6  type: double
Building topology ...
12035 primitives registered
Building areas:  100%
0 areas built
0 isles built
Attaching islands:
Attaching centroids:  100%
Topology was built.
Number of nodes     :   11938
Number of primitives:   12035
Number of points    :   12035
Number of lines     :   0
Number of boundaries:   0
Number of centroids :   0
Number of areas     :   0
Number of isles     :   0
```

The options to the v.in.ascii command are explained in the GRASS manual, but basically apart from the input and output names, we told the command to skip the first two lines since they are header lines and that our x coordinate is in column 4 and the y coordinate is in column 3 of the input file. We also specified cat=0 to indicate we wanted GRASS to create an ID column for us. If the input file had a suitable ID field, we would have used it by specifying its column number with the cat option.

Notice that we didn't specify the | delimiter when using v.in.ascii. That's because it is the default delimiter. If we had used a different delimiter when preparing the text file, we would need to use the fs parameter to specify it.

Once we have our new earthquakes layer imported into GRASS, we can symbolize it by magnitude to see where in Alaska we shouldn't live. In Figure 8.3, on the following page, you can see a portion of southcentral Alaska with the earthquakes symbolized by magnitude.

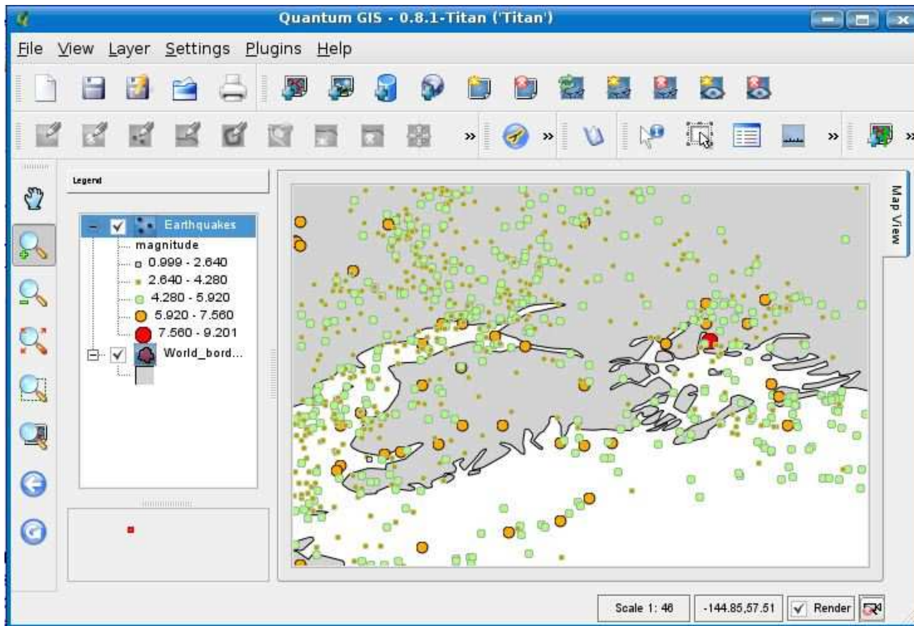


Figure 8.3: Earthquakes rendered in QGIS by magnitude

As you can see, it's relatively easy to import text data into QGIS or GRASS. Our examples dealt only with importing points. As I said earlier, GRASS supports importing all feature types in "standard" mode. See the manual page (g.manual v.in.ascii) for details and examples on importing types other than points.

8.3 Converting Data

Sometimes we are faced with converting data before we can use it, simply because it's not in the format that works with our software. You can probably think of other reasons as well; for example, you want to share the data with someone using another type of software. Or perhaps you might want to transform the data to another projection to make it play nicely with your other data. You might recall we talked a bit about standardizing your data format in Chapter 6, *Data Formats*, on page 91.

A lot of times data is distributed in a form that's convenient for the distributor, not the end user. This is probably the most common circumstance you'll encounter when gathering data from the Internet. Let's take an example.

Importing an E00 Interchange File

Harrison is interested in some data from his local state government. He soon discovers that they, like a lot of other government entities, deliver their data in a format called an E00 file.[3] To make use of the data, Harrison has to convert it into a format his software supports. He has several options for converting the interchange file. First he can search around the Internet for one of several E00 to shapefile converters.[4] The other option is to use GRASS to import it right into his favorite mapset. Once in GRASS,[5] he can also export it in a number of formats to share with his friends. To convert an E00 interchange file to GRASS, Harrison uses the following:

```
GRASS 6.2.2 (alaska_albers):~ >
v.in.e00 file=/home/harrison/itma.e00 type=area vect=quad_boundary
Importing areas...
Over-riding projection check.
Proceeding with import...
Layer: LAB
WARNING: Column name changed: 'ITMA#' -> 'ITMA_'
WARNING: Column name changed: 'ITMA-ID' -> 'ITMA_ID'
WARNING: Column name changed: 'TILE-NAME' -> 'TILE_NAME'
Importing map 3011 features...
Layer: ARC
WARNING: Column name changed: 'ITMA#' -> 'ITMA_'
WARNING: Column name changed: 'ITMA-ID' -> 'ITMA_ID'
Importing map 6705 features...
-----------------------------------------------------
Building topology ...

Imported area vector map <quad_boundary>.

  .... more messages ....

Done.
GRASS 6.2.2 (alaska_albers):~ >
```

If Harrison needs to share the new layer with somebody, he quickly creates a shapefile using v.out.ogr.

3. This is an ArcInfo interchange format that is not directly usable by most GIS software.
4. Such as http://avce00.maptools.org
5. If you need help getting GRASS started, see Appendix C, on page 296.

```
GRASS 6.2.2 (alaska_albers):~ > v.out.ogr input=quad_boundary type=area dsn=. \
  olayer=quad_boundary.shp format=ESRI_Shapefile
Exporting 3011 areas (may take some time) ...
 100%
3011 features written
GRASS 6.2.2 (alaska_albers):~ >
```

You'll find that GRASS supports a lot of input and output conversions for both getting new data into GRASS and exporting it out for use with other applications or to share with others.

Another great way to convert both vector and raster formats is using the GDAL/OGR suite of tools. Since there are so many possibilities, we'll explore these tools in Section 11.2, *Using GDAL and OGR*, on page 186.

8.4 Using GPS Data with QGIS

GPS units are everywhere these days. Between the practical use for the professional and the recreational user, as well as the popularity of geocaching,[6] it seems like everybody is using them. I'm sure you would like to display your GPS adventures on a topographic (read DRG) map, especially after we work through how to create seamless rasters using GRASS in Chapter 10, *Geoprocessing*, on page 149. Well, the good news is there are lots of open source tools available for working with your GPS. You'll need a GPS with an interface cable so you can move data to and from your computer. In this section, we'll show you how to put your data on the map using the GPS plugin that comes with QGIS.

Getting Set Up

Obviously you need QGIS installed and working on your platform. The only other requirement (apart from a GPS unit) is gpsbabel.[7] QGIS uses gpsbabel to import other formats and for GPS downloading and uploading operations. Fortunately, gpsbabel runs on all the same platforms as QGIS, so if you can run QGIS, you can use it with your GPS. Actually, gpsbabel is a remarkable little program. It runs on almost everything and supports 100+ formats and a bunch of GPS hardware. For upload/download, if you have a Garmin or Magellan unit, you should be good to go. For others, see the format list on the web page to see whether yours is listed there.

6. Treasure hunting with a GPS. See http://www.geocaching.com.
7. http://www.gpsbabel.org

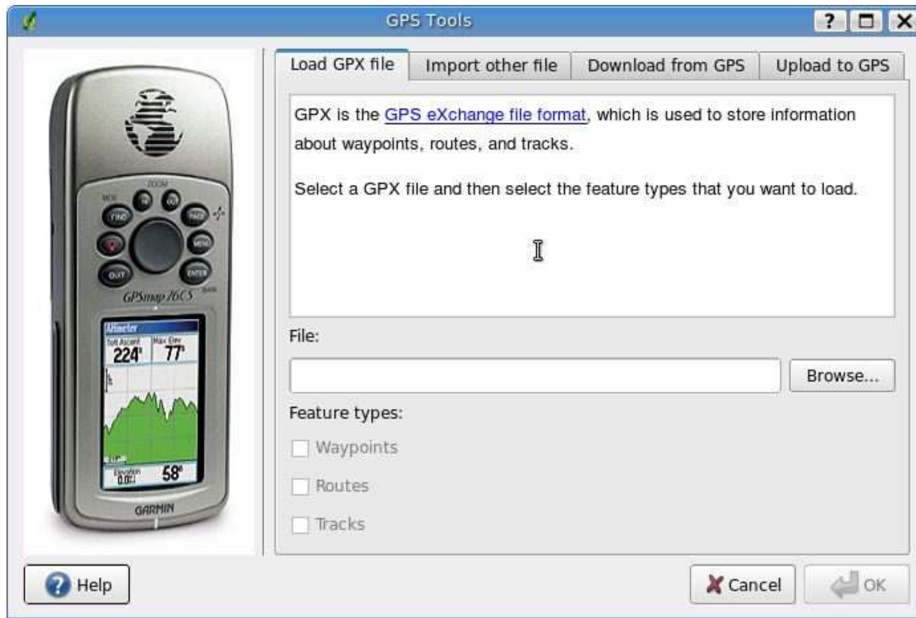


Figure 8.4: GPS plugin in QGIS

The GPS Plugin

The GPS plugin is part of the core distribution of QGIS. This means it is included by default with your installation of QGIS. Like other plugins, you have to load it before you can get at it from the Plugins toolbar. If you don't know how to load a plugin in QGIS, take a look at Section D.4, *Plugins*, on page 339. Once you have the plugin loaded, you'll see a nice little GPS icon on the toolbar. Clicking it will open the interface you see in Figure 8.4.

The first thing to notice is the four tabs across the top of the GPS plugin dialog box. The first two (Load GPX File and Import Other File) allow you to load data stored on a disk, a CD, or maybe a thumb drive. The last two tabs (Download from GPS and Upload to GPS) provide the tools needed to move data to and from your unit. Now that we have the plugin loaded up and ready to go, let's begin by fetching some GPS data from our unit so we can display it.

Joe Asks...

What Is GPX?

GPX stands for GPS Exchange Format. It's a lightweight XML format for interchanging your GPS data between applications, both on the desktop and the Web. GPX can handle waypoints, tracks, and routes. Using the GPX format, you can easily exchange your data between a host of GPS and GIS applications. QGIS directly supports the GPX format using the GPS data provider that is included with all versions of QGIS. For a list of applications that can work with the GPX format, take a look at the TopoGrafix website.*

*. http://www.topografix.com/gpx_resources.asp

Downloading Data from Your GPS

OK, so we assume you have your GPS hooked up to the computer with your interface cable and the unit is powered on. For now we'll also assume that it's a Garmin, because the plugin is set up for that already. Downloading from your GPS is easy. Just click the Download from GPS tab, and select the port your interface cable is using. The port names will vary depending on your operating system. Usually you'll find the appropriate port in the drop-down list.

Since QGIS is feature oriented, you have to specify whether you are downloading waypoints, routes, or tracks. Just choose what you want from the drop-down box.

When you download from your GPS, QGIS creates a new layer and also saves the data in a new GPX file. For it to do that, you have to provide an output filename and a name for the layer. Keep an eye on the big gray OK button in the bottom-right corner. Once you fill in all the required information, it will turn green, meaning you are ready to go.

When you click OK, the data is pulled from the GPS unit and stored on disk, and a new layer is added to the QGIS map canvas. Repeat the process to fetch each of the feature types you want to download (waypoints, tracks, and routes). In Figure 8.5, on the next page, you can see our GPS data added to the map. To make it more interesting, we've added the world mosaic raster and the world_borders shapefile as

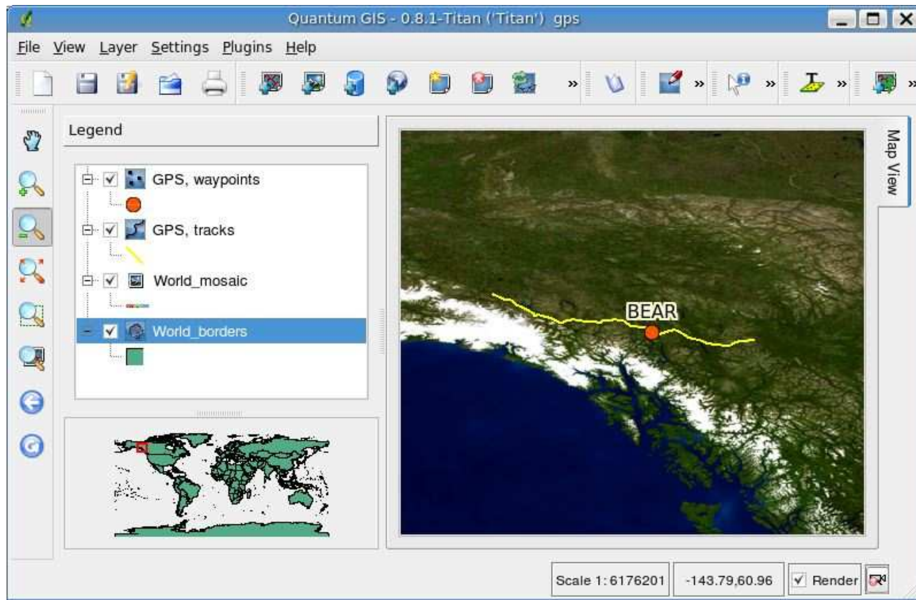


Figure 8.5: Track and waypoint loaded from GPS unit

a backdrop for our data. We've also labeled our lonely little waypoint using the data that came with it from the GPS, in this case, BEAR. Once loaded into QGIS, the GPX data behaves just like any other layer. You can label it, change the symbol type and colors, and rename it in the legend. In case you're wondering, we called the waypoint BEAR because it happens to be the point in Canada where we saw a nice fat black bear feasting on berries. My plan was to geotag the pictures I took on our road trip, but it didn't really work out that way. But that's another story. . . .

Loading and Viewing Data

So far, we have pulled the data from our GPS unit, displayed it, and in the process created a GPX file on disk. The next time we want to display it, we don't have to pull it off the GPS again (well we could, but only if we felt we needed the practice). We already have the file saved to disk—to load it we again use the GPS plugin. You can't load a GPX file into QGIS without it.

From the plugin, we just click the Load GPX File tab if it's not already active and then enter or browse to the location of our .gpx file. Using the three checkboxes below the filename, we can choose to load all the feature types from a GPX file or just some. For each box checked, you will get a separate layer in QGIS.

You may have noticed that the GPX files we created by downloading from our GPS contain only one feature type each. You likely also noticed the three checkboxes when loading a GPX file. QGIS allows you to load multiple feature types because the plugin assumes you may have GPX files from other sources that contain multiple types.

The GPS plugin also allows you to load other formats supported by gpsbabel. Say you have a batch of files from a GPS that you want to view. If you click the Import Other File tab, you can convert them to GPX format so they can be used with QGIS. The real work here is done by gpsbabel, so you must have it installed on your system in order for this to work. This is just a handy way to convert files and get them into QGIS. Of course, you could also just use gpsbabel from the command line to accomplish the same task.

Uploading Data to the GPS

The last thing we'll look at is uploading data from QGIS to your GPS unit. Here are a couple of reasons why you might want to do that:

- You download some routes for trails in your area from your local parks department, and you want to load them on your GPS.
- You have edited your waypoints and tracks from your GPS and want to load them up.

In the second scenario, you'll notice that yes—you can edit your GPX layers in QGIS and change not only the location (more on that in a second) but also the attributes. So if you're like me and have trouble entering text for waypoints on the little spongy rubber buttons, you can now edit the things after the fact to correct and enhance them. You may be thinking that editing the locations kind of violates the intent of a GPS. Well, there may be some circumstances where you might want to move a waypoint, for example, if you have better information gleaned from a data source with better accuracy than your GPS.

The other thing you can do is add new features. So, for example, you could digitize trails from a DRG and upload them to the GPS. Of course, you need to start with an existing GPX file since the plugin doesn't

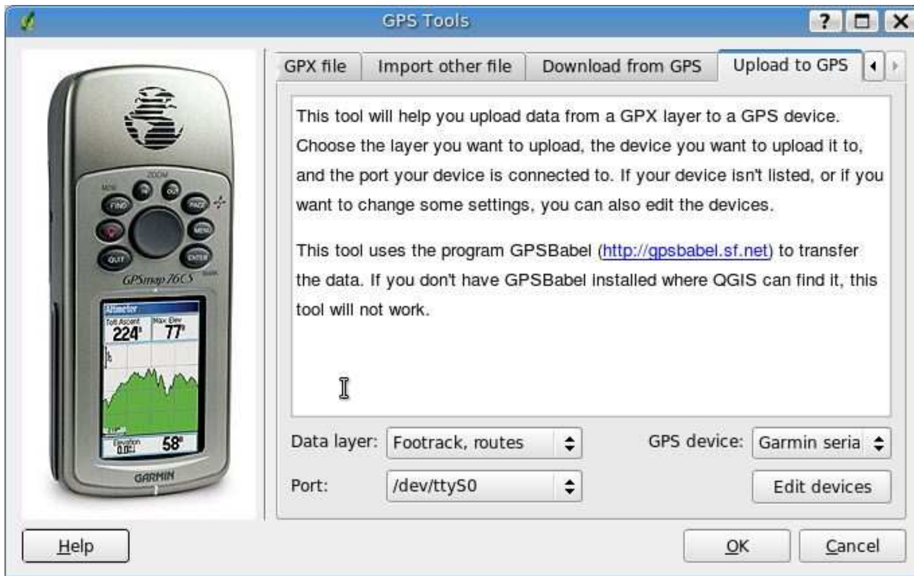


Figure 8.6: Uploading to the GPS

provide a way to create a new one. Once you're done, you can hit the trails confident that you won't get lost (depending on how good your digitizing was). Naturally, the better way is to download the data from the local parks department and put it on your GPS, but sometimes that information isn't available, especially in a GPS format.

If we have some data we want to upload, we can just open the plugin and click the Upload to GPS tab. From the Data Layer drop-down list, you can select the GPX layer you want to upload. Notice that it has to be a layer that's already on the map. You can't upload an arbitrary file using the plugin. Once you select the device and port, just click the OK button and watch the data fly onto your GPS. In Figure 8.6, you can see the plugin ready to upload a routes layer to our GPS unit.

As you can see, the GPS plugin relies on gpsbabel to do the heavy lifting.

8.5 Georeferencing an Image

In case you haven't picked up on it by now, a georeferenced image is one that has associated coordinate information such that it "draws"

where it belongs in the world. An image that isn't georeferenced in GIS is not much better than a photograph. You can still look at it, but you can't really do anything GIS-like with it, such as overlay vector data or digitize from it.

Georeferencing with QGIS

QGIS includes a plugin to georeference an image, provided you have the x and y coordinates needed to establish control points. A control point is a point on the image that you can determine the real-world coordinates for with a high degree of accuracy. The more accurate your control points, the better the "fit" of the georeferenced image.

To begin, load the Georeferencer plugin from the QGIS Plugin Manager. To start the process, click the Georeferencer tool on the plugin toolbar, and select a raster (image) file using the browse button. Once you have the full path to a raster entered into the Raster File text box, click the Enter World Coordinates button. This will open a new window containing your image and a toolbar used to set control points, as well as navigate around the image by zooming and panning. Zoom in on the image to the location of one of your control points, click the Add Point button in the toolbar, and then click the map to add the point. When you click, you will be prompted for the x and y coordinates for the control point.

At this point you have two choices. You can enter the coordinates by hand if you know them, or you can pick the coordinates from the QGIS map canvas. The latter method allows you to use a "control" raster that is already georeferenced. Either method can be accurate, depending the quality of your control points.

Continue to add control points, preferably covering the extents of the raster. In Figure 8.7, on the next page, we have loaded an image and started to add control points by picking them from the QGIS map canvas.

Once the control points are established, enter a name for the world file to be created. The Georeferencer plugin creates a world file to go along with the image, rather than encoding the coordinate information in the raster itself. You can also choose the transformation method. If you choose Helmert, you will either need to enter a name for the new raster to be created or just accept the default name. Now just click the Generate World File button to complete the process. Alternatively, you can use the Generate World File and Load Layer button to load the layer

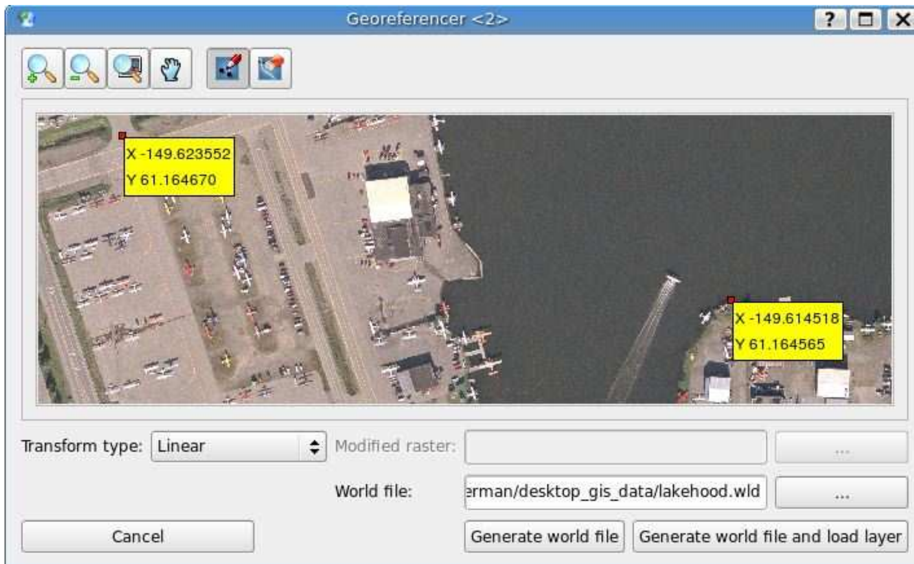


Figure 8.7: Georeferencing an image with QGIS

in QGIS after the world file has been generated. Once you have the layer up in QGIS, check it against any reference layers you may have to make sure the georeferencing was a success.

Georeferencing with GRASS

There are two ways to georeference an image using GRASS. You can use what's termed the "old" way, or you can use the gis.m GUI. The old way involves a sequence of commands as described on the GRASS FAQ:[8] i.group + i.target + i.points/i.vpoints + i.rectify. The "new way" is to use gis.m and the Georectify item under the File menu.

The process is pretty much like that we used with QGIS. We won't look at an example here, but you can explore the "old" and "new" ways on your own if you decide to use GRASS for georeferencing rasters.

Of course, the best option is to find rasters that are already georeferenced. In many cases, you can do this—if not, you have to use your newly acquired skills to get the job done.

8. http://grass.gdf-hannover.de/wiki/Georeferencing

Chapter 9

Projections and Coordinate Systems

If the world were flat, it would be a lot easier—at least on mapmakers. Unfortunately, that's not the case, so we're faced with the age-old problem of depicting features on a spheroid (that's the earth) on a flat piece of paper (or screen).

To solve this problem over the years, people have come up with the concept of map *projections*. The key thing to remember about projections is that none of them is perfect. You simply can't represent the entire earth (or even a small part of it) on a flat surface without some distortion. The amount of distortion varies with the projection. Many projections are quite good when used for a small or regional area. If you try to use the same projection for a larger area, the distortion increases.

Let's look at the main problems with squashing the earth onto a piece of paper. It's impossible for a projection to maintain an accurate portrayal of area, distance, shape, and direction all at once. For this reason, you'll find that some projections are more suited for your use than others. For example, if we're interested in measuring the areas of lakes we've digitized, we want a projection that is equal-area. This means that for any given location on the map, the measured area will be correct. If we are interested in measuring distances, we obviously need an equidistant projection.

In choosing a map projection, we need to decide whether our focus is on shape, direction, area, or distance. Once we know that, we can choose an appropriate projection. Of course, sometimes you don't get

a choice. You are forced to work in a particular projection for one reason or another. When Harrison wanted to display his bird sightings on the DRG, he needed to make sure they were in the same projection. Rather than "warp" the raster, he found it easier to convert his sightings from geographic to UTM, the same projection as the DRG. You can warp your rasters (no, it's not illegal) if you find it more convenient than transforming the dozens of vector layers in your dataset. For an example of warping a raster, see Section 11.2, *Raster Conversion*, on page 196.

There are plenty of books and online resources that delve into the details of projections and datums. Our goal here is to give you a brief yet practical introduction to provide what you need to know to work with your data. At the end of the chapter, you'll find some additional resources you can use to learn more about the sometimes complex world of projections and coordinate systems.

9.1 Projection Flavors

Projections come in three main flavors: planar or azimuthal, conic, and cylindrical. The type indicates how the projection is constructed.

Azimuthal
: In an azimuthal projection, the sphere (that's the earth) is projected onto a flat or planar surface. Examples of azimuthal projections include Orthographic, Stereographic, Gnomonic, Azimuthal Equal Distant, and Lambert Azimuthal Equal Area.

Conic
: In a conic projection, a spherical surface is projected on to a cone. Examples of conic projections include Albers Equal Area, Lambert Conformal, Equidistant Conic, and Polyconic Conic.

Cylindrical
: In a cylindrical projection, the sphere is projected on to the walls of a cylinder. Examples of cylindrical projections include Mercator, Transverse Mercator, Oblique Mercator, Space Oblique Mercator, and Miller Cylindrical. There are also a couple of pseudocylindrical projections: Robinson Pseudo-cylindrical and Sinusoidal Equal Area Pseudo-cylindrical.

Of course, the last projection we need to mention is Geographic, which really isn't a projection at all. It's just a coordinate system of latitude

Joe Asks...

What's a Datum?

Although we could go into a complicated definition, a *datum* is just a model for determining the coordinates of points on the earth. You are likely to encounter the North American Datum of 1927 (NAD 27), the North American Datum of 1983 (NAD 83), and the World Geodetic System of 1984 (WGS 84).

It's important to make sure that either your data is in the same datum or you are using software that can convert between datums on the fly. What happens if you mix datums? Your data won't line up as it should. All projections are based on a datum, so make sure to understand your data before you start trying to put it all together.

and longitude in a given datum. You'll find a lot of data in Geographic—just make sure that the datum matches your intended use.

9.2 Working with Projections

Let's look at what we need to know to work with projections. When using a chunk of data in our OSGIS software, we should determine the following:

- Projection
- Units of measure
- Datum

In practice, most people don't care too much about these things until they have a problem and the data doesn't overlay properly. Or worse yet, it looks fine, and they think it's fine, but it's not. This can lead to making decisions based on bad information. So, it's best to check your projection parameters to make sure that everything is displayed where it belongs.

Harrison just acquired a new megasized bird layer and is excited to use it. Let's look at the ways he can discover the projection information.

Determining the Projection

There are a number of ways to determine the projection for a dataset. In some cases it's pretty easy—in others it can be quite difficult if the person creating the data failed to include the information from the outset. For example, a shapefile is usually (or should be) accompanied by a .prj file containing the projection information. This is just a text file containing the projection parameters in what is known as Well-Known Text (WKT) format, defined by the OGC OpenGIS Simple Features Implementation Specification for SQL. If you open a .prj file in your favorite text editor, you'll see something similar to this:

```
GEOGCS["WGS 84",DATUM
 ["WGS_1984",SPHEROID["WGS 84",6378137,298.257223563,AUTHORITY["EPSG","7030"]],
 AUTHORITY["EPSG","6326"]],
 PRIMEM["Greenwich",0,AUTHORITY["EPSG","8901"]],
 UNIT["degree",0.01745329251994328,AUTHORITY["EPSG","9122"]],
 AUTHORITY["EPSG","4326"]
]
```

From looking at the WKT, we can determine that this layer is in geographic coordinates (GEOGCS), meaning it's not projected. We also see that it is based on the WGS-84 datum and the units are degrees. There is also a bunch of authority information that indicates the EPSG codes for each section. See the sidebar on page 143 for additional information on EPSG.

This gives us enough information to determine whether the layer can be used with the rest of our data or whether we need to do some conversion to make things line up properly. Let's look at the WKT for a projected coordinate system:

```
PROJCS["Albers Equal Area",
  GEOGCS["clark66",DATUM["D_North_American_1927",
    SPHEROID["clark66",6378206.4,294.9786982]],
    PRIMEM["Greenwich",0],
    UNIT["Degree",0.017453292519943295]],
  PROJECTION["Albers"],
  PARAMETER["standard_parallel_1",55],
  PARAMETER["standard_parallel_2",65],
  PARAMETER["latitude_of_origin",50],
  PARAMETER["central_meridian",-154],
  PARAMETER["false_easting",0],
  PARAMETER["false_northing",0],
  UNIT["Meter",1]
]
```

Here we see that the layer is projected (PROJCS); it's based on the Clarke 1866 spheroid, NAD 27 datum; the projection is Albers; and the units are meters. The WKT also contains the parameters (standard parallels, origin, and central merdian) for the Alaska Albers Equal Area Conic projection.

The key things to look for are the PROJCS or GEOGCS at the beginning of the WKT specification, the PROJECTION and DATUM keywords, and the UNIT keyword. These are enough to tell us whether it's suitable for use with our other data.

The second way to determine the projection for a layer is using the gdalinfo command for rasters and ogrinfo for vector layers. Recall that these utilities are part of GDAL/OGR. We took a look at these commands in Chapter 3, *Working with Vector Data*, on page 37 and Chapter 4, *Working with Raster Data*, on page 67. If you look back to those chapters, you'll see that both commands display the projection information in the same WKT format we just looked at.

The final way to determine a dataset's projection is to load it into your OSGIS application and check the properties for the layer. This works fine and is handy if you already have the layer loaded, but it's quicker to look at the WKT for a vector or use one of the GDAL/OGR commands to get the information. Since gdalinfo and ogrinfo work with nearly every format you'll encounter, it's worth installing and using them.

Data Problems

If you find that your data isn't lining up like you expect, it's either a projection problem or you just have lousy data. Seriously, though, most alignment problems are due to either a projection problem or differing datums. The first thing to do is use your sleuthing skills to examine the projection and datum for your layers. If you are seeing a big difference in alignment, it's likely a projection problem. If it's a small difference (less than 500 meters, for example), you likely have a datum problem.

To get to the bottom of it, you have a couple of choices. If your OSGIS application supports on-the-fly projection (and datum shift if required), enable it and make sure that the projections are recognized by the application. If this solves the problem, you don't need to do anything else; just keep in mind that you are transforming data on the fly—you haven't changed the original data in any way.

EPSG

If you are wondering about the EPSG notation that has popped up in the Well-Known Text of coordinate systems, it represents a dataset of coordinate systems formerly distributed by the former European Petroleum Survey Group. In 2005 the European Petroleum Survey Group was absorbed into the OGP Surveying and Positioning Committee. The OGP continues to distribute the dataset.

The Geodetic Parameter Set contains a unique code for each coordinate system, as well as details about the projection. Many OSGIS applications can use the EPSG code as input when doing transformations. If you have PROJ.4 installed, you should have a copy of the .epsg file on your system. This file is a simplified subset of the EPSG definitions and maps the EPSG number to the corresponding PROJ.4 parameters. The full EPSG database and documentation is available from OGP.*

*. http://www.epsg.org

If on-the-fly transformation isn't an option, you will have to manually transform the data to get it into a projection you can use. If you are using PostGIS, you can create a SQL view of your data that transforms the geometries using the OGC transform function. You would then load the view into QGIS or your other application, and the transformation would be done automatically, at the expense of a bit of performance. For example, if we have a towns layer in our PostGIS database that we would like to use with another layer that is geographic, we can create a view to do the job.[1] First let's look at the schema of the table:

```
gis_data=# \d towns                              Table "public.towns"
   Column   |         Type          |              Modifiers
------------+-----------------------+-------------------------------------
 gid        | integer               | not null default
            |                       | nextval('towns_gid_seq'::regclass)
 name       | character varying(20) |
 class      | character varying(35) |
 pop        | integer               |
 shape      | geometry              |
 ...
gis_data=#
```

1. The coordinate system for our towns layer in PostGIS is Albers Equal Area.

Next we'll select a few records from the table to examine the coordinates, just so we can see that our transform works when we're all done:

```
gis_data=# select gid, astext(shape) as coordinates from towns limit 5;
 gid |           coordinates
-----+--------------------------------
   1 | POINT(-820805.1875 506479.46875)
   2 | POINT(-384408.6875 1333234.375)
   3 | POINT(88849.421875 881200.0625)
   4 | POINT(-565926.875 1174128)
   5 | POINT(-307637.65625 1451385.5)
(5 rows)
```

Now we can create the view using the transform function to convert from the projected coordinate system to WGS 84 geographic (EPSG:4326). We know the EPSG code is 4326 because we looked it up using one of the methods that you'll learn about in just a moment. We can now create the view:

```
gis_data=# create view towns_geo as select gid, name, class, pop,
            transform(shape,4326) as shape from towns;
CREATE VIEW
gis_data=#
```

Now we can use our towns_geo view as a layer in QGIS. Just to make sure the transform works, let's select a few records to see whether the coordinates look like they are geographic:

```
gis_data=# select gid, astext(shape) as coordinates from towns_geo limit 5;
 gid |                coordinates
-----+------------------------------------------
   1 | POINT(-157.571463607738 51.336408724444)
   2 | POINT(-155.770900879198 53.6566126757253)
   3 | POINT(-153.605243492496 52.4282023272413)
   4 | POINT(-156.577035823078 53.203783217787)
   5 | POINT(-155.42947766785 53.9856939176996)
(5 rows)
gis_data=#
```

Sure enough, our view is returning coordinates in WGS 84 geographic. The view gives us a quick way to transform our data on the fly and be able to visualize it with our other data. Our original data isn't changed—we're just doing a transform on the fly.

If it comes down to it, you can always transform your data, creating a new dataset in the appropriate projection. We'll cover transformation for both vector and raster layers in more detail later when we look at using command-line utilities. If you're curious, see Section 11.2, *Coordinate*

System Conversion, on page 194 and Section 11.2, *Raster Conversion*, on page 196.

9.3 The PROJ.4 Projections Library

PROJ.4 is a cartographic projections library that is used in many, if not most, open source GIS applications, both on the desktop and on the Internet. It was originally developed by the USGS and is now maintained by a group of volunteers.

You may be wondering why we are mentioning a library—turns out that PROJ.4 also comes with some handy utilities for experimenting with projections and doing interactive transformations:

proj and invproj
: Performs forward and inverse transformations for a large number of projections. With your projection parameters in hand, you can perform a forward calculation using proj (geographic to projected) or an inverse calculation using invproj (projected to geographic). Neither of these utilities does datum shifts.

cs2cs
: Performs transformations between coordinate systems, including datum shifts. With cs2cs, you supply the parameters for both coordinate systems, specifying which is the target or "to" system.

geod and invgeod
: Performs forward and inverse Great Circle (geodesic) transformations. This allows you to calculate latitude, longitude, and back azimuth given a starting point, azimuth, and distance. You can also determine the azimuths (forward and back) and distance between two known points.

nad2nad
: Performs datum conversions between the North American 1927 and 1983 datums. The same conversions can be accomplished using cs2cs.

Let's look at an example. Say Harrison has loaded up some of his bird data in his favorite desktop application and it contains a DRG. If you remember correctly, DRGs come in a UTM projection. Harrison is curious about a couple of bird observations on his map. When he moves his cursor over the points, he sees big numbers for the coordinates in the status bar of his application. He knows his DRG is in UTM Zone 6,

NAD27 datum, but he wants to know the approximate geographic coordinates for the point.[2] PROJ.4 can quickly answer his question. First he has to know the parameters for the UTM projection in order to do the conversion. There are several ways to do this—perhaps the easiest is to look up the proj string in the .epsg file that is installed with PROJ.4.[3] On a Linux system you'll find this in /usr/share/proj/epsg. To locate the projection, you can open the .epsg file in your favorite text editor and search, or you can use grep from the command line:

```
$ grep -C 1 -i "utm zone 6n" /usr/share/proj/epsg| \
    grep -i "nad27"
<26705> +proj=utm +zone=5 +ellps=clrk66 +datum=NAD27 +units=m +no_defs  <>
# NAD27 / UTM zone 6N
<26706> +proj=utm +zone=6 +ellps=clrk66 +datum=NAD27 +units=m +no_defs  <>
```

Here we used grep to search for "utm zone 6n" and told it to print one line on either side of the match (-C 1) and ignore the case (-i). Since we wanted only NAD 27 projections, we piped the output to grep again to show only those lines containing "nad27." From the result we get a couple of things: the EPSG number, in this case 26706, and the string that we need to use with proj. If you can't find the .epsg file on your system, you can search for your projection using the tools on the Spatial Reference website.[4] Entering "nad27 utm zone 6n" as the search string will quickly find the projection. You can then copy the proj parameters from the website.

Now that Harrison has the projection parameters, he can convert his coordinates from UTM Zone 6N to geographic using invproj:

```
$ invproj +proj=utm +zone=6 +ellps=clrk66 +datum=NAD27 \
  +units=m +no_defs
312244.49 6795460.41
150d30'W        61d15'N
```

Harrison entered the coordinates that he read from his status bar (312244.49, 6795460.41) and got the results in degrees and minutes (150d30'W, 61d15'N). Lucky for Harrison, his birds like to roost on nice clean coordinates. Just to convince himself that this works, he plugs

2. If you think there is more than one way to do this, you are right—depending the software you are using.

3. The .epsg file included with PROJ.4 does not include a number of datums outside the United States and Canada. You can find a complete list of datums in your GRASS install in etc/datum.table.

4. http://spatialreference.org

the answer back in to the proj command to see whether he gets his original UTM coordinates:

```
$ proj +proj=utm +zone=6 +ellps=clrk66 +datum=NAD27 \
  +units=m +no_defs
150d30'W  61d15'N
312244.49       6795460.41
```

Sure enough it worked. He could have specified the latitude and longitude using decimal degrees (-150.5 61.25) and gotten the same result. PROJ.4 also has a number of other options, including the ability to customize the output format to your taste. You can see that PROJ.4 can be useful for doing interactive transforms.

If you wanted to transform from a UTM projection to an Albers Equal Area, you would have to do an inverse (invproj) to get the latitude and longitude for the UTM point and then do a forward projection (proj) using the Albers parameters to get the final result. The cs2cs program simplifies this by allowing you to specify both coordinate systems. Let's convert our UTM point from the previous example to Alaska Albers coordinates using cs2cs:

```
$ cs2cs  +proj=utm +zone=6 +ellps=clrk66 +datum=NAD27 \
  +units=m +to +proj=aea +lat_1=55 +lat_2=65 +lat_0=50 +lon_0=-154 \
  +x_0=0 +y_0=0 +datum=NAD27 +units=m +no_defs
312244.49 6795460.41
187115.08       1257043.95 0.00
```

Now we have Albers coordinates 187115.08, 1257043.95 as a result of our transformation. You are probably wondering what the 0.00 means. This is height above (or below) the ellipsoid. Since both datums were NAD27 and they are both based on the same ellipsoid, there is no difference. Let's do a datum shift to illustrate how it's done and compare the results:

```
$ cs2cs +proj=utm +zone=6 +datum=NAD27 +units=m \
  +to +proj=aea +lat_1=55 +lat_2=65 +lat_0=50 +lon_0=-154 +x_0=0 +y_0=0 \
  +datum=NAD83 +units=m +no_defs
312244.49 6795460.41
186991.95       1256960.61 0.00
```

Here we see the result, which is a bit different from the NAD27 result. The total shift in coordinates is approximately 184 meters. If you don't believe it, break out your high-school math book and use the distance formula (Pythagorean Theorem) to check the result.

If you want to transform a lot of points, you can provide the coordinates to cs2cs from a file to do batch conversions. See the documentation for cs2cs for details on the available options.

You may be wondering which of the PROJ.4 utilities you should use. In general, cs2cs is your best bet since it supports transformations between projected coordinate system as well as datum transforms. If you just need to go to/from a projected coordinate system to geographic without a datum transform, the proj and invproj utilities do an acceptable job.

9.4 More Resources

The U.S. Geological Survey has a "poster" that provides an excellent overview of projections and the characteristic of each. The poster is available for download free of charge.[5]

Published in 1987, the USGS Professional Paper, "Map Projections: A Working Manual" provides a good overview of projections and includes the mathematical formulae needed to transform more than twenty-five different coordinate systems. The paper is available online.[6]

You might find it hard to believe, but an animated movie made in 1947 provides an excellent beginner's introduction to the problems of portraying a spherical earth on a flat surface. Entitled *The Impossible Map*, the movie is available online from the National Film Board of Canada.[7]

The American Society for Photogrammetry & Remote Sensing maintains a repository of the "Grids and Datums" column from each issue of its journal.[8] Datums and grids for a number of regions around the world are documented in each column since 1998.

Lastly, using your favorite search engine on the term *map projections* will bring up enough reading material to keep you busy for a while. Becoming an expert on projections takes some time and effort; learning enough to be proficient with your data is as simple as being able to identify what you have and making your software work for you.

5. Download it from http://erg.usgs.gov/isb/pubs/MapProjections/projections.pdf. You can also find a copy of the map projections poster in PDF at http://desktopgisbook.com/projections.
6. http://pubs.er.usgs.gov/usgspubs/pp/pp1395
7. http://www.nfb.ca/animation/objanim/en/films (sort by title to find it)
8. http://www.asprs.org/resources/grids

Chapter 10

Geoprocessing

It's often not enough to have data and look at it. We almost always want to do some sort of manipulation or processing. This is where geoprocessing comes in. We'll use the broad definition of geoprocessing to include any kind of data manipulation and analysis. To some extent, you could consider importing data as a geoprocessing operation. In this chapter, we'll look at some other operations to include the following:

- Projection
- Line-of-sight analysis
- Watershed modeling
- Hillshading
- Clipping features
- More complex importing
- Grid algebra

If you are wondering what tools are available in the OSGIS stack to accomplish these tasks, for the most part you'll find the answer is GRASS. Although some of the other desktop tools provide various levels of support for a few types of operations, for the most part you'll need GRASS. The goal in this chapter is not to instruct you in the use of all the GRASS geoprocessing tools but rather to introduce you by way of example to the possibilities.

10.1 Projecting Data

That's right—we're going to take one more look at projecting data, even though we've given it a good going over in Chapter 9, *Projections and Coordinate Systems*, on page 138. Once again, depending on how you view and use your GIS data, you may find you need to project the data to another coordinate system to make your life easier. Some of the desktop GIS applications support "on-the-fly" projection of data. This means you set the default projection for the map, and every layer that is added is reprojected to that coordinate system, assuming it's different from the default. This is a two-edged sword. On one hand, it's very convenient. On the other, there is a performance penalty in that every point or vertex must be transformed as the features are drawn. The size of the penalty depends on a number of factors and really can't be generalized; however, suffice it to say that as long as you're transforming coordinates on the fly, it's going to be slower.

As we've seen, the solution is to transform the data to a coordinate system that is suitable for your project work. In Section 11.2, *Coordinate System Conversion*, on page 194, you'll see how to use ogr2ogr to transform OGR-supported layers. In this section, we'll look at how to change the coordinate system of our data using GRASS.

GRASS has two commands that are used to project data: r.proj for raster maps and v.proj for vector maps. The key to projecting data is having your locations properly defined and set up. If you need a hand getting started with GRASS locations and mapsets, take a look at Section C.1, *Location, Location, Location*, on page 296.

When you project a map, it ends up in your *current* location and mapset. You could think of it as copying the map from its original location to your current one, transforming the coordinates as it goes. You don't specify any projection parameters when projecting GRASS maps, since all keys on the locations involved and locations always have a projection/coordinate system defined.

Let's look at the usage for projecting a vector map using v.proj:

```
> v.proj help

Description:
 Allows projection conversion of vector files.

Keywords:
 vector
```

```
Usage:
 v.proj [-lz] input=name location=string [mapset=string]
   [dbase=string] [output=name] [--overwrite]

Flags:
  -l   List vector files in input location and exit (a dummy value
       must be given for input)
  -z   (3-D vectors only) Assume z co-ordinate is ellipsoidal
       height and transform if possible
 --o   Force overwrite of output files

Parameters:
     input   Name of input vector map
  location   Location containing input vector map
    mapset   Mapset containing input vector map
     dbase   Path to GRASS database of input location
    output   Name for output vector map
```

As you can see, there aren't many required parameters. You have to know where the map resides, including the full path to the GRASS database if it's in a different database than the target location. Usually you get away with just entering the map name and location and specifying the output name. As usual, GRASS gives you an --overwrite option in case you need to run the command more than once to get it right. We have a lakes map in our alaska_albers location that we can play with. Let's project it to our UTM Zone 6 location alaska_utm6_nad27.[1] We begin by starting GRASS in the location where we want the projected map to reside, in this case our UTM location, and then proceeding:

```
GRASS 6.3.cvs (alaska_utm6_nad27):~ > v.proj input=lakes \
  location=alaska_albers mapset=PERMANENT output=lakes

Input Projection Parameters: +proj=aea +lat_1=55 +lat_2=65 +lat_0=50
+lon_0=-154 +x_0=0 +y_0=0 +no_defs +a=6378206.4 +rf=294.9786982
+nadgrids=/usr/local/grass-6.3.cvs/etc/nad/alaska
Input Unit Factor: 1

Output Projection Parameters: +proj=utm +zone=6 +a=6378206.4
+rf=294.9786982 +no_defs
+nadgrids=/usr/local/grass-6.3.cvs/etc/nad/alaska
Output Unit Factor: 1
Re-projecting vector map...
Building topology ...
177 primitives registered
Building areas:  100%
87 areas built
```

1. Projecting a region the size of Alaska into a single UTM zone is not really appropriate, but it illustrates the process of projecting vectors with GRASS.

```
87 isles built
Attaching islands:  100%
Attaching centroids:  100%
Topology was built.
Number of nodes      :   178
Number of primitives:   177
Number of points     :   0
Number of lines      :   0
Number of boundaries:   89
Number of centroids :   88
Number of areas      :   87
Number of isles      :   87
Number of incorrect boundaries   :   2
Number of duplicate centroids    :   1
GRASS 6.3.cvs (alaska_utm6_nad27):~ >
```

We specified the name, location, and mapset for the map (layer) we wanted to project and specified the output name to be lakes. GRASS gives us a whole bunch of information as it proceeds with the projection process. Once complete, we have our new lakes layer projected to UTM Zone 6—we'll use v.info to see what we created:

```
GRASS 6.3.cvs (alaska_utm6_nad27):~ > v.info map=lakes
+----------------------------------------------------------------------------+
| Layer:    lakes                          Organization:                     |
| Mapset:   gsherman                       Source Date:                      |
| Location: alaska_utm6_nad27              Name of creator:                  |
| Database: /home/gsherman/grassdata                                         |
| Title:                                                                     |
| Map Scale:  1:1                                                            |
| Map format: native                                                         |
|----------------------------------------------------------------------------|
|   Type of Map:  Vector (level: 2)                                          |
|                                                                            |
|   Number of points:       0                Number of areas:      87        |
|   Number of lines:        0                Number of islands:    87        |
|   Number of boundaries:   89               Number of faces:      0         |
|   Number of centroids:    88               Number of kernels:    0         |
|                                                                            |
|   Map is 3D:              0                                                |
|   Number of dblinks:      1                                                |
|                                                                            |
|   Projection: UTM (zone 6)                                                 |
|            N: 12862988.157    S: 0.000                                     |
|            E: 420703.718    W: -2476603.714                                |
|            B: 0.000         T: 0.000                                       |
|                                                                            |
|   Digitize threshold: 0.00000                                              |
|   Comments:                                                                |
|                                                                            |
+----------------------------------------------------------------------------+
```

You can see once you have your locations set up in GRASS, projecting to a new coordinate system is fairly simple. Using the same methodology, you can also project rasters using r.proj, which we'll do as part of our next geoprocessing task. Let's move on to some topics that are more along the lines of "classic" geoprocessing.

10.2 Line-of-Sight Analysis

Line-of-sight analysis (LOS) is interesting from a curiosity standpoint as well as for hard analysis. Suppose you want to know what you can see from the top of the local mountain (assuming you don't live in Kansas). With the right data to work with, LOS analysis can show you all the areas that are visible from a given point on the map. Some practical applications are determining the visibility of features in site planning. Can the new garbage dump be seen from the local park? How many people will be able to see the new 75-foot-tall monster transmission tower (I have a new one in my backyard)? When doing LOS analysis, we can specify not only the location to view from but also the height of the observer (that would be us). Let's take a look at a simple LOS example.

In our example, we will use the GRASS r.los command to create a viewshed (area we can see) from a given point. To do the analysis, we need a raster dataset that has elevation information. A couple of examples are the USGS Digital Elevation Model (DEM) product and the National Elevation Dataset (NED).[2]

We will use a 1:63,360 DEM (that's 1 inch = 1 mile) in our LOS analysis. The steps to get from a raw DEM to our LOS viewshed are as follows:

1. Download the DEM.
2. Import the DEM into our world latitude-longitude mapset.
3. Project the DEM into the Albers coordinate system.
4. Use r.los to do the LOS analysis.
5. Use r.mapcalc to set unwanted values to null.
6. Display the results.

---

2. You can download the NED data from http://seamless.usgs.gov using an interactive web map interface to select your area of interest. You can find links to a good number of the datasets offered by USGS at http://edc.usgs.gov/geodata.

Getting the DEM

The first task is to get the DEM and get it ready for use in the analysis. The DEM we chose is for the Anchorage C6 quadrangle in Alaska.[3] The file (ancc6.gz) came gzipped, so before it can be imported, it must be unzipped. We used gzip -d to unzip it and then renamed it to ancc6.dem. If we wanted to, we could view it right now using QGIS because it supports USGS ASCII DEMs. To import it into GRASS, we need a geographic location since the DEMs coordinates are in degrees of latitude and longitude. The datum of the location must match the DEM as well. In the case of our DEM, that's NAD 27. If you can't remember the gory details of creating a new GRASS location, refer to Section C.1, *Location, Location, Location*, on page 296.

To import the DEM into our geographic location, from the GRASS shell we use the following:

```
r.in.gdal input=ancc6.dem output=ancc6_dem title="Anchorage C6 DEM"
```

Now we need to project the DEM to our Albers coordinate system. You might be asking why we have to project it. Well, the answer is, r.los doesn't work with geographic coordinates. If you try it, you'll get a nice message along the lines of this:

```
ERROR: Lat/Long support is not (yet) implemented for this module.
```

To project the DEM, we use the r.proj command. But first we need to have an Alaska Albers location created using the proper parameters. For this example, we created one and set the default region (part of the creation process) to just the area of our DEM. If you need to know the parameters for a location, you can use g.proj -p from the GRASS shell to print the projection information for the current location.

```
> g.proj -p
-PROJ_INFO-------------------------------------------------
name       : Albers Equal Area
proj       : aea
datum      : nad27
a          : 6378206.4
es         : 0.006768657997291279
lat_1      : 55
lat_2      : 65
lat_0      : 50
lon_0      : -154
x_0        : 0
y_0        : 0
```

3. You can download this DEM at http://agdc.usgs.gov/data/usgs/geodata/dem/63K/demlist_A.htm .

```
no_defs     : defined
-PROJ_UNITS------------------------------------------------
unit        : meter
units       : meters
meters      : 1
```

To project the DEM, we start GRASS in the target location (in this case Alaska Albers) and use the r.proj command:

```
r.proj input=ancc6_dem location=world_lat_lon_nad27 output=ancc6_dem
```

Notice we specify the source location where the geographic version of the DEM resides. Now we have the DEM ready to use in our analysis.

Doing the Line-of-Sight Analysis

Now that the data is in order, we can get down to doing some analysis. To use r.los, we need to now where the observer is located in map coordinates, as well as the maximum distance we want the analysis to consider. We can also apply a height to the observer if we desire. For now, we'll do a simple analysis using a point located on a gravel bar in the river bottom to determine what we can see. To get the coordinates for the observer (that's us standing on the gravel bar), we just used QGIS or GRASS to get the location of the mouse cursor in map units. With that, we can run r.los:

```
r.los input=ancc6_dem output=los_river coordinate=259315,1307037 max_dist=3000
```

This gives us a line-of-site analysis extending 3,000 meters from our location. Depending on your version of GRASS, r.los may set any cell outside the maximum distance to a value of zero, which means that when we overlay the results on the DEM or other background layers, our underlying layers are not visible.[4] Fortunately, there are a couple of ways to fix this problem.

The GRASS r.mapcalc command allows you to do arithmetic operations on the cells in raster layers. A full range of operators and functions is supported. In our case, we want to do something pretty simple—set the cells outside our analysis area to null values. This will allow our background layers to show through. To do this, we will create a new map from the results of r.los. The command is simply as follows:

```
r.mapcalc 'los_river_nulls=if(los_river==0,null(),los_river)'
```

4. As of this writing, GRASS 6.3 release candidate 3 does not have this issue.

The r.mapcalc creates a new raster map named los_river_nulls using the output from the r.los command. The **if** statement says, "If a cell has a value of 0, set it to null; otherwise, set it to its current value." This gives us a raster with all the values outside our analysis area set to null, and we can now see the results with our background layers showing through. In Figure 10.1, on the following page, you can see the results of the analysis, with those areas that we can see from our position (the red circle) shown in light blue. We've overlaid the LOS results on our DRG so we can compare the analysis with the topography. As you might expect, our line of sight is somewhat limited when standing in the river bottom. We can't see very far to the west, basically just along the top of the river bluff. We can see upstream and downstream a fair bit, as well as to the east, which is on the inside of the river bend and consequently doesn't have a high bank.

If we didn't want to create a new layer, it turns out there is an easier way for us to set those pesky 0s to null:

```
r.null map=los_river setnull=0
```

That's it—r.null does the trick. Now the LOS result map has nulls properly set and can be displayed directly. So, why did we go to all the trouble to use r.mapcalc? It's mainly to introduce the concept of map algebra, which we'll look at in a bit more depth later.

To test the LOS ability of GRASS, we took a simple example from the river bottom. We hope that was enough for you to see the power of this type of analysis. We also used a bit of map algebra to tweak the output and make it more appealing when displayed with the background layers. We'll continue our river theme in the next section by looking at the watershed modeling tools in GRASS.

10.3 Hydrologic Modeling

GRASS includes a number of modules for hydrologic modeling, including modules to create and analyze watershed basins, carve out streams in a DEM, trace a drainage path, simulate flooding, and perform a host of other functions. These modules provide a sophisticated toolset for your hydrologic modeling needs.

To illustrate one of these tools, we'll take a simple example and raise the sea level by 100 meters using the Ancc6_dem DEM. The r.lake module allows you to create a new raster map portraying the filling of an area

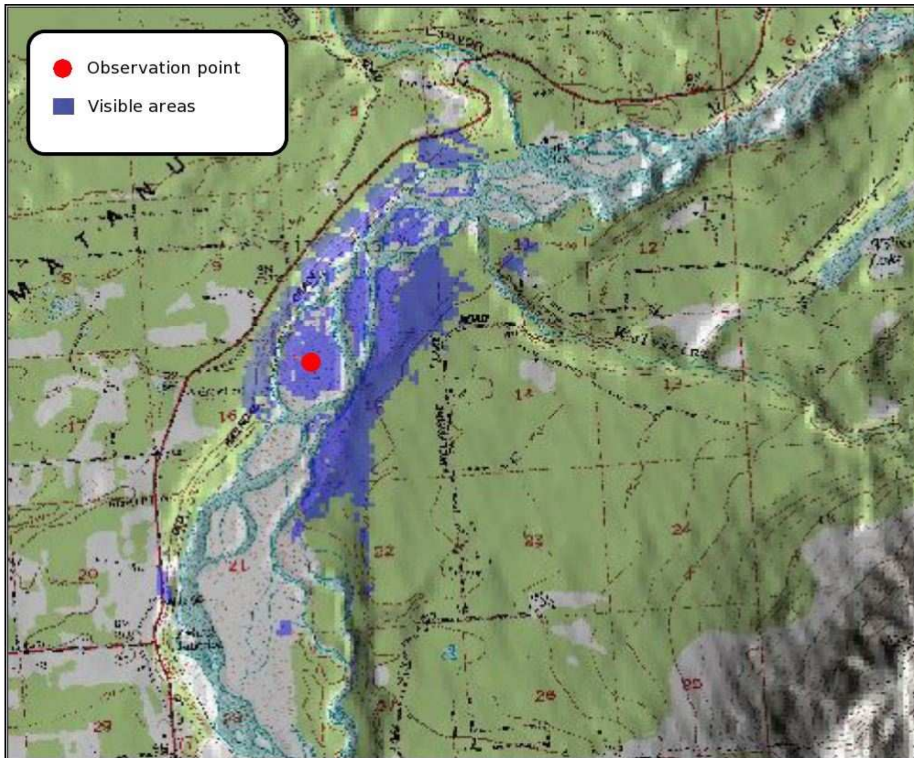


Figure 10.1: Results of line-of-sight analysis in GRASS

on a DEM to a given level. You just specify the start point and the water level. You don't have to be real picky about the start point because GRASS analyzes the DEM and fills it such that the deepest point will be equal to the depth you chose. This means you can actually pick what will become a very shallow area (in other words, a higher elevation), and the lake will be created properly. After all, water does flow downhill.

First let's look at the usage for r.lake:

```
GRASS 6.2.2 (albers_c6):~ > r.lake help

Description:
 Fills lake from seed at given level

Keywords:
 raster
```

```
Usage:
 r.lake [-no] dem=name wl=value [lake=name] [xy=east,north]
   [seed=name] [--overwrite]

Flags:
  -n   Use negative depth values for lake raster map
  -o   Overwrite seed map with result (lake) map
 --o   Force overwrite of output files

Parameters:
   dem   Terrain raster map (DEM)
    wl   Water level
  lake   Output raster map with lake
    xy   Seed point coordinates
  seed   Raster map with seed (at least 1 cell > 0)
```

The command is pretty straightforward. Notice that we can choose to assign negative values to the lake map using the -n option. This means that if we query a given cell, the value will be negative, indicating a depth from the surface of the lake.

The seed coordinates specify the starting point of the calculations. We could use a raster map as a seed as long as it has one cell with a value greater than zero. Why would we want to do this? If we wanted to create a series of maps showing an increasing water level, we could use the previous output as the seed for the next map. Another important point is that the water level must be specified in DEM units—in our case, meters.

To flood our DEM, we pick a point in the southwest corner somewhere and use the following command:

```
r.lake dem=ancc6_dem lake=ancc6_lake_100m xy=258686.903427,1298819.69314 wl=100
```

This creates a new map named ancc6_lake_100m, shown in Figure 10.2, on the next page. Each flooded cell in the raster has a value indicating the depth, while those that are above water are set to null. This means the underlying layer(s) on our map are visible so we can see what land remains.

If you look carefully at the newly create lake, you can see the original river course underneath. It flows from the top center of the map down and then to the west. This was the extent of the water before we flooded the area, apart from a few lakes in the southwest quadrant of the map that are now completely underwater. You can see from the result that raising sea level 100 meters isn't a good thing. We flooded several lakes, along with a bunch of subdivisions and a town or two.

Figure 10.2: Raising sea level by 100 meters

Using r.lake is a simple example of a pretty powerful tool in the GRASS hydrologic modeling toolbox. If you use your imagination, you could combine this tool with a bit of shell script to loop through multiple iterations of rising sea level, saving each image using r.out.mpeg to create an animation. But we'll leave that exercise to you.

10.4 Creating Hillshades

You've no doubt seen those fancy shaded relief maps. Now we are going to see how to create one from a DEM using the GRASS r.shaded.relief module. Again we'll use the Ancc6_dem DEM as the starting point. First let's get a look at the usage and options for r.shaded.relief.

```
GRASS 6.2.2 (albers_c6):~ > r.shaded.relief help

Description:
 Creates shaded relief map from an elevation map (DEM).

Keywords:
 raster, elevation

Usage:
 r.shaded.relief map=string [shadedmap=string] [altitude=value]
   [azimuth=value] [zmult=value] [scale=value] [units=string]
   [--overwrite]

Flags:
 --o   Force overwrite of output files

Parameters:
        map   Input elevation map
  shadedmap   Output shaded relief map name
   altitude   Altitude of the sun in degrees above the horizon
              options: 0-90
              default: 30
    azimuth   Azimuth of the sun in degrees to the east of north
              options: 0-360
              default: 270
      zmult   Factor for exaggerating relief
              default: 1
      scale   Scale factor for converting horizontal units to elevation units
              default: 1
      units   Set scaling factor (applies to lat./long. locations only)
              options: meters,feet
```

The command is pretty straightforward and has several options for creating the shaded relief map from the DEM, including sun angle and altitude. We can also exaggerate the relief to get a more dramatic effect. We'll start simple and create a standard shaded relief map:

```
GRASS 6.2.2 (albers_c6):~ > r.shaded.relief map=ancc6_dem \
  shadedmap=ancc6_shade1
Calculating shading, please stand by.
 100%
Color table for [ancc6_shade1] set to grey

Shaded relief map created and named [ancc6_shade1].
```

This map is all default settings. Generally, the default light settings (altitude and azimuth) produce good results, unless you have particular needs. Notice that when we created the hillshade, it set the color to gray. This is because r.shaded.relief is actually a shell script that runs both r.mapcalc and r.colors to create the hillshade. Let's try one with

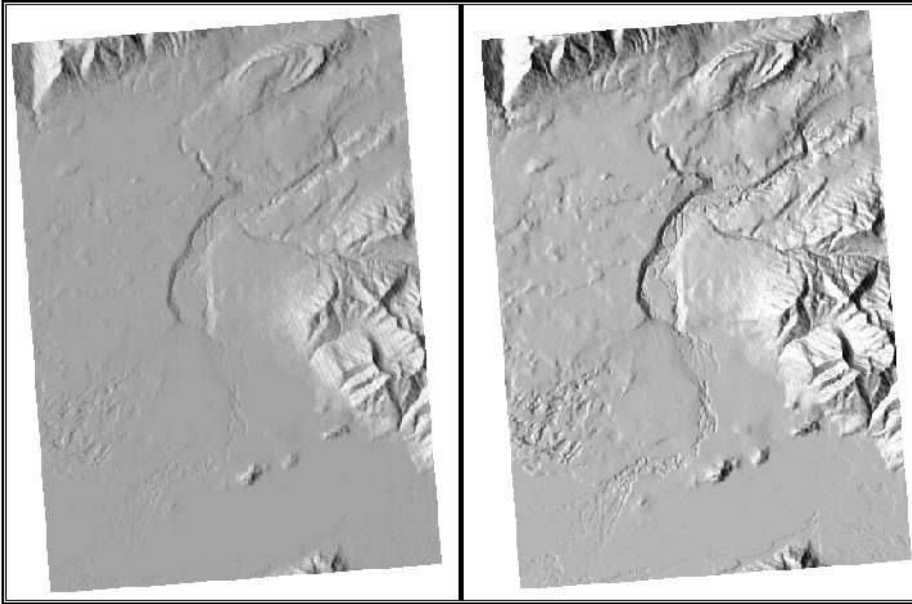


Figure 10.3: Hillshade with no exaggeration (left) and 4X exaggeration (right)

some vertical exaggeration, and then we'll compare:

```
GRASS 6.2.2 (albers_c6):~ >  r.shaded.relief map=ancc6_dem \
  shadedmap=ancc6_shade2 zmult=4
Calculating shading, please stand by.
 100%
Color table for [ancc6_shade2] set to grey

Shaded relief map created and named [ancc6_shade2].
```

Now we have an "out-of-the-box" hillshade and one with exaggerated relief (four times). In Figure 10.3, you can see the two side by side, with the default hillshade on the left and the 4X exaggeration on the right. I'll leave it up to you to decide which looks better.

Colorizing the Hillshade

A gray hillshade is nice but a bit boring. Let's look at how to make our hillshade nicely colored and even export it to a georeferenced TIFF. The process consists of two steps: colorizing the DEM and combining it with the hillshade to make the final product.

To begin, we will create a rules file to define colors by percentage of the range of elevations in the raster. The rules file specifies percentages and a color in RGB notation and is taken directly from the manual page for r.colors.

```
0% 0 230 0
20% 0 160 0
35% 50 130 0
55% 120 100 30
75% 120 130 40
90% 170 160 50
100% 255 255 100
```

We save this as myelevation.rules and will use it in a minute. To create the colored hillshade, we set the colors for the DEM to those in the rules file we just built and then do some magic with the r.his and r.composite commands. Here is a script that does the whole process for us:

```
Line 1  #!/bin/sh
-       r.shaded.relief map=ancc6_dem shadedmap=ancc6_shade zmult=4 --overwrite
-       cat myelevation.rules |r.colors map=ancc6_dem color=rules
-       r.his -n h_map=ancc6_dem i_map=ancc6_shade r_map=ancc6_r g_map=ancc6_g \
5         b_map=ancc6_b --overwrite
-       r.composite -d red=ancc6_r blue=ancc6_b green=ancc6_g output=ancc6_comb \
-         --overwrite
```

Let's take a look at each line of the script to see what it does. Line 2 creates the shaded relief map from the DEM, with a vertical exaggeration of 4. We specified the --overwrite option (you can also use --o) to allow us to run the script multiple times if need be, replacing the existing shaded relief map each time.

On line 3, we apply the color rules to the DEM by piping the contents of the rules file to the r.colors command. Now for the tricky part.

Next we use r.his to create red, green, and blue maps from the DEM and shaded relief as shown on line 4. The hue is taken from the input DEM and specified with the h_map parameter. This sets the color for each cell. The intensity or brightness of each cell is set from the shaded relief map using the i_map parameter. The remainder of line 4 contains the names for the output of the red, green, and blue maps.

Finally on line 6, we put the RGB maps back together with r.composite to create the final map named ancc6_comb. Now we have a nicely colored hillshade, as shown in Figure 10.4, on the next page.

You may be wondering about the value of the composite map we just created. Apart from displaying it in GRASS or QGIS, we can export it

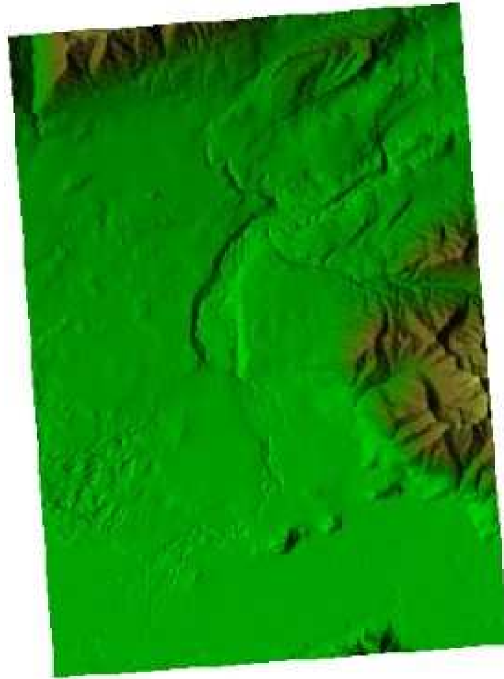


Figure 10.4: Colored shaded relief map created with GRASS

to other formats supported by the r.out.* suite of commands in GRASS. This includes creating a georeferenced TIFF that can be used with other GIS software for those poor folks not using GRASS and/or QGIS. Let's export the colored hillshade to a georeferenced TIFF that we can share:

```
r.out.tiff -t input=ancc6_comb output=ancc6.tif compression=packbit
```

The -t switch tells r.out.tiff to create a world file along with the TIFF. If we use gdalinfo on the newly created file, we can see exactly what we created.

```
> gdalinfo ancc6.tif
Driver: GTiff/GeoTIFF
Size is 410, 551
Coordinate System is `'
Origin = (249831.465000,1318446.415363)
Pixel Size = (53.32700000,-53.30280132)
Image Structure Metadata:
  COMPRESSION=PACKBITS
Corner Coordinates:
```

```
Upper Left  (  249831.465, 1318446.415)
Lower Left  (  249831.465, 1289076.572)
Upper Right (  271695.535, 1318446.415)
Lower Right (  271695.535, 1289076.572)
Center      (  260763.500, 1303761.494)
Band 1 Block=410x6 Type=Byte, ColorInterp=Red
Band 2 Block=410x6 Type=Byte, ColorInterp=Green
Band 3 Block=410x6 Type=Byte, ColorInterp=Blue
```

We can see from the gdalinfo output that our new TIFF is a three-band image and compressed with packbits compression. That's good, because it's what we specified when we created the image. Notice what's missing—there is no coordinate system defined for the image. When you export an image using r.out.tiff, it doesn't encode the coordinate system information into the TIFF. We could add this using gdal_translate. For more information on gdal_translate and friends, see Section 11.2, *Using GDAL and OGR*, on page 186.

When you go to share your georeferenced hillshade maps with the rest of the world, make sure to include the world file. Otherwise, it may fall off the face of the earth when your friends attempt to display it with the rest of their data.

10.5 Merging Digital Elevation Models

In this section we'll look at how to merge DEMs to create a single map layer in GRASS, and there is a very good reason to do so. As you look around the Internet, you'll find that a lot of the available data (DEMs included) is tiled. This means you have to download more than one file to get the complete dataset. Sometimes just having one tile is fine, as long as it covers the area you need. Other times, you might find you need several adjacent tiles to get the coverage you want. To illustrate the process, we'll merge several GTOPO30[5] DEMs into a single layer.

To begin, we fetched all the DEMs for the Americas and stashed them in a directory. The DEM files are distributed in a tar-gzipped format so you'll have to unpack them before proceeding. On Linux or OS X you can just use this:

```
tar -xzf w100s10.tar.gz
```

On Windows use a zip file manager that supports tar.gz files such as 7-zip. Before we proceed with the import, we need to edit the .HDR file

5. http://edc.usgs.gov/products/elevation/gtopo30/gtopo30.html

Exporting Rasters from GRASS

For the colorized hillshade, we used r.out.tiff to create a composite image from each of the raster maps (red, green, and blue) created by r.his. Compositing the maps results in some reduction of color, although this likely won't be noticeable to the human eye.

For exporting single band rasters, r.out.gdal is a better choice. This is because a three-band image is always created by r.out.tiff, even though you specify a single GRASS raster as input.

for each DEM, adding a line containing "PIXELTYPE SIGNEDINT." This ensures that the DEMs will be imported correctly (for more information, see the r.in.gdal manual page). Once we have the DEMs all unpacked and the .HDR files properly edited, we can import them into our world_lat_lon location in GRASS using r.in.gdal:

```
r.in.gdal -e input=./gtopo30/W060N40.DEM output=w060n40
r.in.gdal -e input=./gtopo30/W060N90.DEM output=w060n90
r.in.gdal -e input=./gtopo30/W060S10.DEM output=w060s10
r.in.gdal -e input=./gtopo30/W100N40.DEM output=w100n40
r.in.gdal -e input=./gtopo30/W100N90.DEM output=w100n90
r.in.gdal -e input=./gtopo30/W100S10.DEM output=w100s10
r.in.gdal -e input=./gtopo30/W140N40.DEM output=w140n40
r.in.gdal -e input=./gtopo30/W140N90.DEM output=w140n90
r.in.gdal -e input=./gtopo30/W180N90.DEM output=w180n90
```

We could use the DEMs as is, loading each into GRASS or QGIS for display purposes. However, if we want to do some analysis or even create a combined shaded relief map, we need to put them all together. To do this, we use the GRASS r.patch command. The usage for r.patch is pretty simple. All you do is provide a list of input DEMs and a name for the output. There are a couple of caveats, though. First, of course, you have to have a GRASS location for the area of interest, and second, make sure you set your GRASS region to the area covered by the combined DEMs. You can set the region using g.region. To "patch" the DEMs together, we use the following:

```
r.patch input=w060n40,w060n90,w060s10,w100n40,w100n90,w100s10, \
  w140n40,w140n90,w180n90 output=americas_dem
```

We just created a merged DEM named americas_dem, consisting of nine input DEMs. In Figure 10.5, on the next page, you can see the result,

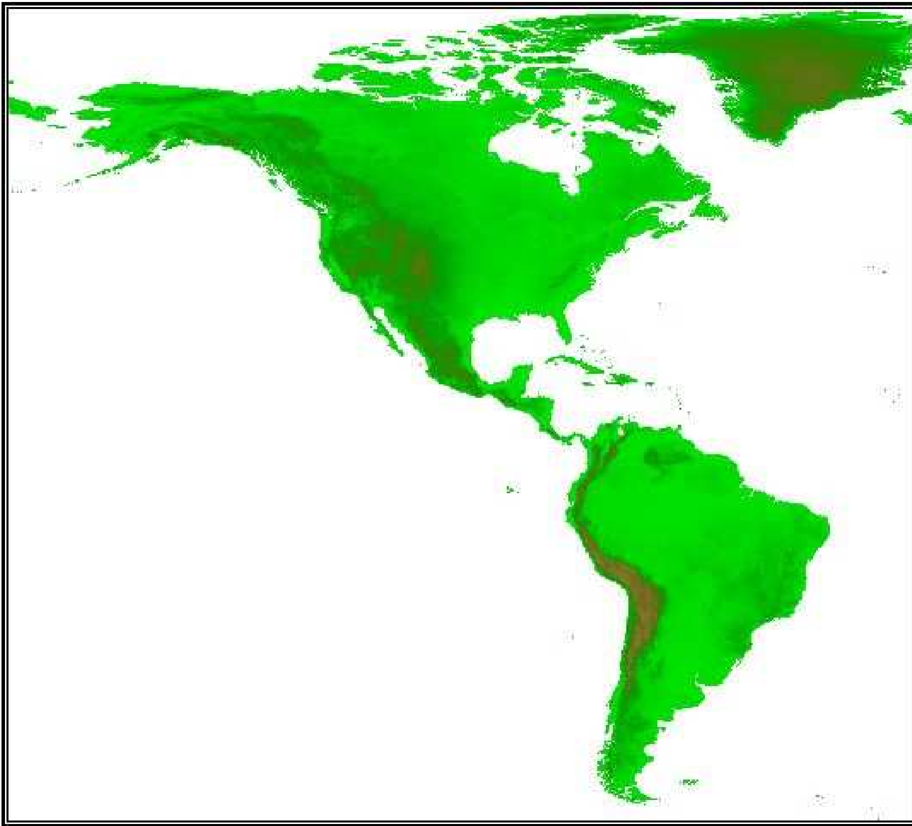


Figure 10.5: Merged GTOPO30 DEM

with the color map set to the same we used in Section 10.4, *Colorizing the Hillshade*, on page 161. We could now create a nice hillshade from the DEM or use it in some sort of analysis, such as melting the Antarctic ice sheet and determining the effect on sea level using r.lake.

10.6 Clipping Features

Sometimes you want to create or modify a dataset by constraining it to an area of interest. We call this *clipping*, and you can do it for both raster and vector datasets using GRASS. In this section, we'll look at clipping the "collars" from a USGS DRG to allow them to display nicely side by side. We'll also look at clipping vector features to create a subset of a larger dataset.

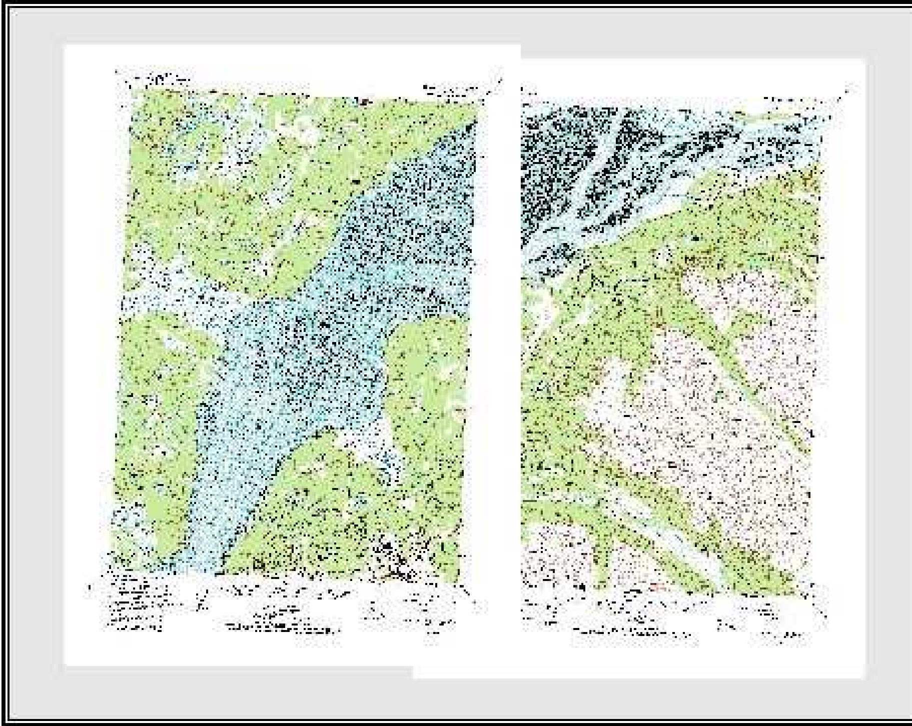


Figure 10.6: Overlapping collars on DRGs

Clipping Rasters with GRASS

If you are wondering why would we want to clip rasters in the first place, let me give you an example. When you download a DRG from the USGS or other source, more than likely it will have collars around the image. A collar is that nice white paper border (well, it would be paper if it wasn't digital) that contains information about the map, including the quadrangle, scale, date published, and other tidbits of information. This is all good information, except when we want to display more than one of these rasters side by side. In that case, we end up with the situation shown in Figure 10.6. The collar of the DRG on top of the map stack blots out information from the DRG below it. To make a seamless data display, we need to remove the collars.

In its original form, a DRG looks just like the paper map you could buy from your local map store. That's because the USGS DRGs are scanned from those original maps and include not only the good stuff

(contours, lakes, rivers, and so forth) but also the metadata as we indicated previously. When you plop them into GRASS, QGIS, or another GIS application, they look just like you threw them down on the kitchen table and tried to match them up. Fortunately, GRASS provides a fairly easy way to make the maps play nicely with each other.

The steps to clip a raster are as follows:

1. Create a vector map to be used as the area of interest.
2. Convert the vector map to a raster map.
3. Use the new raster map as a mask for clipping.
4. Create the newly clipped raster using raster algebra.
5. Clean things up.

Let's work through the process and see whether we can't make our rasters fit together nicely. First we have to import the rasters into GRASS in a proper location. Generally you'll find your DRG is in UTM coordinates. You'll need a GRASS location in the appropriate UTM zone in order to import the raster. If you are fortunate enough to be working with data all in the same zone, then you're all set. If your rasters span UTM zones or you are working on a more regional scale, you may need to project the rasters to a different coordinate system. You can easily do this before you import into GRASS using gdalwarp. For examples of gdalwarp, see Section 11.2, *Using GDAL and OGR*, on page 186.

For the sake of our example, I'm going to project the DRGs to the Alaska Albers projection, since that's where I ultimately want to use them. This will eventually allow me to create a seamless DRG layer for the whole state. To warp the DRGs from UTM Zone 6 to Alaska Albers, I used the following command:

```
gdalwarp -t_srs "+proj=aea +lat_1=55 +lat_2=65 +lat_0=50 +lon_0=-154 \
  +x_0=0 +y_0=0 +ellps=clrk66 +datum=NAD27" i61149c6.tif i61149c6_albers.tif
```

Now you're thinking that doesn't look so simple, but the ugly-looking part of the command comes from the need to specify the projection in proj format. Since the Alaska Albers coordinate system in meters doesn't have an EPSG code, we have to spell it all out. If you're lucky, there will be an EPSG code for your target projection, and you can just use the EPSG:srid notation with gdalwarp to project the raster. For example, had I wanted to use map units of feet, the EPSG projection 2964 is perfect, and the gdalwarp command would have been this:

```
gdalwarp -t_srs EPSG:2964 i61149c6.tif i61149c6_albers.tif
```

We're now ready to import the DRG into our Alaska Albers GRASS location that we already have set up. To do this, we'll use r.in.gdal from the GRASS shell. In fact, this whole process will be done using shell commands rather than the GUI interface. Once we're done, we can check the results by using gis.m in GRASS or by loading the rasters into QGIS. To import the raster, do use:

```
r.in.gdal input=i61149c6_albers.tif output=ancb8_collars
```

Now we have the raster complete with collars in GRASS. The next thing we need is a vector area map that outlines just the "good" portion of the DRG in which we are interested. In most cases, you can find a vector quadrangle boundary layer somewhere on the Internet that is perfectly suited for this task. If not, you'll have to warm up your GRASS digitizing skills and create a new vector map by digitizing the four corners of the DRG. If you do find a vector layer of the quadrangles for your area, you have a bit of work to do as well, since we want only one of the quadrangle polygons. In the case of Alaska, the quadrangle vector map has 3,011 polygons, representing both the 1:250,000 and 1:63,360 scale quadrangles. To clip the DRG, we need to create a new vector map by extracting the quadrangle of interest. We do this using the v.extract command:

```
v.extract input=itma output=ancb8 where="TILE_NAME='ANCB8'"
```

The input map is itma, and it contains the quadrangle boundaries. We want to create a new map named ancb8 with the boundary of the Anchorage B8 quadrangle. Notice the key part of the v.extract command: the where clause. This tells GRASS to extract only features where the attribute TILE_NAME is equal to "ANCB8," giving us a single polygon, which is what we want.

To use the boundary of the quadrangle as a mask, our new vector map has to be converted to a raster using v.to.rast:

```
v.to.rast input=ancb8 output=ancb8_itma use=val
```

This creates a raster map named ancb8_itma that covers the area of the polygon in ancb8. The use=val parameter tells GRASS to set the cells to the value specified by the value parameter. Of course, you noticed that we didn't specify a value parameter. That's because it defaults to 1 if not specified, and this is exactly what we want. If we were to load up the ancb8_itma raster in GRASS or QGIS and look at the cell values, we'd find that they are indeed all set to 1. This is important—when we use this map as a mask, only those cells lying in our area of interest

will have a value of 1. When we get to the final step, this will cause every cell in our DRG outside the bounds of ancb8_itma to be set to null, effectively stripping the collars.

We are now ready to actually do the clipping operation. First we set the GRASS region to that or our ancb8_collars DRG:

```
g.region rast=ancb8_collars
```

We then use the ancb8_itma raster we created from our vector quadrangle boundary as a mask:

```
g.copy ancb8_itma,MASK
```

Now that the mask is set, we use a very simple bit of map algebra to create the clipped DRG:

```
r.mapcalc ancb8=ancb8_collars
```

Notice the r.mapcalc operation looks like it just creates a new raster from every cell in our original DRG. The magic is in the mask, which controls which cells in the new raster are set to the same values as those in the original. Cells outside the mask are set to null. The last step is to remove the mask:

```
g.remove MASK
```

Repeating the process for the adjacent DRG (ANCB7) gives us two rasters that we can now display seamlessly, as shown in Figure 10.7, on the next page. Comparing this to what we started with in Figure 10.6, on page 167, you can see that we have attained success. If we wanted to, we could combine the DRGs into a single raster using r.patch, similar to the method described in Section 10.5, *Merging Digital Elevation Models*, on page 164.

To put it all together, the sequence of commands we used to get from overlapping to seamless nirvana is shown here in the form of a bash script:

```
# import the DRG
r.in.gdal input=i61149c6_albers.tif output=ancb8_collars
# Extract the quad boundary from the boundary map
v.extract input=itma output=ancb8 where="TILE_NAME='ANCB8'"
# Convert the extracted vector quad feature to a raster map
v.to.rast input=ancb8 output=ancb8_itma use=val
# Set the region to operate on to that of our DRG
g.region rast=ancb8_collars
# Set the mask for the operation to the raster created from the
# quad boundary vector
g.copy ancb8_itma,MASK
```

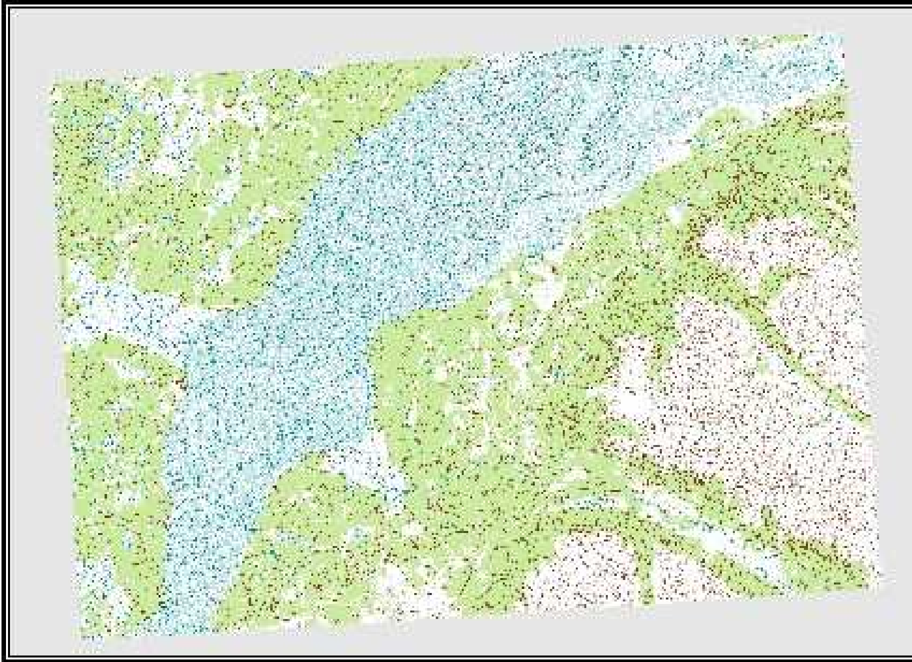


Figure 10.7: Seamless display of clipped DRGs

```
# Use map algebra to create the "clipped" raster
r.mapcalc ancb8=ancb8_collars
# Delete the mask
g.remove MASK
```

Clipping Vectors with GRASS

Clipping a vector map in GRASS is simpler than the raster exercise we just went through. Basically, we need to specify the map we want to clip and the map to be used as the clipping layer. Once we have our data in order, we'll use the v.overlay command to do the work.

In this example, we will clip out the rivers that are contained in a single quadrangle. Our starting situation is shown in Figure 10.8, on the following page, with the quadrangle shown in yellow. The first task is to extract just the TANA5 quadrangle from our itma quadrangle boundary map:

```
v.extract input=itma output=tana5 where="TILE_NAME='TANA5'"
```

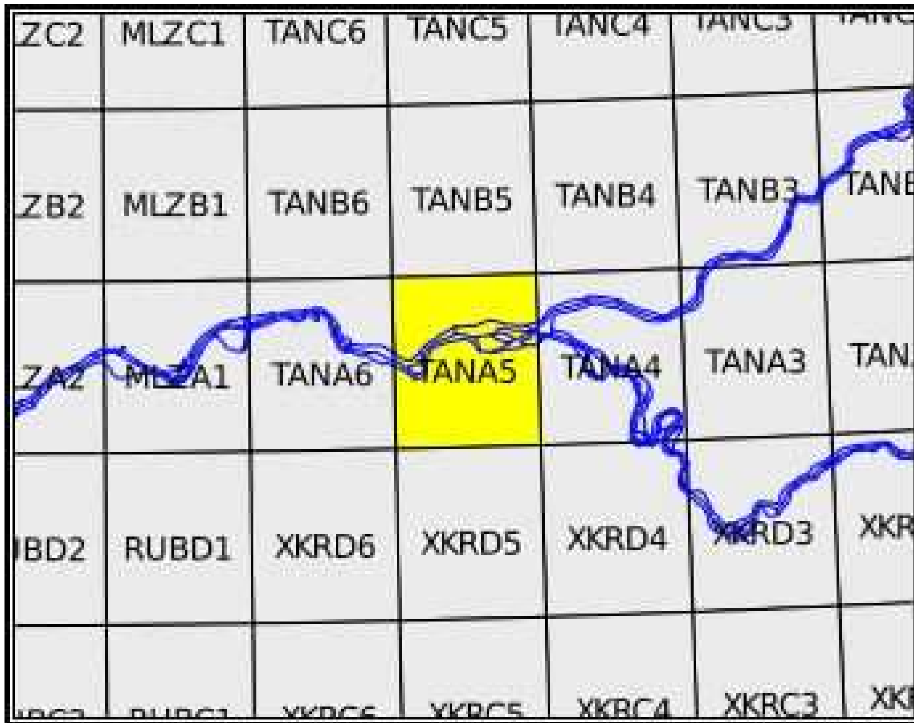


Figure 10.8: Rivers and the quadrangle for clipping

This gives us the new vector map tana5 that contains just the quadrangle of interest. This is the same method we used in our raster clipping process. To clip out the rivers, we simply do an intersection of the two maps using the v.overlay command:

```
v.overlay ainput=majrivers atype=line binput=tana5 operator=and \
  output=majrivers_tana5
```

In Figure 10.9, on the next page, you can see the result of the clipping operation. The rivers that fall within the Tanana A5 quadrangle are all that remain in our new vector layer (majrivers_tana5). We've also included the Tanana A5 quadrangle as a backdrop in the figure. Looking at the v.overlay command, you can see that we specified the majrivers map as the first input and indicated its type using the atype parameter. The tana5 vector map we created using v.extract was specified as the second input map using the binput parameter. The key in this operation is declaring the proper operator (and) since v.overlay has four possibilities.

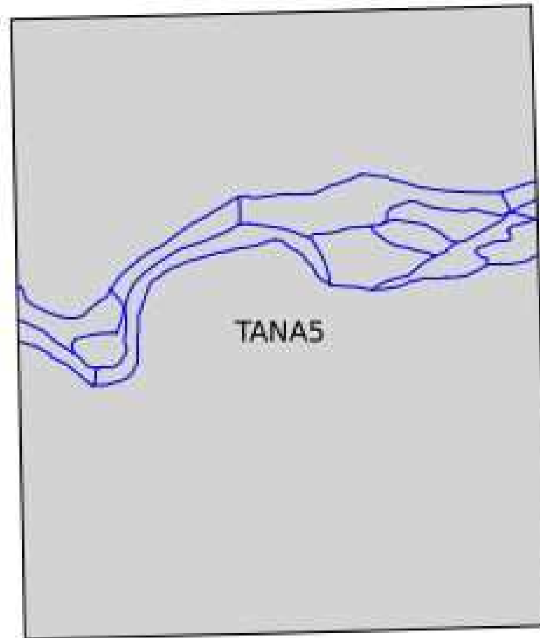


Figure 10.9: Rivers clipped to a quadrangle boundary

Clipping features from larger map layers to create smaller ones is a common GIS operation, especially when your project is focused in a smaller area and you don't need all the extra features running around your map. The GRASS v.overlay command provides a quick and easy way to subset your data into new map layers.

We'll talk about some other vector overlay operations when we get to Chapter 12, *Getting the Most Out of QGIS and GRASS Integration*, on page 208.

Chapter 11

Using Command-Line Tools

Command-line tools provide a powerful way to manipulate data, especially when you want to process them in batches using a script. This chapter describes some of the more common and useful command-line tools and illustrates how to use them to perform common data manipulation, conversion, and map generation tasks. We will take a look at the following:

- Generic mapping tools (GMT)
- Converting and appending data using GDAL/OGR
- PostGIS

11.1 GMT

For a very brief introduction to GMT, see Appendix A, on page 269. In this section, we will take a look at using GMT to create nicely formatted maps for displaying and printing. But before we can do that, we need to make sure you have GMT installed. If not, take a look at Section B.5, *GMT*, on page 293 for some hints to get you started.

The GMT commands create Encapsulated PostScript (EPS) output. If you are using Linux or OS X, you should already have the tools you need to view .eps files. On Windows you will need a viewer that supports EPS. One such viewer is GSview, which allows EPS files to be viewed and printed. For other options, use your favorite Internet search engine to find a suitable application that works for you.

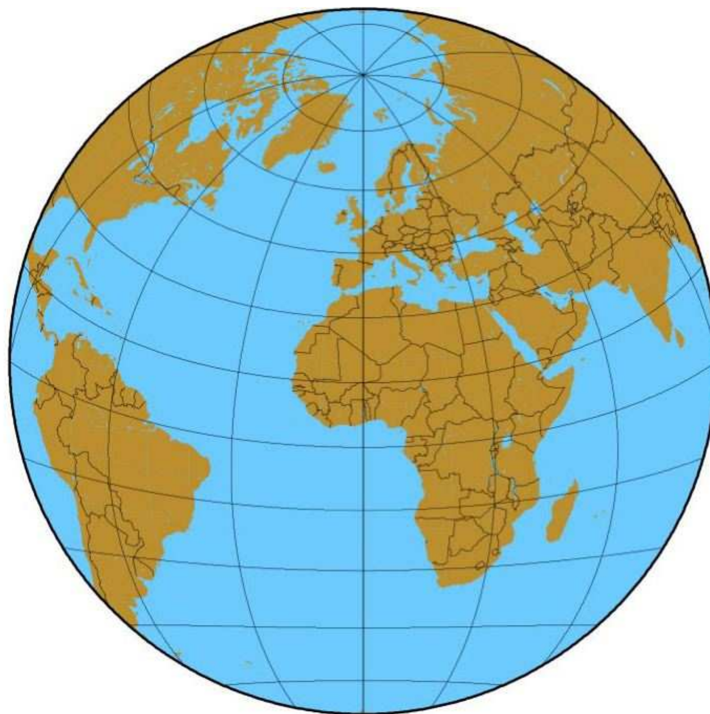


Figure 11.1: Hemisphere view of Earth created with GMT

A Simple GMT Example

To get started, let's take a look at how to generate a simple globe like that shown in Figure 11.1. The code is pretty simple, although it's a bit arcane at first glance:

Download gmt_simple_world.sh

```
pscoast -JA0/20/4.5i -Bg30/g15 -Dl -A2000 -G187/142/46 -S109/202/255 \
  -R0/360/-90/90 -P -N1 > simple_hemi.eps
```

Let's examine the switches used to generate the image. First, GMT's pscoast command requires information about the coordinate system you want to use. This is specified by using the -J switch. GMT supports a nice selection of coordinate systems including the following:

- Albers Conic Equal Area
- Lambert Conic Conformal
- Equidistant Conic

- Lambert Azimuthal Equal Area
- Stereographic Equal Angle
- Orthographic
- Azimuthal Equidistant
- Gnomonic
- Mercator
- Transverse Mercator
- Universal Transverse Mercator
- Oblique Mercator
- Cassini Cylindrical
- Cylindrical Equidistant
- General Cylindrical
- Miller Cylindrical
- Miscellaneous

Each projection has a specific argument that must be supplied to the -J switch. Looking back at the globe example, you'll see that -JA was used to specify the Lambert Azimuthal Equal Area projection. I know because the programs that make up GMT provide a complete description of what's expected as input when you run them with no options. For example, if we enter the pscoast command, we get several screens of options and switches. The first part contains the available projection switches and their syntax:

```
$ pscoast
pscoast 3.4.5 - Plot continents, shorelines, rivers, and borders on maps

usage: pscoast -J<params> -R<west>/<east>/<south>/<north>
  [-A<min_area>[/<min_level>/<max_level>]] [-B<tickinfo>] [-C[<fill>]]
  [-D<resolution>] [-Eaz/el] [-G[<fill>]]
  [-I<feature>[/<pen>]] [-K]
  [-L[f][x]<lon0>/<lat0>/<slat>/<length>[m|n|k]]
  [-M[<flag>]] [-N<feature>[/<pen>]] [-O]
  [-P] [-Q] [-S<fill>]
  [-U[dx/dy/][label]] [-V]
  [-W[<pen>]] [-X<x_shift>]
  [-Y<y_shift>] [-bo[s][<n>]]
  [-c<ncopies>]

  -J Selects map proJection. (<scale> in cm/degree, <mapwidth> in cm)
  -Ja|A<lon0>/<lat0>/<scale (or radius/lat)|mapwidth> (Lambert Azimuthal
        Equal Area)
  -Jb|B<lon0>/<lat0>/<lat1>/<lat2>/<scale|mapwidth> (Albers Equal-Area
              Conic)
  -Jc|C<lon0>/<lat0><scale|mapwidth> (Cassini)
```

```
  -Jd|D<lon0>/<lat0>/<lat1>/<lat2>/<scale|mapwidth> (Equidistant Conic)
  -Je|E<lon0>/<lat0>/<scale (or radius/lat)|mapwidth>  (Azimuthal
        Equidistant)
  -Jf|F<lon0>/<lat0>/<horizon>/<scale (or radius/lat)|mapwidth>  (Gnomonic)
  -Jg|G<lon0>/<lat0>/<scale (or radius/lat)|mapwidth>  (Orthographic)
  -Jh|H<lon0>/<scale|mapwidth> (Hammer-Aitoff)
  -Ji|I<lon0>/<scale|mapwidth> (Sinusoidal)
  -Jj|J<lon0>/<scale|mapwidth> (Miller)
  -Jk|K[f|s]<lon0>/<scale/mapwidth> (Eckert IV (f) or VI (s))
  -Jl|L<lon0>/<lat0>/<lat1>/<lat2>/<scale|mapwidth> (Lambert Conformal
            Conic)
  -Jm|M (Mercator).  Specify one of two definitions:
     -Jm|M<scale|mapwidth>
     -Jm|M<lon0>/<lat0>/<scale|mapwidth>
  -Jn|N<lon0>/<scale|mapwidth> (Robinson projection)
  -Jo|O (Oblique Mercator).  Specify one of three definitions:
     -Jo|Oa<orig_lon>/<orig_lat>/<azimuth>/<scale|mapwidth>
     -Jo|Ob<orig_lon>/<orig_lat>/<b_lon>/<b_lat>/<scale|mapwidth>
     -Jo|Oc<orig_lon>/<orig_lat>/<pole_lon>/<pole_lat>/<scale|mapwidth>
  -Jq|Q<lon0>/<scale|mapwidth> (Equidistant Cylindrical)
  -Jr|R<lon0>/<scale|mapwidth> (Winkel Tripel)
  -Js|S<lon0>/<lat0>/[<slat>/]<scale (or radius/lat)|mapwidth>
          (Stereographic)
  -Jt|T (Transverse Mercator).  Specify one of two definitions:
     -Jt|T<lon0>/<scale|mapwidth>
     -Jt|T<lon0>/<lat0>/<scale|mapwidth>
  -Ju|U<zone>/<scale|mapwidth> (UTM)
  -Jv|V<lon0>/<scale/mapwidth> (van der Grinten)
  -Jw|W<lon0>/<scale|mapwidth> (Mollweide)
  -Jy|Y<lon0>/<lats>/<scale|mapwidth> (Cylindrical Equal-area)
  -Jp|P[a]<scale|mapwidth>[/<origin>] (Polar [azimuth] (theta,radius))
  -Jx|X<x-scale|mapwidth>[l|p<power>][/<y-scale|mapheight>[l|p<power>]]
          (Linear projections)
  (See psbasemap for more details on projection syntax)
```

Each projection requires different parameters. In our example we used -JA0/20/4.5i. This selects the projection (Lambert Azimuthal Equal Area) and sets the longitude to 0 degrees, the latitude to 20 degrees, and the width of the map to 4.5 inches. Note that either -J or -j can be used. Specifying uppercase indicates that the last parameter (in our case 4.5i) is width. Had we used a lowercase *j*, GMT would interpret the last parameter as a scale value. Widths can be specified using c, i, p, or m, which correspond to centimeters, inches, points (1/72 of an inch), and meters.

You can quickly see two things:

- GMT has a lot of options.
- You might want to read the manual.

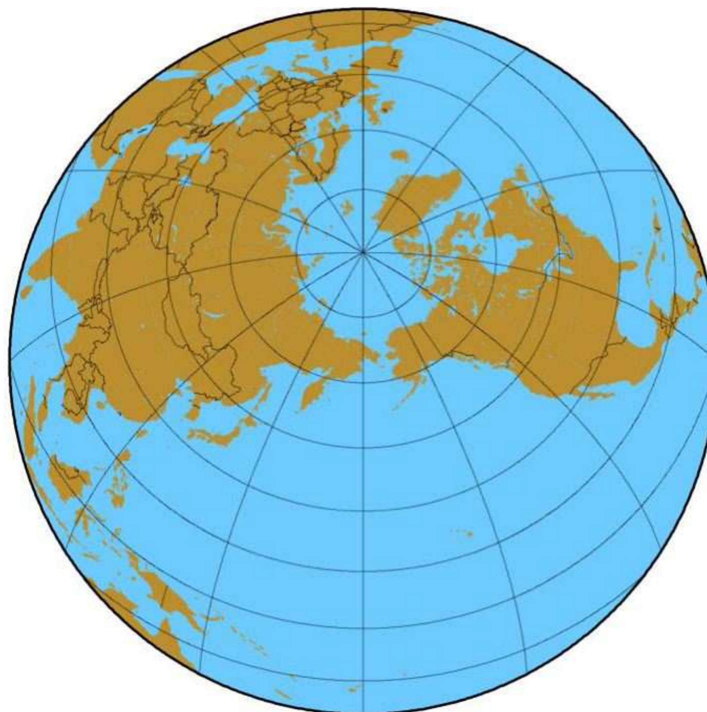


Figure 11.2: Globe centered on 180/65

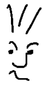
This is an important point—sometimes it takes a bit of digging and looking at examples to find the switches, arguments, or parameters needed to accomplish your goal. Reading the manual is a good place to start.

Let's see what happens if we modify the -J switch a bit. Let's flip the view around 180 degrees and move it closer to the North Pole. To do this, use -JA180/60/4.5i. Leave all the other parameters the same, and run the pscoast command. Our command is now as follows:

Download gmt_simple_world.sh

```
pscoast -JA180/65/4.5i -Bg30/g15 -Dl -A2000 -G187/142/46 -S109/202/255 \
  -R0/360/-90/90 -P -N1 > simple_180_world.eps
```

Looking at Figure 11.2, you see that indeed we are now looking at the International Date Line, and our view is centered at 60 degrees north latitude.



Joe Asks...

Where Does the GMT Data Come From?

GMT actually provides a number of datasets in five resolutions ranging from fine to crude. When installing GMT, you can choose which resolutions you want to include.

There are also tools available to convert other formats for use with GMT. Use your favorite search engine to find tools applicable to your situation.

Let's take a look at the other switches used to create the globe. The `-B` switch defines the intervals for the boundary tick marks. In the globe case, these are the lines of longitude and latitude. The arguments to the `-B` switch indicate a gridline spacing of 30 degrees in the x (longitude) direction and 15 degrees in the y (latitude) direction. Note how the x and y settings are separated by a forward slash.

The `-D` switch selects the resolution of the dataset used in creating the globe. The available choices are `f`, `h`, `i`, `l`, and `c`, which correspond to full, high, intermediate, low, and crude. Some of these options may not be available to you if you didn't install all the data sets with GMT. For the globe, we used the low resolution data set.

To control the display of features, the `-A` switch allows you to specify that features below a certain size not be drawn. In our example, we specified that features with an area greater smaller 2,000 square kilometers should not be displayed.

The fill color used for the countries is specified using the `-G` switch. The color can be specified using RGB notation, a shade of gray, or a pattern. In the globe, we used `187/142/46` to create a light brown color. We could have specified a fill pattern using `-Gp100/30`. This fills the land masses with pattern number 30 at a resolution of 100 dpi. If we want to get the highest possible resolution for the pattern, we can use a resolution of 0. Specifying `-GP` inverts the pattern. GMT has 90 predefined patterns available for your use, and you can find examples of each in the GMT Technical Reference. The same options apply for filling the water areas in GMT, except we use the `-S` switch. There are a number of variations for specifying fill colors, and these are well documented in the GMT manuals and tutorial.

The other major switch used in generating the globe is -R. This specifies the extent of the map we want to generate. In the case of the globe, we obviously wanted the entire planet, so we specified an x range of 0 to 360 degrees and a y range of -90 to 90. The range is specified as west/east/south/north. In our next example, we will use -R to constrain our map to a smaller area.

The other switch of interest is -N1. This tells GMT to draw national boundaries in addition to the coastline. Other arguments to -N allow you to draw state boundaries within the Americas and marine boundaries.

The -P switch simply sets the page orientation to portrait. Landscape is the default.

A Flat Example

Let's shift gears a bit and look at another example of using GMT, this time for a smaller area. For this example, we'll create a map of Alaska and annotate it. As I said before, the -R switch controls the extent of our map. Alaska ranges from about 172 degrees east longitude to 130 degrees west. Using 360 degrees for the entire globe, this translates to a region extending from 172 degrees to 230 degrees.

For the Alaska map, we will use the Albers Equal Area Conic projection. Looking at the syntax for pscoast reveals that this requires the use of the -Jb switch. In this case, we use the lowercase *b* to indicate that we will specify the size of the map using a scale. First let's look at the code in gmt_alaska.sh:

Download gmt_alaska.sh

```
pscoast -Jb-154/50/55/65/1:12000000 -R172/230/51/72 -B10g5/5g5 -W1p/0/0/0 \
  -I1/2p/0/192/255 -I2/2p/0/192/255 -I3/1p/0/192/255 -I4/1p/0/192/255 \
  -G220/220/220 -S0/192/255 -L210/54/54/1000 -P -N1/1p/0/0/0 -Dl \
  >gmt_alaska_coast.eps
```

This looks like quite a complex command, but it's really not too bad once you get past all the numbers and slashes.

Projection

First note we specified the projection using -Jb-154/50/55/65/1:12000000. Let's pick that apart a bit to see what's happening. The Albers projection requires the longitude of the central meridian, the latitude of the origin, and the latitude of the two standard parallels. That's what you see specified as -154/50/55/65. These are the standard values used for the Albers projection in Alaska. You can actually specify any values

you want, but if there is a standard for the area you are mapping, you should use it.

The remaining part of the -Jb switch is the size of the output. In this case, we specified it as a scale of 1:12,000,000. This means that one unit on the map represents 12,000,000 units on the ground (in this case meters). If we just wanted output to fit on a page, we could specify -JB-154/50/55/65/6.0i to get a 6-inch-wide image.

Map Extent

To set the map extent, we use the -R switch. In this case we already determined that Alaska ranges from 172 to 230 degrees longitude and roughly 51 to 72 degrees north latitude. To create the map covering this area, we use -R172/230/51/72.

Grid Lines

In this example, we not only want to draw grid lines but also want to annotate them. This is done using -B10g5/5g5. This tells pscoast to draw grid lines 5 degrees apart for both latitude and longitude. The annotation is drawn at 10 degree intervals for longitude and 5 degree intervals for latitude. If you look at the documentation for pscoast, you will see that the first number after the -B is the annotation interval followed by the grid line interval. This notation gives you a lot of flexibility in drawing and labeling gridlines.

Rivers

To make our map more interesting, we'll add rivers to it. GMT comes with several levels of river detail that are specified with the -I switch. The levels we are using are as follows:

- Permanent major rivers
- Additional major rivers
- Additional rivers
- Minor rivers

Note the -I options we specified in the pscoast command. One is required for each river level we want to include on the map. The first two (major rivers) are drawn using a pen width of 2 (2p), while the third and fourth level are drawn with a width of 1 (1p). We use the same color (0/192/255) for each river. If we wanted to include the intermittent major rivers (fifth level), we would add -I5/1p/0/192/255 to the pscoast command.

Fill Colors

Next we specify the fill colors for the land and water areas using the -G and -S switches and add the RGB values to specify the color. For land we use a light gray with RGB values of 220/220/220. For the water 0/192/255 gives us a nice cyan color. Keep in mind that we could also use a pattern or shade for filling land and water areas.

Scale Bar

A scale bar can easily be added to the map using the -L switch. Scale bars can be simple or fancy. In this case, we'll just create a simple one and place it in an open area on the map. How do we know it's open? Well, part of the process is running pscoast and tweaking the options and then running it again until we get the look we want. To create the scale bar, we need the latitude and longitude of the point where we want to place it. Since scale varies as we move further from the equator, we also specify the latitude at which we want the scale calculated. Lastly, we indicate the length the scale bar should span. The default is kilometers, but you can append m for miles or n for nautical miles. Putting it all together, we have -L210/54/54/1000, which gives a 1,000 km scale bar calculated at 54 degrees north latitude and originating at 210 degrees longitude and 54 degrees latitude.

The Last Bits

The remainder of the command tells pscoast to use portrait mode (-P), draw country boundaries in black using a pen width of 1(-N1/1p/0/0/0), and use the low-resolution data (-Dl). The low-resolution dataset is the default, but we specified it here so you could see the syntax.

The Result

You can see the result of all these command switches and options in Figure 11.3, on the next page. We have a nice map of Alaska, with grid lines, borders, and degree annotations. The land and water is filled as we specified, and the scale bar is sitting nicely in the Gulf of Alaska.

Overlaying Data

Now that we have used most of the common options, let's look at one more example with pscoast. This time we'll generate a world map and overlay point data from a delimited text file. You can take that concept and expand it to create overlays of multiple datasets. In this case, we will overlay the location of registered Quantum GIS users throughout the world.

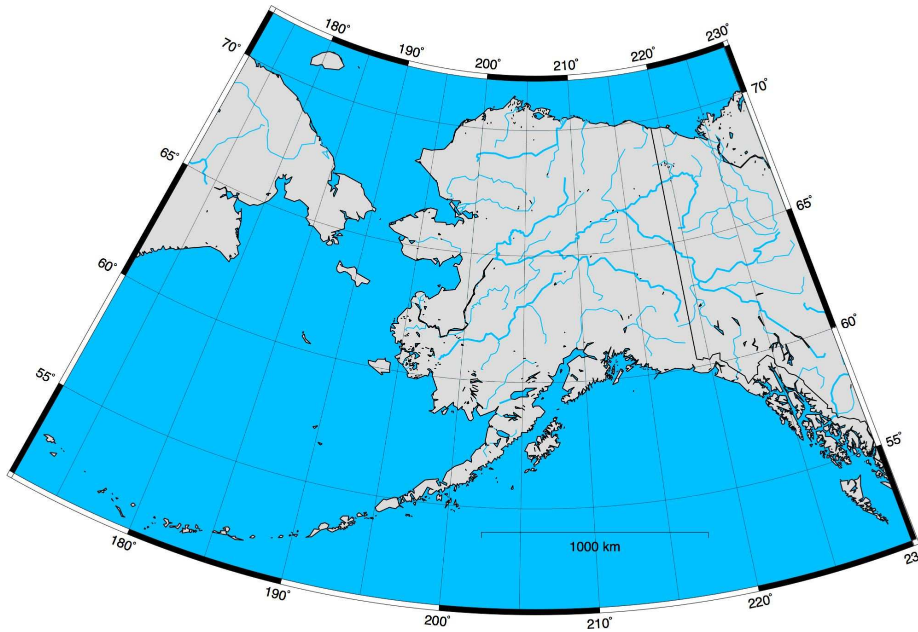


Figure 11.3: Alaska coastline generated with GMT

First let's look at the command to generate the base map:

Download gmt_qgis_users.sh

```
pscoast -JN0/38 -R-180/180/-90/90 -K -W -G220/220/220 -S0/192/255 -N1 \
  -P -B30g5:."Quantum GIS Users": > qgisusers.eps
```

About the only thing new in this command is we added a title to the map by appending a colon and a period to the -B arguments and then the title string. If you are getting the idea the -B switch has lots of permutations, you are correct. Some have called it the most complicated (or confusing) switch in the GMT suite of tools. Fortunately, it's well documented.

Note that we used -JN to specify the Robinson projection, centering the map at 0 degrees longitude with a width of 38 centimeters.

This gives us a Robinson base map of the world with grid lines and annotation of the tick marks. To add an overlay of data, we need to make a couple of modifications to the base map. First we need to specify that we want to be able to write to the output file in "append" mode. This is done using the -K switch. This allows us to overlay the data created with the next command. Without it, our next command would generate

GMT and Multiple Commands

When you want to create a more complex map, you will wind up running multiple GMT commands. The trick is to make sure you specify in the first command that more PostScript code will be appended to the output. Without this, you will end up with every command generating a new page in the output. This can be useful, but when you are trying to create an overlay of multiple commands, it's annoying. Create the base map using the -K switch, and then in subsequent commands include the -O to invoke overlay mode.

a new page in the output, and we would have to hold it up to the window to see the overlay.

To add the overlay, we need two things: a text file containing the coordinates of each point and the psxy command. Here is a snippet from the text file we will use to create the map:

```
-4.0000000000,36.0000000000
-71.9146000000,42.8053750000
-122.7833300000,53.9176000000
5.6667000000,51.9667000000
-0.9519000000,51.4450000000
11.0800000000,46.0400000000
-124.0800000000,40.8800000000
```

These are just x and y values (longitude and latitude) for each point we want to create on the map. To add these, we use psxy, making sure to include the -O switch so the points overlay the base map. The complete code to generate the base map and the overlay is as follows:

Download gmt_qgis_users.sh

```
pscoast -JN0/38 -R-180/180/-90/90 -K -W -G220/220/220 -S0/192/255 -N1 \
  -P -B30g5:."Quantum GIS Users": > qgisusers.eps
psxy qgis_users.txt -JN -O -R -Sc0.15c -G255/0/0 >>qgisusers.eps
```

Note that in the psxy command we didn't need to supply any arguments to the projection or extent switches since they were fully specified when the base map was created. We included the overlay switch and a color for the points using -G. The -Sc switch indicates we want to plot the points using circles. There are several other symbol types you can use including star, bar, diamond, and ellipse.

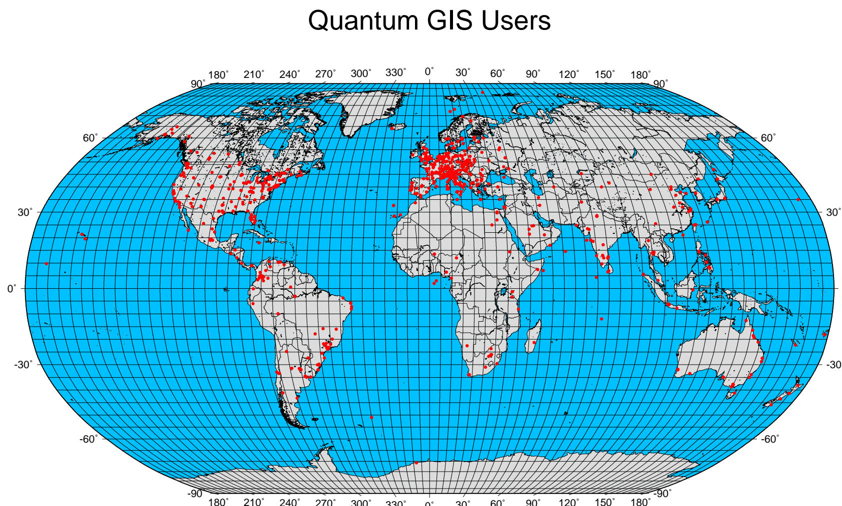


Figure 11.4: Quantum GIS users plotted on a Robinson projection using GMT

We specified the symbol size as 0.15 cm. You should recognize the rest of the parameters from our previous discussion.

The final result is shown in Figure 11.4. This combines the base map with the user data. We could have added other point data by running another psxy command.

As you can see, GMT is a handy tool for creating maps from the command line. We've really only scratched the surface of its capability.

More GMT

GMT has many more features than we have covered—there are sixty-four commands at last count. You should have the basics down, allowing you to venture forward and generate even more impressive maps. Make sure to consult the GMT documentation for additional information, including cookbook recipes and tutorials.

For additional information on using GMT with GRASS, see "Producing Press-Ready Maps with GRASS and GMT" by Dylan Beaudette in *OSGeo Journal*, Volume 1, May 2007.[1]

11.2 Using GDAL and OGR

We have seen examples of the GDAL and OGR utilities previously in several sections. Now we will take a more focused look at the utilities and how they are used. You will quickly see that this set of tools belongs in your toolkit, especially if you plan to do any data manipulation.

If you want a quick overview of the formats supported by GDAL and OGR, as well as a brief summary of each utility, see Section A.2, *GDAL/OGR*, on page 283 in our survey of OSGIS software.

Getting Information

One of the key uses of the GDAL/OGR utilities is getting information about a supported raster or vector file. The commands used are gdalinfo and ogrinfo, respectively. Let's take a better look at each of these utilities.

Vector Information

You download a shapefile from the Internet and unzip it. Now you have a batch of files sitting there (remember, a shapefile consists of at least three files). What attributes does the shapefile contain? What kind of features does it store—points, lines, or polygons? What projection or coordinate system does it use? We are in luck; ogrinfo can answer all those questions for us.

The ogrinfo utility can provide information on both a single layer and all layers in a directory. For example, for a summary of all shapefiles in a directory, we just have to provide the directory name:

```
$ ogrinfo ./desktop_gis_data
INFO: Open of `./desktop_gis_data'
using driver `ESRI Shapefile' successful.
1: cities (Point)
2: AKvolc_v3 (Point)
3: world_borders (Polygon)
```

From the output we can see that the directory desktop_gis_data contains three shapefiles: cities, AKvolc_v3, and world_borders. The results for each

1. http://www.osgeo.org/files/journal/final_pdfs/OSGeo_vol1_GRASS-GMT.pdf

shapefile includes its type. This is good summary information, but what if we want more detail? By specifying the layer name, ogrinfo will give us very detailed information about the layer:

```
$ ogrinfo -so -al ./desktop_gis_data cities
INFO: Open of `./desktop_gis_data'
using driver `ESRI Shapefile' successful.

Layer name: cities
Geometry: Point
Feature Count: 606
Extent: (-165.270004, -53.150002) - (177.130188, 78.199997)
Layer SRS WKT:
GEOGCS["GCS_WGS_1984",
    DATUM["WGS_1984",
        SPHEROID["WGS_1984",6378137,298.257223563]],
    PRIMEM["Greenwich",0],
    UNIT["Degree",0.0174532925199433]]
NAME: String (40.0)
COUNTRY: String (12.0)
POPULATION: Real (11.0)
CAPITAL: String (1.0)
```

The result gives us a detailed summary of information about the cities layer. We can see it is a point layer with 606 features. The coordinate system is WGS84, meaning the coordinates are in latitude and longitude. We also get a summary of the fields and their types, along with the extent of the layer. Armed with this information, we can easily determine whether a layer is suitable for our use and is in an appropriate coordinate system.

Notice the -so switch in the previous example. We used it in combination with the -al switch in order to get detailed information about the layer. The -so switch tells ogrinfo to print a summary only; otherwise, it would also print each record in the shapefile, complete with all the attributes as well as the coordinate information. There are times you may want to view all the information, perhaps dumping it to a text file for further use.

We can use the OGR utilities with more than just shapefiles. To get a quick list of the supported drivers for your installation of OGR, use the --formats switch. The formats you find available will depend on how your version of OGR was compiled. If our version contains support for PostgreSQL, we can get information on layers stored in our PostGIS-enabled database.

```
$ ogrinfo "PG:dbname=gis_data host=madison"
INFO: Open of `PG:dbname=gis_data host=madison'
using driver `PostgreSQL' successful.
1: edit_test (Point)
2: country (Multi Polygon)
3: air_intl_buffer_500k12 (Polygon)
4: bug_test (Polygon)
5: air_intl_buffer_500k14 (Polygon)
6: 64districts (Multi Polygon)
7: 94election (Multi Polygon)
8: admin_nps (Multi Polygon)
9: admin_nra (Multi Polygon)
10: admin_nwr (Multi Polygon)
...
```

An important part of using ogrinfo with PostGIS is the connection string, specified with "PG:" and followed by the appropriate parameters. In our case, we needed only to specify the database name and the host. Depending on how your database authentication is set up, you may need to include "user=" and "password=" (with the appropriate values) in your connection string.

This database contains more than 100 layers, so we truncated the listing—but you get the idea. We didn't have to connect to the database, log in, and issue some SQL to determine what layers were available. We didn't even have to be on the same host as the database in order to get the information—of course this assumes you have a properly set up database with appropriate permissions. Let's get the details for the country layer, remembering to use the -so switch so we don't dump the whole world to our command shell:

```
$ ogrinfo -so -al "PG:dbname=gis_data host=madison" country
INFO: Open of `PG:dbname=gis_data host=madison'
using driver `PostgreSQL' successful.

Layer name: country
Geometry: Multi Polygon
Feature Count: 251
Extent: (-180.000000, -90.000000) - (180.000000, 83.623596)
Layer SRS WKT:
GEOGCS["WGS 84",
    DATUM["WGS_1984",
        SPHEROID["WGS 84",6378137,298.257223563,
            AUTHORITY["EPSG","7030"]],
        AUTHORITY["EPSG","6326"]],
    PRIMEM["Greenwich",0,
        AUTHORITY["EPSG","8901"]],
    UNIT["degree",0.01745329251994328,
        AUTHORITY["EPSG","9122"]],
    AUTHORITY["EPSG","4326"]]
```

```
Geometry Column = shape
cntry_name: String (40.0)
color_map: String (1.0)
curr_code: String (4.0)
curr_type: String (16.0)
fips_cntry: String (2.0)
gid: Integer (0.0)
gmi_cntry: String (3.0)
landlocked: String (1.0)
pop_cntry: Integer (0.0)
sovereign: String (40.0)
```

The output looks similar to that for a shapefile. The thing to note is, in addition to the fields in the layer, ogrinfo also identifies the name of the geometry column for us—in this case it's shape.

Raster Information

For getting information on your rasters, gdalinfo is the tool to use. We introduced this back in Section 4.1, *Viewing Raster Data*, on page 67 where we examined a GeoTIFF. Let's look at some of the options and formats associated with the utility.

To get a list of all the supported formats at your disposal, use the switch --formats. When you do this, you're likely going to get a long list. I won't list all fifty-three of them here, but just the first few as an example:

```
$ gdalinfo --formats
Supported Formats:
  VRT (rw+): Virtual Raster
  GTiff (rw+): GeoTIFF
  NITF (rw+): National Imagery Transmission Format
  HFA (rw+): Erdas Imagine Images (.img)
  SAR_CEOS (ro): CEOS SAR Image
  CEOS (ro): CEOS Image
  ELAS (rw+): ELAS
  AIG (ro): Arc/Info Binary Grid
  AAIGrid (rw): Arc/Info ASCII Grid
  SDTS (ro): SDTS Raster
  DTED (rw): DTED Elevation Raster
  PNG (rw): Portable Network Graphics
  JPEG (rw): JPEG JFIF
```

We can use gdalinfo on files that aren't "strictly" GIS files. For example, here is the output for a JPEG from a digital photo:

```
$ gdalinfo -mm DSCN3898.JPG
Driver: JPEG/JPEG JFIF
Size is 1280, 960
Coordinate System is `'
Metadata:
  EXIF_ImageDescription=
```

```
  EXIF_Make=NIKON
  EXIF_Model=E4300
  EXIF_XResolution=(300)
  EXIF_YResolution=(300)
  EXIF_DateTime=2007:02:17 07:29:59
...
Corner Coordinates:
Upper Left  (    0.0,    0.0)
Lower Left  (    0.0,  960.0)
Upper Right ( 1280.0,    0.0)
Lower Right ( 1280.0,  960.0)
Center      (  640.0,  480.0)
Band 1 Block=1280x1 Type=Byte, ColorInterp=Red
    Computed Min/Max=0.000,255.000
Band 2 Block=1280x1 Type=Byte, ColorInterp=Green
    Computed Min/Max=1.000,255.000
Band 3 Block=1280x1 Type=Byte, ColorInterp=Blue
Computed Min/Max=0.000,255.000
```

There was a lot more metadata in the output, forty-six lines in total. Basically, every EXIF field a camera stores was dumped. The point is, gdalinfo can provide detailed information for the formats it supports—and, yes, just in case you were wondering, you can georeference a JPEG. You won't see any coordinate system information for the digital photo; however, if there was a world file associated with it, the information would be included in the output.

Using gdalinfo is a quick and efficient way to get information on your rasters, without having to open them in your GIS application. For complete information on all the options, see the GDAL documentation.[2]

Converting Data

The GDAL and OGR utilities allow you to convert data between formats, optionally changing some of the characteristics in the process. In this section, we'll look at options and techniques for data conversion, both raster and vector.

Vector Conversion

First off, let's think about why you might want to convert from one vector format to another:

- You have data in a format that isn't usable in your desktop GIS application.
- You need to provide data (to someone or to another application) in a different format than what you are using.

2. http://www.gdal.org/gdalinfo.html

- The data is in the wrong coordinate system or datum.
- You want to create a subset of the data based on a bounding box.
- You want to create a subset of the data based on an attribute query.
- You want to load data into PostgreSQL/PostGIS.

These all sound like good reasons for doing a conversion. Let's look at a few simple examples to get you started with ogr2ogr.

Format Conversion

First let's convert a vector layer from one format to another. In the simplest case, we specify the format we want to convert to, the source layer, and the destination:

```
$ ogr2ogr -f GML cities.gml cities.shp
$ ogrinfo cities.gml
Had to open data source read-only.
INFO: Open of `cities.gml'
      using driver `GML' successful.
      1: cities
```

We just converted the cities.shp shapefile to GML.[3] Notice there was no output or confirmation from ogr2ogr, but the file was created and contains all features in the cities layer in GML. Going the other direction is just as easy:

```
$ ogr2ogr -f "ESRI Shapefile" cities_from_gml.shp cities.gml
$ ogrinfo cities_from_gml.shp
INFO: Open of `cities_from_gml.shp'
      using driver `ESRI Shapefile' successful.
      1: cities_from_gml (Point)
```

Notice that we specify the format we are converting *to* using the -f switch. Remember you can get a list of supported formats passing the --formats switch to ogr2ogr. If the format name contains spaces, you'll have to quote it as we did for the ESRI shapefile conversion.

Data Loading

We can use ogr2ogr to load data into a PostGIS-enabled PostgreSQL database. If you are loading just shapefiles, you could use the shp2pgsql utility that comes with PostGIS. Otherwise, you will find ogr2ogr handy for loading other data types.

---

3. Geographic Markup Language—see http://www.opengeospatial.org/standards/gml.

Let's load the GML file we created into PostgreSQL:

```
$ ogr2ogr -f PostgreSQL -a_srs EPSG:4326 "PG:dbname=gis_data host=madison"\
  cities.gml
$ ogrinfo -so -al "PG:dbname=gis_data host=madison" cities
INFO: Open of `PG:dbname=gis_data host=madison'
      using driver `PostgreSQL' successful.

Layer name: cities
Geometry: Unknown (any)
Feature Count: 606
Extent: (-165.270004, -53.150002) - (177.130188, 78.199997)
Layer SRS WKT:
GEOGCS["WGS 84",
    DATUM["WGS_1984",
        SPHEROID["WGS 84",6378137,298.257223563,
            AUTHORITY["EPSG","7030"]],
        TOWGS84[0,0,0,0,0,0,0],
        AUTHORITY["EPSG","6326"]],
    PRIMEM["Greenwich",0,
        AUTHORITY["EPSG","8901"]],
    UNIT["degree",0.017453292519943328,
        AUTHORITY["EPSG","9122"]],
    AUTHORITY["EPSG","4326"]]
FID Column = ogc_fid
Geometry Column = wkb_geometry
capital: String (1.0)
country: String (12.0)
name: String (25.0)
population: Integer (0.0)
```

Well, that worked—the GML file was successfully loaded into PostgreSQL. Using ogrinfo, we confirmed that it was loaded and had the proper coordinate system. If you look at the load command, you will notice we specified the coordinate system with the -a_srs switch using an EPSG code of 4326 (WGS 84, latitude/longitude). We did this because the GML file contains no projection information, even though the shapefile we created it from did. This allows our newly created PostGIS layer to play nicely with other WGS 84 layers in our database. Note that by specifying -a_srs, we aren't reprojecting or changing the data in any way. All we are doing is assigning the coordinate system to the PostGIS layer when it is created.

As you suspected, we can also unload data from PostGIS to a supported format. For example, we'll unload one of our layers to GML:

```
$ ogr2ogr -f GML volcanoes.gml "PG:dbname=gis_data host=madison" volcanoes
```

We can also specify a where clause in ogr2ogr to create a subset of our data. This works quite well for extracting features from PostGIS where the database might be quite large and we need only a small set of the data for our purpose. To extract a subset, use the -where switch and enclose the clause in double quotes:

```
$ ogr2ogr -f "ESRI Shapefile" strato_volcanoes.shp \
  "PG:dbname=gis_data host=madison" volcanoes -where "type = 'Stratovolcanoes'"
$ ogrinfo -so -al strato_volcanoes.shp
INFO: Open of `strato_volcanoes.shp'
      using driver `ESRI Shapefile' successful.

Layer name: strato_volcanoes
Geometry: Point
Feature Count: 7
```

We just created a shapefile containing only volcanoes of type Stratovolcanoes from our original layer in PostGIS. The PostGIS layer contains ninety-three volcanoes and our use of the where clause whittled that down to seven matches. If all you want to do is display a subset of a PostGIS layer, remember that QGIS supports subsets on the fly; otherwise, this is another useful technique for moving data around.

A further option when unloading or converting data is to specify a bounding rectangle using the -spat option. This will allow you to extract only those features within the rectangle, creating a spatial subset. You need to specify the bounds of the rectangle in the same coordinate system as the layer. To illustrate, let's extract a small subset of the cities layer we loaded into PostGIS and list the results using ogrinfo.

```
$ ogr2ogr -f "ESRI Shapefile" cities_subset.shp \
  -spat -152 58 -148 62 "PG:dbname=gis_data host=madison" cities
$ ogrinfo  -al cities_subset.shp
INFO: Open of `cities_subset.shp'
      using driver `ESRI Shapefile' successful.

Layer name: cities_subset
Geometry: Point
Feature Count: 2
Extent: (-149.449997, 60.119999) - (-149.172974, 61.188648)
Layer SRS WKT:
GEOGCS["GCS_WGS_1984",
    DATUM["WGS_1984",
        SPHEROID["WGS_1984",6378137,298.257223563]],
    PRIMEM["Greenwich",0],
    UNIT["Degree",0.017453292519943295]]
capital: String (1.0)
country: String (12.0)
name: String (25.0)
```

```
population: Real (11.0)
OGRFeature(cities_subset):0
  capital (String) = N
  country (String) = US
  name (String) = Seward
  population (Real) =        2699
  POINT (-149.449996948241989 60.119998931884801)

OGRFeature(cities_subset):1
  capital (String) = N
  country (String) = US
  name (String) = Anchorage
  population (Real) =      184300
  POINT (-149.172973632811988 61.188648223877003)
```

The cities layer in PostGIS has 606 features. You can see that our spatial subset has two features, both contained within the latitude/longitude rectangle we specified. Notice that the extent of the new layer is less than what we specified as the spatial boundaries. The bounding rectangle is specified as xmin, ymin to xmax, ymax—in this case -152, 58 to -148, 62. The extent of the new layer is smaller, because it represents the extent of the features *extracted*, not the search rectangle. You can probably think of ways in which creating subsets by attributes or spatial boundaries can come in handy.

A recent addition to the OGR utilities is support for KML. This allows you to export an OGR supported data source to KML for use in Google Earth.[4]

```
$ ogr2ogr -f KML volcanoes.kml "PG:dbname=gis_data host=madison" volcanoes
```

Depending on your version of GDAL/OGR, if you try to use ogrinfo on volcanoes.kml, you will get an error message. Preliminary support for reading KML files is not available in GDAL prior to version 1.5.

Coordinate System Conversion

We can also change the coordinate system of a layer using ogr2ogr. You can do this even if you don't want to change the format of the layer. For our cities.shp we created earlier, we could convert it from WGS 84 (latitude/longitude) to some other coordinate system, such as U.S. National Atlas Equal Area. To do this, we need to know either the projection parameters or the EPSG code. As we saw in Chapter 9, *Projections and Coordinate Systems*, on page 138, there are a number of ways

4. Be aware that Google Earth expects your KML to be in geographic coordinates.

to find this—additional ways include querying the spatial_ref_sys table in PostGIS and using the projection search feature in the QGIS projection dialog box. If you like, you can download all the EPSG codes in several database formats from OGP.[5] Another handy reference for coordinate systems is the Spatial Reference website, which provides an interactive web interface that allows you to find and display spatial reference information.[6]

Let's convert the cities shapefile to the U.S. National Atlas Equal Area projection (EPSG:2163) and check the result:

```
$ ogr2ogr -f "ESRI Shapefile" -t_srs EPSG:2163 cities_2163.shp cities.shp
$ ogrinfo -so -al cities_2163.shp
INFO: Open of `cities_2163.shp'
      using driver `ESRI Shapefile' successful.

Layer name: cities_2163
Geometry: Point
Feature Count: 606
Extent: (-11983157.793768, -9388276.306186) - (11909182.755140, 11453696.678979)
Layer SRS WKT:
PROJCS["US National Atlas Equal Area",
    GEOGCS["Unspecified datum based upon the Clarke 1866 Authalic Sphere",
        DATUM["Mean_Sea_Level",
            SPHEROID["Clarke_1866_Authalic_Sphere",6370997,0]],
        PRIMEM["Greenwich",0],
        UNIT["Degree",0.017453292519943295]],
    PROJECTION["Lambert_Azimuthal_Equal_Area"],
    PARAMETER["latitude_of_center",45],
    PARAMETER["longitude_of_center",-100],
    PARAMETER["false_easting",0],
    PARAMETER["false_northing",0],
    UNIT["Meter",1]]
NAME: String (40.0)
COUNTRY: String (12.0)
POPULATION: Real (11.0)
CAPITAL: String (1.0)
```

A couple of things to note about the coordinate conversion: First, we didn't do a format conversion—the output is a shapefile, just like the input. Second, we used -t_srs to transform the coordinates to the desired projection. When we look at the results using ogrinfo, we see that indeed the coordinate system was changed to U.S. National Atlas Equal Area.

When would coordinate conversion from the command line be useful? Apart from the reasons we've listed at the beginning of this section,

5. http://www.epsg.org
6. http://spatialreference.org

suppose you just received a CD containing 300 shapefiles that need to be transformed to a coordinate system compatible with your other data. Using the OGR utilities with a bit of simple shell, Ruby, or Python script would make this a simple and quick task. Loading each layer into a GUI and running a tool to transform the coordinates is fine for one layer, but not for 300.

Raster Conversion

The reasons for doing a raster conversion are pretty much the same as those for vector conversion, with the exception of loading into PostGIS. Let's add a couple more reasons to the list:

- You want to change the compression type of an image.
- You want to set a no-data value in the image to allow displaying certain areas as transparent.
- You want to rescale (reclass) the pixel values in an image.

In this section, we'll do a few raster conversions to illustrate some of the possibilities available with gdal_translate and gdalwarp.

Extracting Part of a Raster

Let's start by pulling out a piece of the world mosaic raster using a latitude/longitude rectangle. Since I happen to know the coordinates that cover Alaska, we'll use it in our example:

```
$ gdal_translate -a_ullr -180 72 -129 50 -projwin -180 72 -129 50 \
  world_mosaic.tif alaska_mosaic.tif
Input file size is 8192, 4096
Computed -srcwin 0 409 1161 501 from projected window.
0...10...20...30...40...50...60...70...80...90...100 - done.
```

Take a look at that command. The -projwin option specifies the area we want to clip out of the image using the coordinate system. Note we could use the -srcwin option to specify the clip area using pixel coordinates for the upper-left corner and a size in the x and y directions, also in pixels. We also used the -a_ullr option to force the output image to have the bounding coordinates we want; otherwise, it would be offset by a half-pixel in both the x and y directions (see Section C.2, *Using the Command Line*, on page 307 for more information on the cause of this offset).

To check to see whether this worked, we can open the new file in QGIS or one of the other applications we have discussed. Since we just want to see whether it worked, we could use any a graphic viewer on our system that supports TIFF. In Figure 11.5, on the next page, you can

Figure 11.5: Alaska derived from the world mosiac

see the result of our effort. You can see from the figure that indeed we cropped out Alaska from the world mosaic.

Let's run gdalinfo on the new file and examine a few of the details. We'll specify the no-metadata switch to cut down on the amount of output.

```
$ gdalinfo -nomd alaska_mosaic.tif
Driver: GTiff/GeoTIFF
Size is 1161, 501
Coordinate System is:
GEOGCS["WGS 84",
    DATUM["WGS_1984",
        SPHEROID["WGS 84",6378137,298.2572235629972,
            AUTHORITY["EPSG","7030"]],
        AUTHORITY["EPSG","6326"]],
    PRIMEM["Greenwich",0],
    UNIT["degree",0.0174532925199433],
    AUTHORITY["EPSG","4326"]]
Origin = (-180.000000000000000,72.026367187500000)
Pixel Size = (0.043945312500000,-0.043945312500000)
Corner Coordinates:
Upper Left  (-180.0000000,  72.0263672) (180d 0'0.00"W, 72d 1'34.92"N)
Lower Left  (-180.0000000,  50.0097656) (180d 0'0.00"W, 50d 0'35.16"N)
Upper Right (-128.9794922,  72.0263672) (128d58'46.17"W, 72d 1'34.92"N)
Lower Right (-128.9794922,  50.0097656) (128d58'46.17"W, 50d 0'35.16"N)
Center      (-154.4897461,  61.0180664) (154d29'23.09"W, 61d 1'5.04"N)
Band 1 Block=1161x7 Type=Byte, ColorInterp=Red
Band 2 Block=1161x7 Type=Byte, ColorInterp=Green
Band 3 Block=1161x7 Type=Byte, ColorInterp=Blue
Band 4 Block=1161x7 Type=Byte, ColorInterp=Alpha
```

The information for the raster confirms that it was assigned the same coordinate system as the source image since we didn't specify otherwise. You can also see the bounding coordinates of the image exactly match those we specified with -projwin. You might also notice we didn't specify an output format. This is because the default output format for gdal_translate is a GeoTIFF. If we wanted to convert the format at the same time, we would have specified it with the -of switch.

GDAL supports more than sixty raster formats, some of which are read-only. Those that are read-write we can be specified as an output format. Let's take a simple example and convert the alaska_mosaic.tif to a PNG:

```
$ gdal_translate -of PNG  -co "WORLDFILE=YES" alaska_mosaic.tif \
  alaska_mosaic.png
Input file size is 1161, 501
0...10...20...30...40...50...60...70...80...90...100 - done.
```

Notice we provided the -co option to create a world file in addition to the PNG. This allows us to use the PNG in a GIS application and have it display in the proper location. When using world files, you have to make sure they stay with the raster file. This is one of the advantages of a GeoTIFF, since it encodes the coordinate system information right in the raster.

Changing the Coordinate System

Let's do another example and change the coordinate system of alaska_mosaic.tif to Alaska Albers Equal Area, a projection commonly used for Alaska data. The EPSG code for the projection is 2964; however, it specifies the units as feet rather than meters. All my other Alaska data is in meters, so for this transformation, we'll go the hard way and specify the projection parameters in proj format:

```
$ gdal_translate -b 1 -b 2 -b 3 alaska_mosaic.tif alaska_mosaic_noalpha.tif
$ gdalwarp -t_srs '+proj=aea +lat_1=55 +lat_2=65 +lat_0=50 +lon_0=-154 +x_0=0 \
  +y_0=0 +ellps=clrk66 +datum=NAD27' alaska_mosaic_noalpha.tif \
  alaska_mosaic_albers.tif
Creating output file that is 1296P x 944L.
Processing input file alaska_mosaic_noalpha.tif.
:0...10...20...30...40...50...60...70...80...90...100 - done.
```

In Figure 11.6, on the following page, you can see the result of the warping process. Notice that we translated the image before the warp using gdal_translate. This was to remove the alpha channel from the original image. The alpha channel is used to determine transparency for each pixel in the image. In the case of our mosaic, it was found by

Figure 11.6: Alaska mosaic warped to Alaska Albers projection

trial and error that the oceans were transparent. When warped, they turned black. To get around this problem, we used gdal_translate with a series of -b switches to specify which bands in the image should be used in the output image. This effectively strips the alpha band. If you look back at the gdalinfo output for alaska_mosaic.tif, you'll see that it reported Bands 1 through 3 as red, green, and blue, respectively, as well as Band 4 as alpha. Once we removed the alpha band, the warp gives us the expected result.

Warping an image with gdalwarp is a quick and efficient way to change the coordinate system as opposed to other methods one might use. As you can see, you have to know a little bit about the data you are working with in order to be successful in getting the results you want. If you have more than one image to process and want to combine them, you can use wildcards with gdalwarp to mosaick them on the fly, creating a single image in the process.

Raster Transparency

The last raster example we'll look at is setting a transparency value. This is useful when you want the layers underneath the raster to be visible. To set the transparency, we'll use the GDAL Virtual Format (VRT). A VRT file is a description of the raster, stored in XML, that can be modified using a text editor. The first step is to use gdal_translate to create the VRT file. In this case, we'll use the survey plat that we digitized from Section 8.1, *Digitizing*, on page 120. Running the gdal_translate command on the plat (l1-1.tif) creates the VRT for us:

```
$ gdal_translate -of VRT l1-1.tif l1-1.vrt
Input file size is 9568, 12754
```

Before we can set the transparency, we have to determine the index of the white color values in the raster. For this, we use gdalinfo with the -nomd option since we don't need the metadata, just the color table information:

```
$ gdalinfo -nomd l1-1.tif
Driver: GTiff/GeoTIFF
Size is 9568, 12754
Coordinate System is `'
Corner Coordinates:
Upper Left  (    0.0,    0.0)
Lower Left  (    0.0,12754.0)
Upper Right ( 9568.0,    0.0)
Lower Right ( 9568.0,12754.0)
Center      ( 4784.0, 6377.0)
Band 1 Block=9568x12754 Type=Byte, ColorInterp=Palette
  Color Table (RGB with 2 entries)
    0: 255,255,255,255
    1: 0,0,0,255
```

From the output, we can see that RGB 255,255,255 (white) is at index 0 in the raster. We now have everything we need to set the transparency using the VRT file. To set the transparency, we add a <*NoDataValue*> tag inside the <*VRTRasterBand*> tag, using 0 as the index value. Our modified VRT file contains the following:

Download l1-1.vrt

```
Line 1  <VRTDataset rasterXSize="9568" rasterYSize="12754">
-         <Metadata>
-           <MDI key="TIFFTAG_SOFTWARE">AccXES Version 4.2        Build 106</MDI>
-           <MDI key="TIFFTAG_DATETIME">2004:02:26 16:11:38</MDI>
5           <MDI key="TIFFTAG_XRESOLUTION">400</MDI>
-           <MDI key="TIFFTAG_YRESOLUTION">400</MDI>
-           <MDI key="TIFFTAG_RESOLUTIONUNIT">2 (pixels/inch)</MDI>
-         </Metadata>
```

```
-     <VRTRasterBand dataType="Byte" band="1">
10      <Metadata>
-         <MDI key="NBITS">1</MDI>
-       </Metadata>
-       <ColorInterp>Palette</ColorInterp>
-       <NoDataValue>0</NoDataValue>
15      <ColorTable>
-         <Entry c1="255" c2="255" c3="255" c4="255"/>
-         <Entry c1="0" c2="0" c3="0" c4="255"/>
-       </ColorTable>
-       <SimpleSource>
20        <SourceFilename relativeToVRT="1">11-1.tif</SourceFilename>
-         <SourceBand>1</SourceBand>
-         <SrcRect xOff="0" yOff="0" xSize="9568" ySize="12754"/>
-         <DstRect xOff="0" yOff="0" xSize="9568" ySize="12754"/>
-       </SimpleSource>
25    </VRTRasterBand>
-   </VRTDataset>
```

The tag we added is found in line 14 of the VRT file. Notice the file also contains a reference to the raster in the <*SourceFilename*>. This is important—you still need the original raster in order for the VRT to work properly.

When the VRT file is displayed by software that uses GDAL for raster access, white will be transparent, allowing the underlying layers to show through. In Figure 11.7, on the following page, we have loaded the parcel shapefile we digitized in Section 8.1, *Digitizing*, on page 120, into QGIS, overlaid by our VRT file. You'll notice that the parcels (green polygons) are visible, proving that the white of our original raster is now transparent. You can use VRT files in a number of ways, including setting transparent values for adjacent images so they can be displayed together without blotting each out. The key is to use gdalinfo to get the color index number and then create and edit the VRT file(s).

For more information on the VRT format, see the GDAL Virtual Format Tutorial.[7]

11.3 Creating a Spatial Index for Shapefiles

A spatial index can improve the performance of your mapping application, whether it be on the desktop or the server.[8] The index speeds up drawing, selecting, and identifying features by allowing the software

7. http://www.gdal.org/gdal_vrttut.html
8. This assumes that your software supports the use of the spatial index in .qix format.

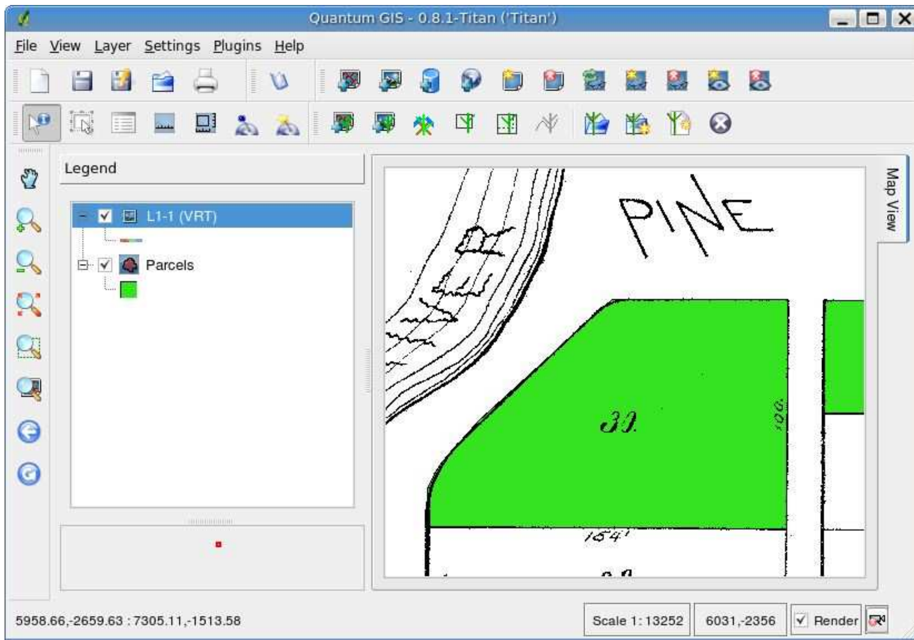


Figure 11.7: VRT raster over the parcel shapefile

to quickly locate the features of interest. There is more than one way to create a spatial index. In Section D.1, *Spatial Indexes*, on page 333, you will see how to create one using QGIS. Of course, there is a way to do this from the command line as well. You'll find this useful if you have a whole batch of shapefiles you want to index. Rather than loading each one into QGIS, opening the properties dialog box, and clicking the button to build the index, you can just write a simple script to do the job. To create a spatial index, use the shptree application. You can get shptree in a number of ways. It's included in the FWTools[9] distribution as well as in MapServer.[10] You're probably going to want FWTools anyway, because it contains all the OGR and GDAL utilities we've been using and a lot more goodies. Let's look at the usage:

```
$ shptree
Syntax:
    shptree <shpfile> [<depth>] [<index_format>]
```

9. http://fwtools.maptools.org
10. http://mapserver.gis.umn.edu

```
Where:
 <shpfile> is the name of the .shp file to index.
 <depth>   (optional) is the maximum depth of the index
           to create, default is 0 meaning that shptree
           will calculate a reasonable default depth.
 <index_format> (optional) is one of:
           NL: LSB byte order, using new index format
           NM: MSB byte order, using new index format
       The following old format options are deprecated:
           N:  Native byte order
           L:  LSB (intel) byte order
           M:  MSB byte order
       The default index_format on this system is: NL
```

Creating an index is easy, despite all the somewhat confusing options for shptree. In fact, the defaults are usually fine, and you can just use the following:

```
$ shptree earthquakes.shp
creating index of new  LSB format
```

As you can see, there isn't much in the way of feedback. When the command is complete, you'll find a file with a qix extension:

```
$ ls -l *.qix
-rw-r--r-- 1 gsherman gsherman 72 Mar 21 01:22 earthquakes.qix
```

That's all there is to it. Make sure the spatial index stays with the rest of the shapefile when you copy or move it somewhere else. The spatial index will work with QGIS and MapServer and probably any spatial application that used OGR for reading shapefiles.

11.4 PostGIS

PostGIS comes with a couple of utilities for moving data in and out of a PostgreSQL database. Although you can accomplish the same results with ogr2ogr, the utilities supplied with PostGIS have some additional options that you will find useful. The limitation, of course, is that only shapefiles are supported. Given the flexibility and capability of the OGR utilities, this isn't a problem. We can still get the data from here to there safely and in the form we need.

Importing Shapefiles

To import a shapefile into PostGIS, use the shp2pgsql command. Let's look at the options and syntax.

```
$ shp2pgsql
RCSID: $Id: shp2pgsql.c 2513 2006-10-14 14:22:10Z mschaber $ RELEASE: 1.2.1
USAGE: ./shp2pgsql [<options>] <shapefile> [<schema>.]<table>
OPTIONS:
  -s <srid>  Set the SRID field. If not specified it defaults to -1.
  (-d|a|c|p) These are mutually exclusive options:
      -d  Drops the table, then recreates it and populates
          it with current shape file data.
      -a  Appends shape file into current table, must be
          exactly the same table schema.
      -c  Creates a new table and populates it, this is the
          default if you do not specify any options.
      -p  Prepare mode, only creates the table.
  -g <geometry_column> Specify the name of the geometry column
     (mostly useful in append mode).
  -D  Use postgresql dump format (defaults to sql insert statements.
  -k  Keep postgresql identifiers case.
  -i  Use int4 type for all integer dbf fields.
  -I  Create a GiST index on the geometry column.
  -S  Generate simple geometries instead of MULTI geometries.
  -w  Use wkt format (for postgis-0.x support - drops M - drifts coordinates).
  -W <encoding> Specify the character encoding of Shape's
     attribute column. (default : "ASCII")
  -N <policy> Specify NULL geometries handling policy (insert,skip,abort)
    -? Display this help screen
```

As you can see, shp2pgsql has quite a few options. Let's examine the major ones we need to know in order to get a shapefile into the database. PostGIS uses the concept of a spatial reference ID (SRID), and this option is specified using -s. For the most part, this is equivalent to an EPSG code; however, you can define your own SRIDs in the database. The spatial_ref_sys table contains the spatial reference systems (think projections) that PostGIS knows about as well as the SRID for each. If we look at the record for SRID 4326 in the spatial_ref_sys table, we find that it's the same as EPSG 4326 that we used when doing vector data conversions with OGR. In fact, the authority for that SRID is EPSG.

The -d, -a, -c, and -p switches provide some management options for creating the layer. As you can see, -c is the default option, and you don't need to specify it. The -g switch allows us to specify a name for the column in the table that will hold the feature geometry information. You don't need to specify this unless you don't like the default name or have existing data or standards.

A layer in PostGIS without a spatial index isn't a happy layer. Or at least we won't be happy using it because performance will suffer. You might be thinking we should always specify the -I switch. However, building a

spatial index can be a time-consuming affair for large tables. You might want to defer building the index, especially if you are loading a lot of data with a script.

So, what do we really need to specify? Not much when it comes down to it, but here's one word of caution: before you start going wild loading data, think about the SRID you should use and then specify it for every layer.[11] If you don't do it up front, you'll regret it later. Let's load a shapefile as an example:

```
$ shp2pgsql -s 4326 -I cities.shp cities_pg | psql -d gis_data
Shapefile type: Point
Postgis type: POINT[2]
BEGIN
NOTICE:  CREATE TABLE will create implicit sequence "cities_pg_gid_seq"
  for serial column "cities_pg.gid"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index
    "cities_pg_pkey" for table "cities_pg"
CREATE TABLE
                   addgeometrycolumn
-------------------------------------------------------
 public.cities_pg.the_geom SRID:4326 TYPE:POINT DIMS:2

(1 row)

CREATE INDEX
INSERT 0 1
INSERT 0 1
INSERT 0 1
...
COMMIT
```

OK, let's analyze what we did here. We knew beforehand that the EPSG code for cities.shp was 4326, so we specified that as the SRID. The only other option we used was -I to build the spatial index. Other than that, we give the shapefile name and the name of the table we want to create, in this case cities_pg. If we had ended the command there and hit Enter, we would have seen a slew of SQL statements scroll by. That's because shp2pgsql sends its output to stdout (in other words, your terminal or command window). We could pipe that to a file using > and then use the file in an application that was capable of running SQL commands from a file to send it to the appropriate database. We can take a shortcut, though, and just pipe the output from shp2pgsql directly to psql, the PostgreSQL interactive terminal. Passing the name of the database to

11. The shapefiles have to be in the same projection as the SRID you plan to use—loading them into PostGIS won't transform them.

psql using the -d switch is all we need to send the SQL to the database and create the table.

As the import proceeds, the results are printed to the screen. I truncated the INSERT statements in the example, since there are 606 of them—one for each city. Notice that before the import began we get some feedback about what's going on, including the creation of the geometry column.

If we use psql to examine the table after it's loaded, we find that our spatial index was created as part of the import. Partial output from the \d command in psql shows the GIST index cities_pg_the_geom_gist was created on the geometry column.

```
Indexes:
  "cities_pg_pkey" PRIMARY KEY, btree (gid)
  "cities_pg_the_geom_gist" gist (the_geom)
```

Our table is ready to use in our GIS applications; however, you should run the VACUUM ANALYZE command to have PostgreSQL collect statistics to improve performance.

Exporting to a Shapefile

To export data from your PostGIS database to a shapefile, use the pgsql2shp command. This command has fewer options than its counterpart:

```
$ pgsql2shp
RCSID: $Id: pgsql2shp.c 2513 2006-10-14 14:22:10Z mschaber $ RELEASE: 1.2.1
USAGE: ./pgsql2shp [<options>] <database> [<schema>.]<table>
       ./pgsql2shp [<options>] <database> <query>

OPTIONS:
  -f <filename>  Use this option to specify the name of the file
     to create.
  -h <host>  Allows you to specify connection to a database on a
     machine other than the default.
  -p <port>  Allows you to specify a database port other than the default.
  -P <password>  Connect to the database with the specified password.
  -u <user>  Connect to the database as the specified user.
  -g <geometry_column> Specify the geometry column to be exported.
  -b Use a binary cursor.
  -r Raw mode. Do not assume table has been created by
     the loader. This would not unescape attribute names
     and will not skip the 'gid' attribute.
  -k Keep postgresql identifiers case.
  -? Display this help screen.
```

Basically, we just need to provide the name for the shapefile we want to create and the connection information for the database. You probably realize this means you can use this command from a remote machine. Let's be wishy-washy and export our layer back out of the database:

```
$ pgsql2shp  -f cities_out.shp gis_data cities_pg
Initializing... Done (postgis major version: 1).
Output shape: Point
Dumping: XXXXXXXX [606 rows].
```

Things worked as expected—we got all 606 cities out of the database and into a new shapefile. We could use ogrinfo to check it out, but trust me, it's the same as what went into the database. Notice that we didn't specify anything for the database connection. That's because we ran pgsql2shp on the same host as the database server. If we were doing an export from a remote server, we would have to specify host, user, and password. If your database server runs on a nonstandard port, you will have to specify it as well.

If your table has more than one geometry column, you can specify which to use for exporting the features with the -g switch. You might be wondering how you get a table with more than one geometry column. All I'll tell you is it didn't happen with shp2pgsql. Seriously, though, creating a table with more than one geometry column is done programmatically through SQL or custom applications using the programmer's API for PostgreSQL.

That's pretty much the quick tour for getting data out of your PostgreSQL database. There are other ways too, including writing custom scripts using APIs that know how to talk to the database. Although pgsql2shp works well for exporting to shapefiles, you might find that ogr2ogr provides a better solution for exporting to other formats.

Chapter 12

Getting the Most Out of QGIS and GRASS Integration

We have mentioned the QGIS-GRASS integration in a number of places so far. In this chapter, you will see how QGIS can serve as a front end for viewing and editing GRASS data, as well as for performing analysis and data conversion. We're venturing into some powerful territory here, allowing you to harness the geoprocessing power of GRASS.

QGIS supports GRASS through the use of a plugin. The plugin provides access to GRASS data and functions and is distributed with all official QGIS packages. Actually, it consists of a data provider to bridge between a GRASS map layer and the QGIS map canvas and a plugin to provide the user interface. Well, enough of the boring details; let's get started by loading up the GRASS plugin and seeing how it all works.

To review, when you initially start QGIS, there are no plugins loaded. To load a plugin, you use the Plugin Manager, found in the Plugins menu. The Plugin Manager provides a list of all the available plugins and whether they are currently loaded (indicated by a checkbox next to their names). To load the GRASS plugin, click the checkbox next to its name, and click the OK button. This loads the GRASS plugin, adds a GRASS menu to the Plugins menu, and adds a new toolbar to the GUI. For a review of plugins in QGIS, see Section D.4, *Plugins*, on page 339.

If you have followed along with some of the previous GRASS examples, you probably recall how to load the GRASS plugin in QGIS and create a location. If not, refer to Section C.1, *Location, Location, Location*, on page 296 for a refresher. You might also want to take a look at Section C.2, *Importing with QGIS*, on page 313 for information on how to

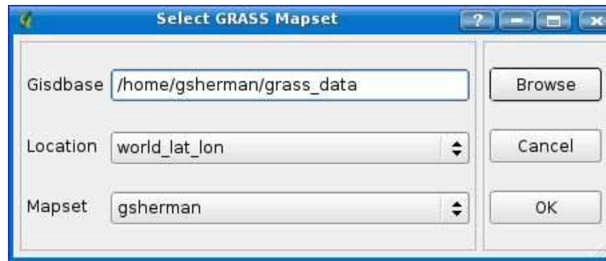


Figure 12.1: Selecting a GRASS mapset in QGIS

import vector layers using the GRASS toolbox in QGIS. We will now expand on those concepts by exploring the toolbox in depth, using it first to load some more data and then to do a bit of data conversion and analysis. From this point on, we assume a working GRASS install and a ready-to-use mapset.

12.1 Loading and Viewing Data

If you work through the GRASS basics in Appendix C, on page 296, you end up with two vector layers in your mapset: cities and world_borders. Our goal now is to add the world mosaic to GRASS using the toolbox. To do this, we first add world_mosaic.tif to QGIS as a regular raster layer. Once it's loaded up, we need to open our mapset using the Open mapset menu item in the GRASS menu (remember the GRASS menu is located under the Plugins main menu). The mapset we want is in our world_lat_lon location. In Figure 12.1, you can see the dialog box used to open a mapset in QGIS. Notice that you can open any mapset in any location in any GRASS database using this dialog box. As you change the Gisdbase location, either by typing a path or by browsing to it, the Location and Mapset drop-downs change to reflect what's available.

Once we open the mapset, the Open GRASS tools tool becomes active on the GRASS toolbar, as do the region and vector edit tools. Now that the mapset is open, we can open the GRASS toolbox. Click the Open GRASS Tools tool and wait as the toolbox initializes. The tools in the toolbox are added dynamically. In fact, you can customize the tools and add more GRASS functionality. The downside (well sort of) is that the more tools in the box, the longer it takes to open. Once it's up and initialized,

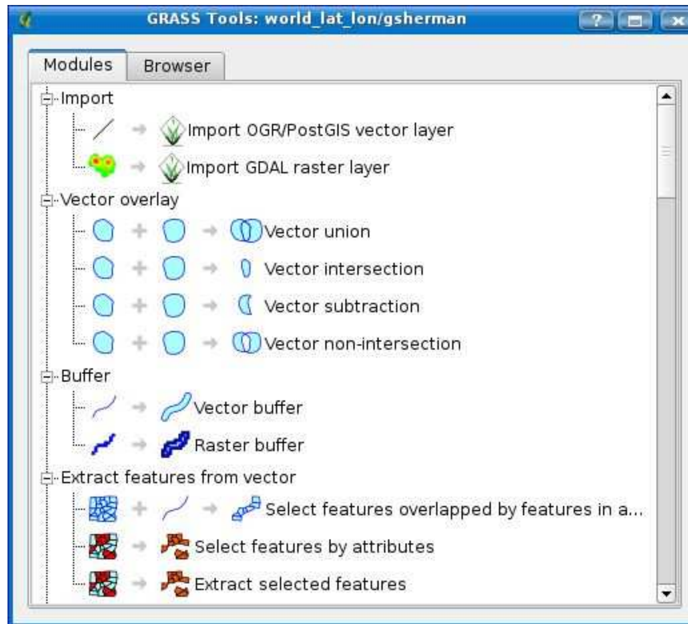


Figure 12.2: The GRASS tools in QGIS

you are presented with a collection of tools, as shown in Figure 12.2, on the following page.

Before we import our raster, let's look at the toolbox for a minute. The tools are arranged in a tree structure by category. In Figure 12.2, you can see the following categories: Import, Vector Overlay, Buffer, and Extract Features from Vector. But that's not all. Notice the scrollbar in the toolbox—it has a ways to go to get to the bottom. That's where the rest of the GRASS tools are hiding for the moment. We'll look at some of them a bit later.

To import the raster, we click the Import GDAL raster layer tool. You must have a raster loaded in QGIS before using this tool, which we do so we are in good shape.[1] Each tool in the GRASS toolbox has its own page for accepting input and providing feedback as it runs. Typically the tool page will contain Options, Output, and Manual tabs. You enter the

1. The same is true for vectors. To import either type using the toolbox, you must load the GDAL/OGR-supported layer in QGIS first. To import a vector, use the Import OGR/PostGIS vector layer tool.

required parameters for the tool on the Options tab; then when the tool is run, the output shows up on the Output tab. Clicking the Manual tab displays the manual page for the GRASS tool you are working with.

The Options tab contains a drop-down box populated with the layers eligible for conversion. In this case, since we have only world_mosaic.tif loaded, it's the only thing in the list. To convert it, we just need to supply a name for the output map. Rather than be original, we'll use world_mosaic for the output name. Now all that's left to do is click the Run button and watch the output fly by. Once the raster is imported, we can review the contents of the Output tab to look for any problems or see the results and details of the conversion. Not only that, but it also provides a good way to learn about what's going on inside GRASS. Assuming all went well, we now have the world_mosaic loaded in to GRASS. Load it up in QGIS using the Add GRASS raster layer tool.

Those are the basics of getting data into GRASS using the import capabilities of the toolbox. Notice that not only can we load an OGR vector layer, but we can load PostGIS as well. Sometimes it makes sense to convert your data, and other times you can just use them in their native form in QGIS with your GRASS data. If you need to manipulate the data, it's best to bring it into GRASS.

Once you've added a GRASS map to the map canvas, it works pretty much like any other layer in QGIS. You can identify features, make selections, view the attribute table, and label features. The one thing you can't do is edit the data—at least not through the editing tools we've seen thus far in QGIS.

12.2 Editing GRASS Data with QGIS

Editing vector GRASS maps in QGIS is also done via the GRASS plug-in. The editing tools are designed to work with the underlying GRASS vector model, which is different from a shapefile or PostGIS data store. GRASS is topological, meaning it understands the relationship between adjacent features and stores common boundaries only once. Shapefiles and PostGIS data, on the other hand, are nontopological—each feature is stored in its entirety, with no regard for adjacent features.

To be able to edit a GRASS map, you obviously need to have the mapset open and the map loaded into QGIS. You may think I'm stating the obvious here, but in fact you can load a GRASS map *without* opening the mapset. If you do that, the toolbox, region tools, and editing tool

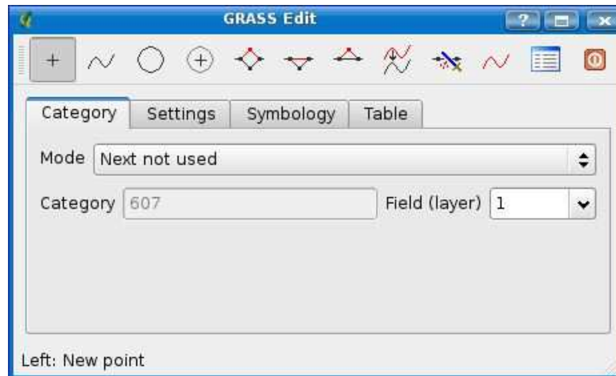


Figure 12.3: GRASS edit tools in QGIS

will be disabled. Now that everything is in order, we can click the Edit GRASS Vector layer tool, opening the edit tools shown in Figure 12.3, on the following page.

Which layer are we editing? The answer is the one highlighted in the legend when you bring up the edit tools. Fortunately, if the highlighted layer isn't a GRASS vector map, the Edit GRASS Vector layer tool will be disabled. So, the trick is to make sure the layer you want to edit is highlighted before opening the edit tools.

The edit tools closely mirror those used when digitizing in GRASS, a process you can learn about in Section C.5, *Digitizing and Editing in GRASS*, on page 323.

First we will look at the basics of the editing tools and then get into the specifics. The task we'll use to illustrate simple editing is adding a new city to the world. Since our mapset is already open, we can load the cities layer and use it for our editing task. Each feature in GRASS has a category field named cat that serves as the identifier. When we add a new point (city), we have a choice of how that category is created, as shown in the Mode drop-down box. The choices are as follows:

- *Next Not Used*: The next unused category will be assigned automatically.
- *Manual Entry*: You will enter the category manually when you create the feature.
- *No Category*: No category will be assigned to the new feature.

Figure 12.4: Adding attributes to a GRASS feature

We'll explore some of these options later, but for now, automatically assigning the next available category seems like a good way to go. The first tool on the toolbar is New point. Let's use it to create a new city in Alaska named, for lack of a better term, Quantum GIS City. First we zoom in to where we want to create the city, choose the New point tool by clicking it, and then click the map to place the feature. When we click, the city is created on the map, and the GRASS Attributes dialog box opens to allow us to enter the information for the new city. In Figure 12.4, you can see the dialog box with the information for our new city (OK, I know that Alaska isn't a country, although some Alaskans wish it were). The category (607) was automatically assigned for us, and we entered the name, country, population, and capital. When we click Update, the information is updated in the GRASS database. We could choose to create another new record or delete this one entirely, in which case our feature would have no attributes.

We added the city, but nothing is saved until we close the editing toolbox. Although I said when you click Update, the attributes are saved—actually they are queued up and ready to be saved. Once we close the editing tools, our changes are saved, and the new city is rendered on the map using the same symbology as the rest of the cities.

Creating Multiple Layers in a Map

One of the features of GRASS is the ability to create more than one layer per map. This means you can have point, line, and polygon layers within a single map. Let's look at a practical example of how this might be useful.

We created Quantum GIS City, and it had a population of one. Now after a major infusion of capital because of it's cottage GIS industry, the population has boomed, and the city has grown by orders of magnitude. But in our GRASS layer, it's just a point. This is where we can take advantage of multiple layers in a map. We can digitize the boundaries of the city, creating either a polygon or line layer. We end up with both the point location and the boundaries and can use one or both at the same time in QGIS, depending on what information we are trying to convey.

To create a polygon for Quantum GIS City, start editing the cities GRASS map. This time instead of adding a point, we choose the New boundary tool on the toolbar and digitize the boundary of our burgeoning city. Once we close the boundary, the attribute dialog box pops up, allowing us to fill in the attributes. Then we use the New centroid tool to add a centroid to the boundary, creating the polygon. Again the attributes dialog box pops up, and we can enter the information again. Now we have three features for our city and attributes for each. Now you are asking yourself, why did we have to enter the attributes each time? The short answer is you don't. Since each of the feature types is for the same city, we should have created the point first, filled in the attributes, and then used its cat for the other features. You can do this by changing the mode in the edit toolbox to Manual and entering the category value of the point in the Category field. When we digitize the boundary or add a centroid, it will automatically populate the attribute fields from those associated with the point. This saves time and ensures that the attributes for each feature type are consistent.

Now we have more versatility in our cities map, in that we can display points when looking at the worldwide or country view and can display lines or polygons when zoomed in for more detail. By now you should be thinking of all kinds of possibilities this provides, since many "features" may have a point or polygon representation in real life. With shapefiles we would have to create a separate file for each feature type. PostGIS supports geometry collections, but QGIS doesn't know how to visualize them since it makes the assumption of one feature type per layer.

Creating a New Map

What if we want to create a new GRASS vector map and add data to it? We could pop out to the GRASS command line or GUI and do it, or we can use the Create a new GRASS vector item from the GRASS plugin menu in QGIS. Before you can do this, though, you have to open the mapset where you want the new vector map to live.

The steps involved in creating a new GRASS map are as follows:

1. Open the mapset where the map will reside.
2. Click the Create a new GRASS Vector menu item.
3. In the New Vector Name dialog box, enter the name for the new map.
4. Click OK.
5. Add features to the map using the edit tools.

When you create a new layer, the edit tools automatically pop up so you can add features. QGIS doesn't add your new map to the canvas yet, so it may be a bit confusing as to what you are doing. Don't worry—the layer will show up once we close the edit tool dialog box.

When first created, the layer has only one attribute: cat. Let's say we are creating a new map named water_wells. This map will contain the point locations of water wells, and we will digitize them off a raster map we have showing the locations. Having only the cat attribute by itself isn't very useful when it comes to storing information about the wells. We need some additional information, such as owner name, depth, and whether it's an active well. To do this, we use the Table tab in the edit dialog box.

In Figure 12.5, on the next page, you can see the table I constructed for the water wells map. To build the table, you just use the Add Column button to add a new column and then give it a name and set the type and length as appropriate. You can see in the figure I added the owner, depth, and active fields. For the active field, I chose to use a single character that will contain a "Y" or "N" to represent yes or no. Once everything is set up, clicking the Create/Alter Table button saves the changes. We can now digitize the wells and add the attributes we need. We'll start by adding a new well within the Quantum GIS City polygon and assigning the owner as the city, depth of 220, and a Y in the active column to indicate it is in use. From there, you can go forth in like fashion and digitize all the water wells.

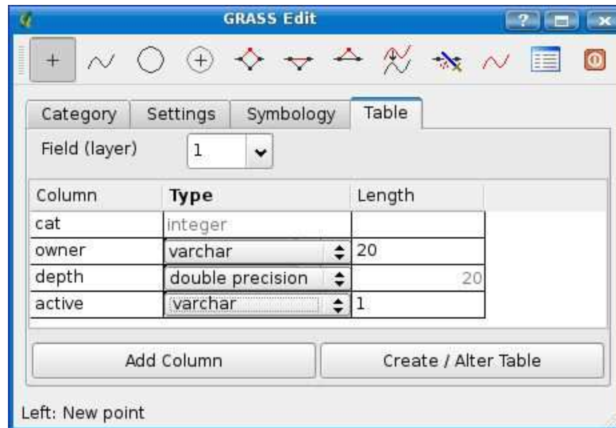


Figure 12.5: Adding columns to the new GRASS map table

We quickly realize in using our shiny new water_wells map that it is lacking something. We should have had a name field to store the name of the well. For residential wells, this probably isn't important, but it's likely that the managers of Quantum GIS City will want names on their wells to more easily manage them. No problem—we can easily add the new column and fix any existing wells so they have a name.

To add a name column, begin editing the water_wells map, and click the Table tab. We now see the table with its current columns and their types. To add a new column, just click the Add Column button, and fill in the details.

What about the well(s) we already created? The GRASS edit tools allow you to edit the attributes for any feature. To edit the attributes for an existing feature, click the Edit attributes tool. This brings up the same dialog box you see when entering the attributes for a newly created feature. Now there is a blank name field. We just fill it in with the name of the well, as shown in Figure 12.6, on the following page, and click the Update button. You'll have to repeat this for all the existing wells. Of course, this illustrates the point that you should think about your requirements before creating the data. It will save you time and energy, especially if you realize it too far into your project.

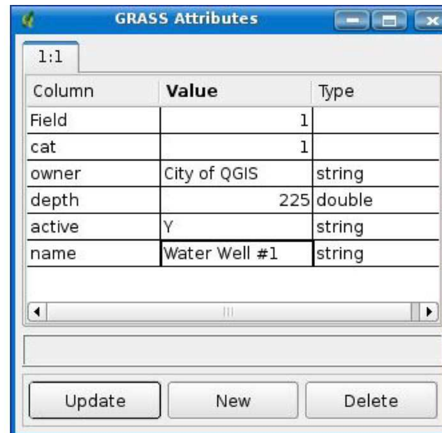


Figure 12.6: Editing the attributes of an existing feature

The Rest of the Story

That pretty much covers the basics of editing GRASS maps with QGIS. Before we move on, let's look at some settings you can customize while editing.

The first is the Settings tab. Currently there is only one setting—the snap tolerance. This controls how close you have to be to a feature for the current tool to snap to it. It's set in screen pixels. You can customize it for your use if you find the default value of 10 is too large (or small). Having too large a snap tolerance may result in selecting (and possibly deleting/moving) the wrong feature.

You can also customize the size and colors used for displaying items by clicking the Symbology tab. This allows you to customize the line width used when digitizing, as well as the marker size used for points and centroids. Below that is a list of the colors that can be customized for things like background, highlight, point, line, boundary, centroid, and so forth. It's good to have options, but in most cases I find the default colors to be fine for digitizing.

To complete our editing saga, the results of our efforts are shown in Figure 12.7, on the next page. Each of the GRASS layers (feature types) for the cities map is shown in the legend. I've renamed them so you can tell which layer type they represent. We also see the water_wells map, containing two wells. You'll notice that the wells are rendered by their status—active or inactive. We can see from the map that the Quantum GIS City fathers have decommissioned old Municipal Well #1.

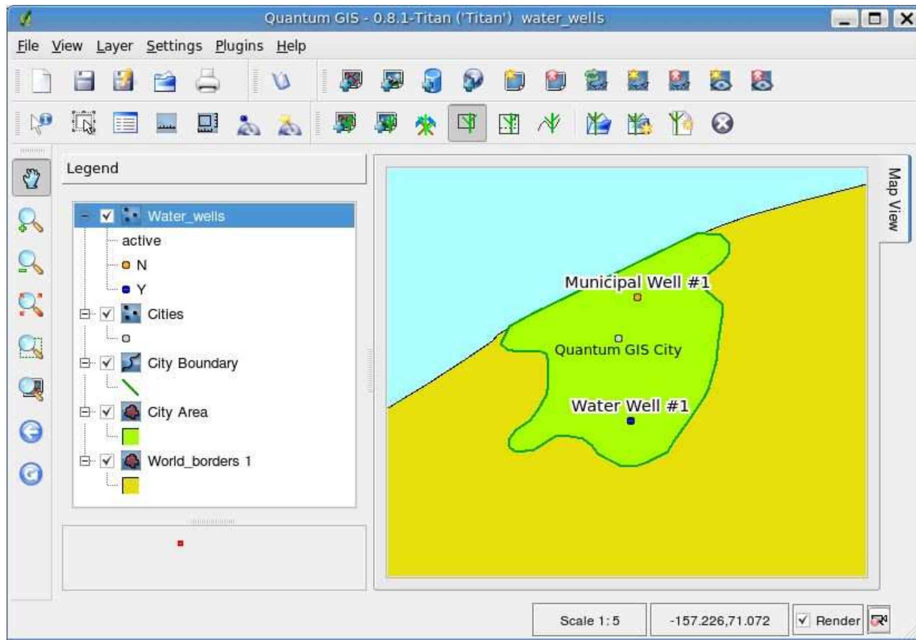


Figure 12.7: Completed city map with water wells

Which method of GRASS editing should you use: QGIS or GRASS gis.m? It depends mainly on preference, how comfortable you are with the tools, and whether you can perform all the tasks you need to from the chosen interface. In the next section, we will take a look at the analysis and conversion tools available in the GRASS toolbox in QGIS. This may help you decide whether you can do most of your GRASS work in QGIS or whether gis.m is the way to go.

12.3 Using Analysis and Conversion Tools

Creating and drawing data in GIS is just part of the picture, unless of course all you want is a pretty map. A big part of GIS and what makes it a powerful tool is the ability to do analysis of spatial relationships. In this section, we plan to take a look at some of the tools available in the QGIS GRASS toolbox that allow you to do both conversion and analysis. But before we go there, let's complete our look at the GRASS Tools dialog box.

We got our first look at the toolbox back in Figure 12.2, on page 210.

We used a couple of the tools to convert some vector and raster data into GRASS but didn't really look at the toolbox that closely. If you refer to the figure, you'll see at start-up there are two tabs: Modules and Browser. The Modules tab contains all the GRASS tools you can run from the toolbox. These tools are added at runtime from a configuration file, and there is actually a way to customize and add to the tools that are presented in the toolbox, assuming you have attained the appropriate level of GRASS mastery. The tools are categorized by function (also customizable) in the module list.

We will use some of the tools in the module list shortly. The Browser tab contains the GRASS browser and allows you to view all the maps in your current mapset, as well as manage them. Before we dive into the modules, let's learn a bit about the browser.

Using the Browser

To activate the browser, simply click the Browser tab. All the mapsets in the current location are displayed on the left in a tree structure. Typically you will see the PERMANENT mapset, along with the one or more user mapsets. Remember, the PERMANENT mapset contains read-only maps that are shared among users and are generally base layers everybody needs.

If a mapset contains maps, you will be able to expand the tree. The maps are further categorized into raster, region, and vector nodes. Expanding one of the nodes will show you a list of all the maps associated with it. If you click a map, the pane on the right displays information about the map. In Figure 12.8, on the next page, you can see the browser with the Cities map selected. Notice all the good information displayed in the pane on the right? We can get a good overview of the map, including the number of each feature type (in this case 607 points, 0 lines, 1 boundary, 1 centroid, 1 area, and 1 island). The original layer as imported from the shapefile had 606 points. The other features were added when we digitized the city limits of Quantum GIS City and its point location. We can also see the extents of map—in this case it takes up most of the world.

As we work with GRASS maps, a history is recorded. You can see in our browser example the command used to create the Cities map from the original shapefile. This is displayed in the right pane, just below the extent information. Also note on the left under the Cities node, there are three layers, one for each feature type. These layers are prefixed

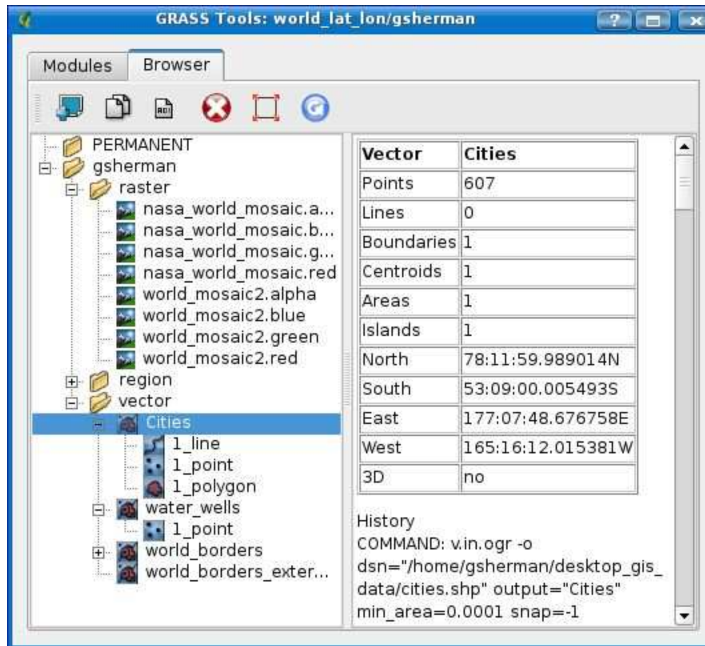


Figure 12.8: GRASS browser in QGIS

with a number, followed by the feature type. Using the browser gives you a quick overview of your maps and layers, as well as some detailed information about the number of features and the history of the map.

Let's take a look at the toolbar in the browser because it has a number of useful functions for working with and managing our maps.

The first tool is the Add selected map to canvas tool. Its purpose seems pretty obvious—you select a map and click the tool, and it gets added to the map canvas in QGIS. What gets added depends on what you have selected in the list of maps and layers. If you have a map selected, for example, Cities, and click Add selected map to canvas, all the layers under Cities will be added. This means the point, line, and polygon layers get added as separate layers in QGIS, and we will have three entries in the legend. If you had selected just the polygon layer of the Cities map, only it would be added. This feature gives you a quick way to add every layer for a map or to be more discriminating and add only the particular layer you need.

The next tool is Copy selected map, which allows you to copy the currently selected map into the current mapset. There are a number of reasons why you might want to copy a map. A typical case is when you are about to do some fancy (read: dangerous) conversion or edit—you might want to make a copy in case things go bad. When you click Copy selected map, you are prompted for a name for the new map. Give it a name, and click OK to create the new map. Note that in order for this to work, you have to highlight a map in the browser list—the copy won't work on a layer of a map.

The Rename selected map tool allows you to rename a map. This can come in handy when doing a risky edit operation. Before you begin, make a copy of your map. If things go badly during the edit operation, you can just delete the original map that is now fouled up and rename the copy to the original, so you can try again. Don't forget to make another clean copy at this point. When naming or renaming a map, you're not allowed to enter spaces since they are not valid in a GRASS map name—don't think that your spacebar is broken.

The red button with the big X in the middle is the Delete selected map tool. Use this one with caution, because once it's gone, it's gone. Fortunately, you have to confirm the delete operation, giving you a chance to change your mind. When you delete a map, it is removed from the list of maps in the browser. Interestingly enough, if it happens to be on the QGIS map canvas, it isn't removed, and you can still identify features and view the attribute table. If you try to edit it, the operation will fail since the underlying data structures have been removed.

There are two other buttons on the toolbar. The first allows you to set the GRASS region to the currently selected map. This information is saved, and the next time you run GRASS, the region will be restored. Unlike a lot of file system browsers, the GRASS browser doesn't continuously poll or receive notification when the contents of your mapset has changed. The remaining button allows you to refresh the browser contents when you have added new maps or layers and want to view their information in the browser.

Now that we have a handle on the browser, we'll move on to looking at some of the basic modules. The browser will come in handy later when we need to view or manage some of the output maps from our analysis and conversion activities.

Working with Modules in the Toolbox

In previous sections we've seen how to use the import modules in the toolbox to import both vector and raster data. In this section, we'll move beyond that and look at some of the other modules and what you can do with them. First a word about how the modules in the toolbox work: unlike working in GRASS, the modules in the toolbox require a layer loaded in QGIS to use as input to the module. For example, when we imported the world_mosaic.tif into GRASS, we loaded the TIFF into QGIS first and then used the import tool. Let's start our exploration of the toolbox by creating a buffer or two.

Buffering Vector Features

We can use the toolbox to buffer point, line, or polygon features. Buffers are useful for visualizing "things" that are within a given distance of other "things." For example, suppose Harrison has spotted eagles' nests in an area where a new trail is to be built. He is concerned about the potential for all the eager hikers disturbing the baby eagles and would rather keep them at least 500 meters from the nest. Harrison goes off to help the city planners with a bit of analysis.

With a map of eagles' nests, we can create a polygon layer that buffers each nest by a distance of 500 meters. Basically, it's like drawing a circle with a radius of 500 meters around each nest. Once we have the buffer layer, we can use it to site the trail to avoid the nests. Of course, this is just a simple example. In practice, buffers are an important part of GIS analysis in many disciplines. On the flip side, you might also analyze a proposed trail by buffering it and seeing whether it overlaps any nests.

When you buffer a feature, you must specify the distance in map units. In other words, if your map is in latitude and longitude, you would specify the distance in decimal degrees. This usually isn't very practical, so in most cases a projection that uses meters or feet for units of measure is used. Obviously to be successful (and accurate) in your analysis, you have to know a little bit about your data, its projection, and units of measure.

In Figure 12.9, on the following page, we have taken a hypothetical grouping of eagles' nests and have applied a 500-meter buffer to them. The nests and the buffers are displayed over a topographic map. In this case, we are interested in the distance of each nest from the roads in the area. From looking at the map, we can see that all of the nests

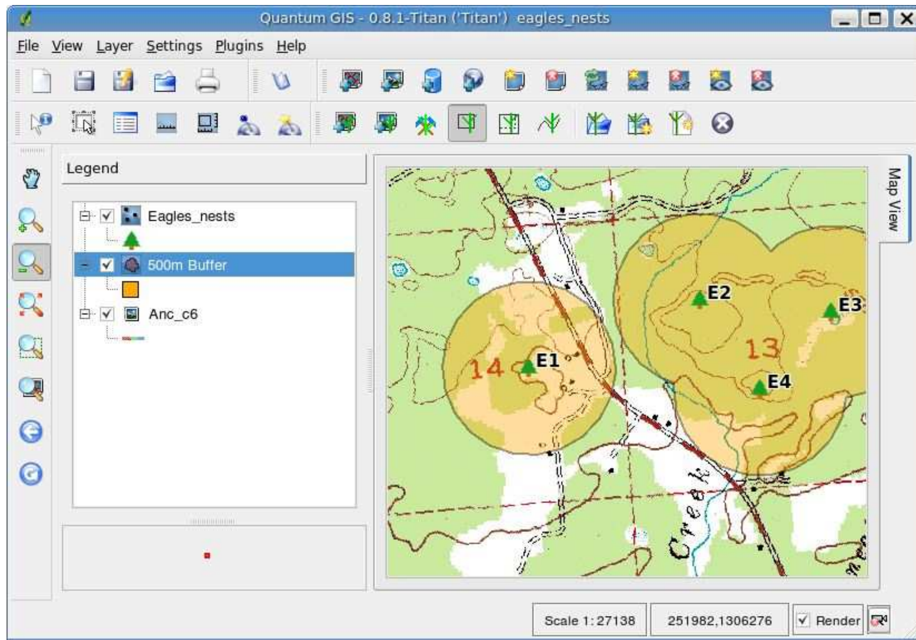


Figure 12.9: Buffered eagle nests created with GRASS and QGIS

except E1 are at least 500 meters from the nearest road. Nest E4 comes close to being within the buffer distance of the road but is still at least 500 meters away. If we were doing an impact analysis, this simple tool provides a quick way to visualize the relationships.

To create the buffer, we first put the eagles' nest map over the topographic raster map. Then from the GRASS toolbox, we selected the Vector Buffer module. When you click the module, it opens a new tab for the buffer operation in the toolbox. Since we had only one GRASS map loaded, the eagles_nest map is chosen as the input vector map. We then just specify the buffer distance in map units, in this case 500 meters, and eagles_nests_500m_buf as the name for the output map. When we click Run, the buffer layer is created, and we can review the output from the buffer processing if we want. To add the newly created buffer to the map canvas, we just click the View Output button. It sounds like a lot of steps, but in reality it takes about ten seconds to create a buffer, depending of course on how fast you type.

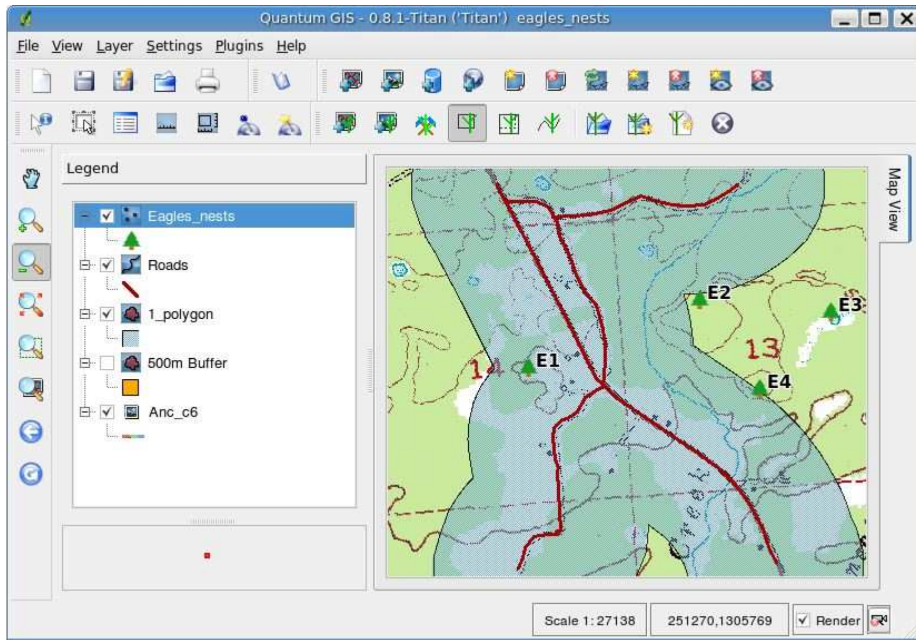


Figure 12.10: Roads buffered using GRASS in QGIS

Let's help the eagles shop for a new nest site. In this case, we need to buffer the roads by 500 meters to define the areas unsuitable for nesting. We first start by creating a new empty roads map using our QGIS-GRASS skills. Then we digitize the main roads from the topographic raster map using the GRASS edit tools. Since this is a one-shot analysis, we can get away without entering attributes for our roads. If we were going to use the road map in future work, we would need to put a little more thought into the attributes and enter them as we digitize.

Once we have the roads digitized, we can use the Vector Buffer module to buffer the roads. When we display the roads and the buffer, we can easily see sites in Figure 12.10 that aren't suitable for marketing to Mr. and Mrs. Eagle.

You can also buffer a raster map, in which case the cells of the raster are buffered based on distance zones you set up. If you are interested in this kind of buffer processing, the GRASS manual is your friend.

You can see the utility of a simple geoprocessing task like buffering. With the GRASS plugin in QGIS, this kind of GIS analysis is easy to do.

Vector Overlays

Now we turn our attention to a group of modules that have to do with vector overlays. These modules are found in the toolbox under the heading "Vector overlay." The modules available to us include the following:

Vector Union
: This module overlays two vector maps to create a new map containing the union or combination of the features. The attribute tables from the two maps are merged, and a prefix is added to column names so you can tell from which map they originated. The boundaries are not dissolved so all original boundaries of the features are still visible.

Vector Intersection
: This creates a new map containing the portions of features common to features on the input maps. Where two polygons overlap, only the common portion will be included in the resulting map.

Vector Subtraction
: The second polygon map is subtracted from the first. The result for each feature is that portion of the feature in the first map not overlapping a polygon feature in the second map.

Vector Non-intersection
: This removes the common portion between polygon features. If two polygons overlap, the overlapping part will be removed, resulting in what looks like a hole in the combined polygons.

In Figure 12.11, on the following page, you can see a graphic illustration of the results of each vector overlay operation. In the center are two polygons, and surrounding them is the result. The graphics in the GRASS toolbox also provide a visual cue for each overlay operation in case you need reminding. You can start thinking about how you would use these modules for something real. We'll take a simple example to illustrate the use of the Vector Subtraction command. This will be sufficient to illustrate the use of the modules, and then you can go crazy with the others.

Let's take a semireal but mostly manufactured example using logging.[2] Suppose the logging company is allowed to cut certain stands of trees,

2. Before you take this example seriously, I confess I know little about timber industry practices—it's just an example.

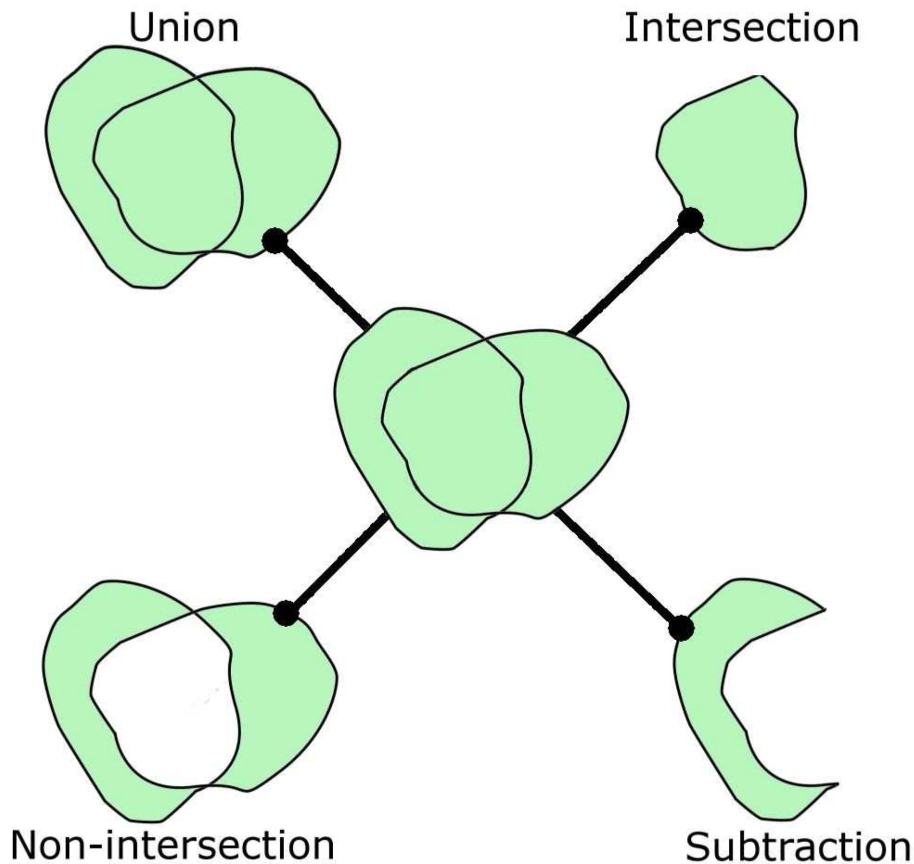


Figure 12.11: Result of each type of vector overlay operation

based on species and age. We have a polygon map outlining the stands eligible for cutting. A stream runs through the logging area. To protect both the stream and the fish populations, there is a 100-meter setback requirement from any activity. We need to identify the portions of the eligible stands that are "legal" for harvest.

First we prepare our data and make sure we have a good and accurate polygon map of the eligible stands. Next we need to buffer the stream to 100 meters to create the second polygon map needed for the analysis. Once we have those, we can proceed with the subtraction operation. In Figure 12.12, on the next page, you can see the stands map and the digitized streams along with the buffer. You can guess by looking at it

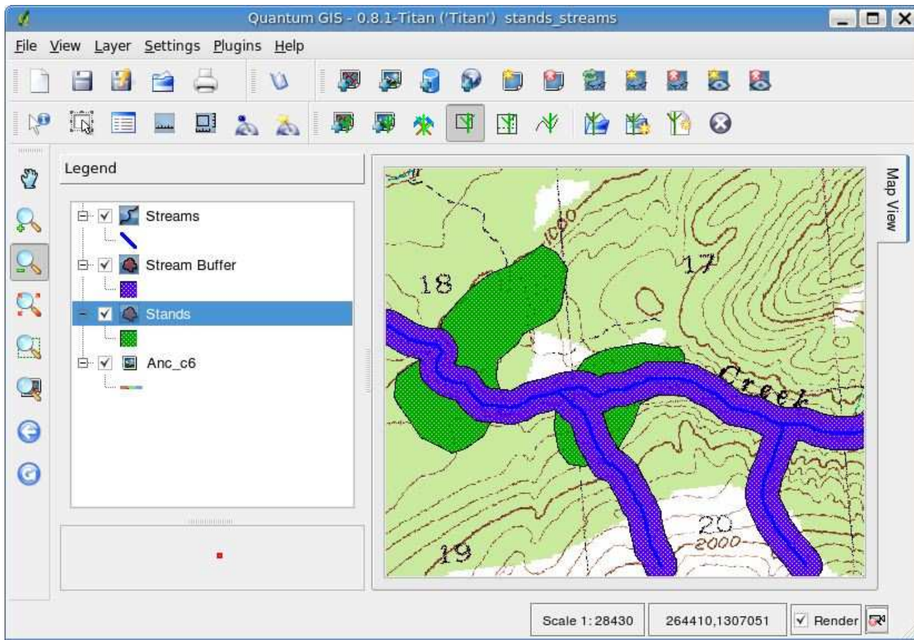


Figure 12.12: Timber stands and stream buffers

where the likely "no logging" areas are, but by doing the analysis we will be able visualize it to aid in making decisions.

Now to subtract the portions of the stands that are not eligible for logging. To do this, we use the Vector Subtraction module and enter the name of the stands map as the first vector input map on the module input screen. The stream buffer map is entered as the second vector map. Then we specify a name for the output map. Since it will contain polygons of the eligible areas, we'll name it eligible_stands. We click the Run button, and off we go. The result of this operation is shown in Figure 12.13, on the following page.

You can see from the results that the upstream stand has been carved up into three fairly small pieces, one of which is between a fork in the stream. If we pretend to know something about logging, we might say that the upstream stand (the one to the right) doesn't look like it's too promising in terms of both size and location. The downstream stand has been cut in two but is still fairly sizeable. This example serves to show how the vector overlay modules can be used for visual analysis

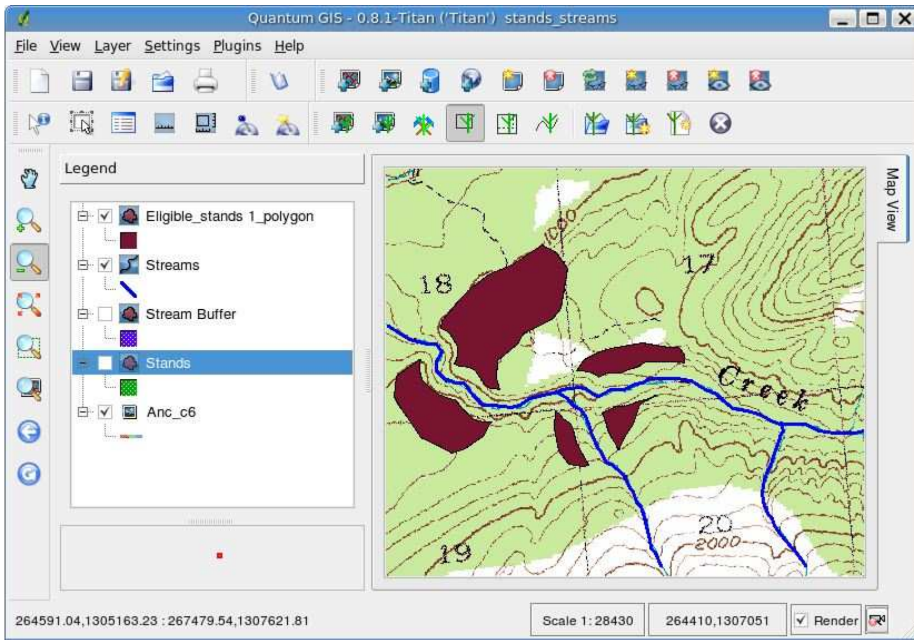


Figure 12.13: Eligible logging areas after vector subtraction

of spatial relationships. OK, enough pretending that I know anything about the timber industry.

Creating a Contour Map

To further illustrate the power of the GRASS modules in the toolbox, we'll create a contour map from the Anchorage DEM we used in the LOS analysis in Section 10.2, *Line-of-Sight Analysis*, on page 153. Creating a contour map is quite simple, but you need to be aware of the limitations. When using a DEM, the map will be only as good as the original data. If the cell size is 50 meters, you can't expect to create contours at 20 meters. The same holds true for any raster source we might want to use.

To create a contour map, we first fire up QGIS and open the GRASS mapset that contains the DEM. To make a contour map, we first have to add the DEM to the map canvas. Once that's done, we can open the GRASS toolbox and locate the r.contour module. It is under the "Generate vector contour lines" heading in the list of modules.

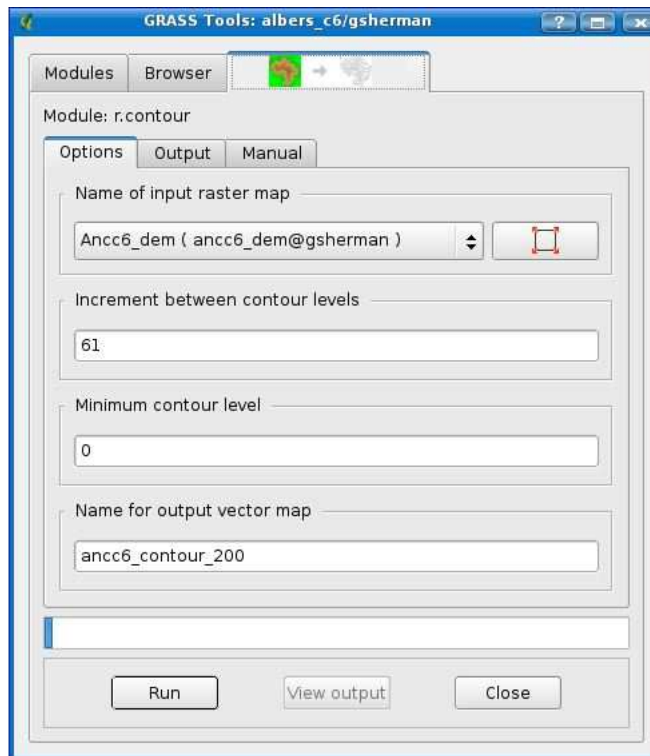


Figure 12.14: Setting up to contour a DEM

When you click r.contour, the module Options tab is displayed, as shown in Figure 12.14. Here we have filled in the parameters for creating the contour map. The first step is select an eligible GRASS raster from the Name of Input Raster Map drop-down. We'll contour the Ancc6_dem DEM at an interval of 200 feet. Since the DEM is in meters, we need to convert that to feet. Using 3.28 ft/m gives us roughly 61 meters, which we entered in the increment field. The Minimum Contour Level setting specifies how "low" we want to contour. In the case of our DEM, we think everything is above sea level, so we just leave that set at zero. The only other thing to specify is the name for the output map.

When we click Run, the magic happens, and the vector contour map is created. We can then add it to the map using the View Output button. The result is shown in Figure 12.15, on the next page, draped over the original topographic map and the shaded relief. You may find you need

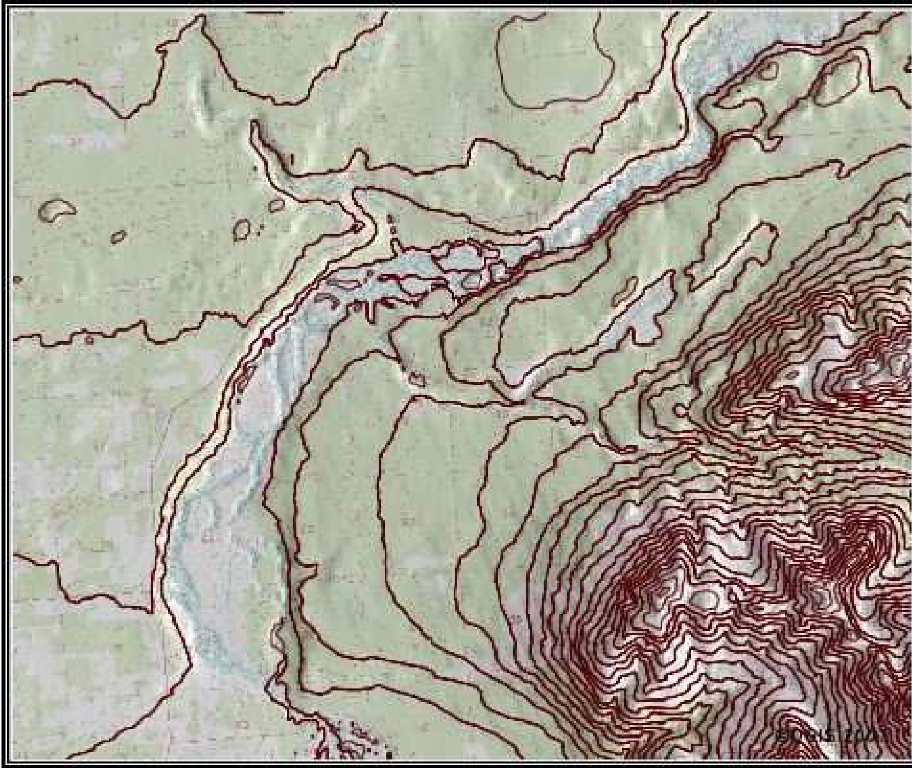


Figure 12.15: Result of contouring the DEM

to adjust the parameters to get the result you want. If so, you can use the browser to delete the contour map and start over. Make sure you remove the map from the QGIS canvas first!

In case you are wondering, we could of course accomplish the same thing from the GRASS shell or command line. The command used to create the contour map is as follows:

```
r.contour input="ancc6_drg@gsherman" output="ancc6_contour_200" \
   minlevel=0 step=61 cut=0
```

The QGIS module leaves off a few parameters that we could have used from the GRASS shell. (I know that we're talking about QGIS, but this is good stuff.)

The full option list for the command is as follows:

```
GRASS 6.2.2 (albers_c6):~ > r.contour help

Description:
  Produces a GRASS binary vector map of specified contours from GRASS
  raster map layer.

Keywords:
 raster

Usage:
 r.contour [-qn] input=name output=name [levels=value[,value,...]]
   [minlevel=value] [maxlevel=value] [step=value] [cut=value]
   [--overwrite]

Flags:
  -q   Suppress progress report & min/max information
  -n   Suppress single crossing error messages
 --o   Force overwrite of output files

Parameters:
     input   Name of input raster map
    output   Name for output vector map
    levels   List of contour levels
  minlevel   Minimum contour level
  maxlevel   Maximum contour level
      step   Increment between contour levels
       cut   Minimum number of points for a contour line (0 -> no limit)
             default: 0
```

Note we could have specified both minimum and maximum contour levels, as well as an interval and a step value. There is also a cut option to specify how many points constitute a contour line. The point of showing you the GRASS usage of this command is this—don't count on the QGIS-GRASS modules to provide you with all the options for a particular operation. For this reason, it's good to always read the manual entry (available from the Manual tab for each module) to see what you might be missing.

Map Algebra

If you've been following along with QGIS and actually opened the toolbox, you're probably thinking "Wow, there's a lot of modules in there." You're right, and there is no way we are going to give you an example of each. The goal is to get you started with the basics, and you can develop your skill set to meet your needs. That said, there is one last module we want to look at, just because it's a bit different from the others.

Map or "grid" algebra allows you to perform operations on raster maps in GRASS. This can be useful for a number of things, depending on your data. You might recall that we have already used some map algebra (r.mapcalc) in Chapter 10, *Geoprocessing*, on page 149 when processing rasters.

The QGIS-GRASS toolbox includes a graphic means to design a set of operations to create a new raster from a set of input maps. Essentially you are creating a model that can be run to perform the operation(s). To illustrate, we'll convert our DEM from meters to feet, a simple matter of multiplication.

To convert the DEM, the value of each cell in meters must be multiplied by 3.28 to convert it to feet. Given that our simple little DEM contains 225,500 cells, this is no trivial matter. Fortunately, the r.mapcalc module makes this easy to do.

First we'll look at a complete "model" and then explain the process of putting it together. In Figure 12.16, on the next page, you can see the completed model ready to run.

So, how did we build the model? Basically, it's a select-drag-drop operation for each component. The tools on the toolbar allow you to add a map, constant, function, and connector. We started out by adding the DEM, which must already be loaded into QGIS; otherwise, it won't show up in the list of available maps. We then added the constant 3.28 and a multiply operator. The output "widget" was already on the model when we started. Once all the parts are in place, we just use the Add connection tool to connect them, making sure they are in the proper sequence. The last step is to enter a name for the output map, and then we are ready to run it.

When we run the model, it's really just building up a GRASS r.mapcalc command for us in the background and executing it. It multiplies each cell in the Ancc6_dem by 3.28 and stores the value in the output map. When complete, we have a new raster map that looks just like the original when displayed in QGIS, but the units are in feet rather than meters. If you think you may want to run the model again, you can save it for future use by clicking the Save tool in the toolbar. When it comes time to run the model again, start the r.mapcalc module, and load the model using the Open tool on the toolbar.

This simple example illustrates how to use the r.mapcalc module to build a model and run it. We didn't look at all the functions, but there

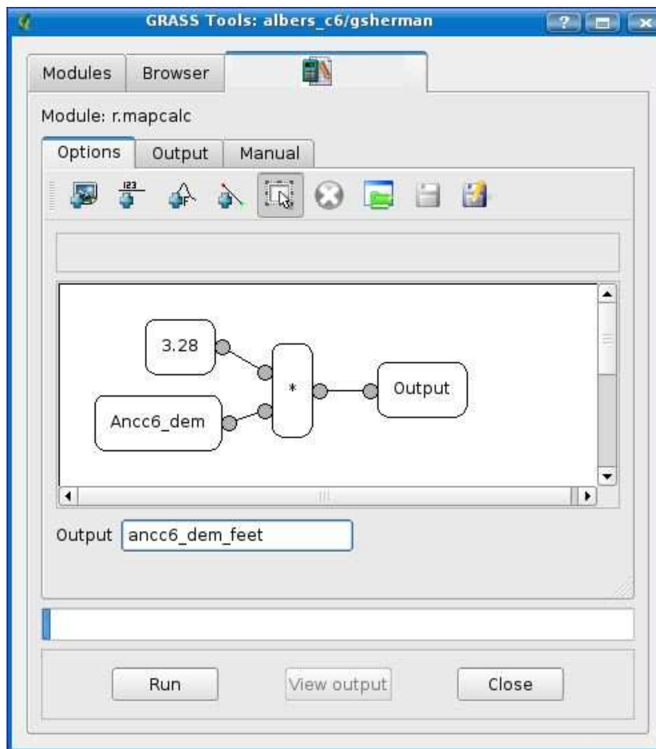


Figure 12.16: Mapcalc model for converting DEM from meters to feet

are eighteen operators (arithmetic and logical) and thirty-two functions, including trigonometric, logical, log, and others. So, you can get a lot fancier than we did with the r.mapcalc module.

12.4 Summing It Up

That concludes our tour of the QGIS-GRASS plugin and toolbox. As you may have guessed, we just scratched the surface here. It's up to you to explore further and see what other magic awaits you. As I said at the beginning of this chapter, we're bordering on advanced territory here. Keep in mind as you explore the QGIS-GRASS modules that there may be extra options and "power" available from the GRASS shell.

Joe Asks...

What Options Are There for Printing Maps?

At some point you're going to want to print from your OSGIS application. If you're using QGIS, you can use the map composer to create a map complete with legend, scale bar, and annotations. You can export from the map composer to a PNG or SVG for printing or further processing in a graphics application.*

GRASS provides the ps.map module to produce high-quality PostScript output suitable for printing. A text file containing mapping instructions is used as input to ps.map. There are a lot of instruction keywords that can be used in creating your map. For help getting started with ps.map, see the GRASS manual page. You can find examples of ps.map scripts on the GRASS wiki.†

If you are using GMT to create maps, the output is already suitable for printing or inclusion in other documents. If you are using one of the other OSGIS applications, check the manual for information on creating hard-copy output.

*. The current version of the map composer has a number of issues that limit the quality of the output. These issues are being addressed, and let's hope the composer will improve in the next release.

†. http://grass.gdf-hannover.de/wiki/Ps.map_scripts

There are advanced uses for many of the modules in the toolbox that are beyond the scope of our discussion. If you are already an advanced GIS user, you are probably picturing them right now. If you are a casual or intermediate user, you may find that the QGIS-GRASS integration provides access to a rich set of tools for performing data import and conversion. Be careful, though—once you start down that path, you may end up as an advanced user with more ideas than time.

The QGIS-GRASS tools lower the barrier of entry into the world of geoprocessing with GRASS. As you progress in your GIS journey, you'll likely find yourself using the GRASS shell to get at even more power and options.

Chapter 13

GIS Scripting

Most GIS users that I know end up doing a bit of programming, regardless of the software they are using. There is always some little task that is easier done with a script or a bit of code. In this chapter, we'll look at some methods for automating tasks in OSGIS software. You don't have to be a programmer to do a bit of script writing, especially when you can get jump-started by downloading examples and snippets.

The script languages available to you depend on the application you are using. Applications and tools with a command-line interface (CLI) can be scripted with most any language available. Others have bindings for specific languages. Some nonexhaustive examples include the following:

- *GRASS*: Shell, Tcl/Tk, Perl, Ruby, Python
- *QGIS*: Python
- *GDAL/OGR*: Shell, Perl, Ruby, Python
- *PostGIS*: Any language that works with PostgreSQL, such as Perl, Python, PHP, and Ruby

Some OSGIS applications even provide bindings that allow you to write a custom application using a language such as Python. In this chapter, we will explore some of the techniques used with these applications.

13.1 GRASS

Since the real core of GRASS is comprised of CLI applications, it's pretty easy to use most any scripting language to perform tasks. From Perl, Python, Ruby, and Tcl/Tk, you can "call" an application and capture the output. This makes GRASS easy to automate.

Shell Game

What do we mean by a shell? It's a command interpreter provided with your operating system. If you use OS X, Linux, or a Unix variant, you likely have bash, csh, and/or ksh available to you. Windows has cmd, which has its own language and probably isn't going to be real helpful in shell scripting. Check out MSYS* and Cygwin† for Windows alternatives.

*. http://mingw.org
†. http://cygwin.org

Probably the simplest way to automate GRASS tasks is using the scripting capabilities of your shell. On Linux and OS X, this is a pretty natural thing to do, because both come with a fully capable shell. On Windows, you may have to install a Unix-like shell such as MSYS or Cygwin to be able to accomplish the same results. You can check up on the progress of the GRASS Windows support at the GRASS website.[1]

13.2 QGIS

At version 0.9.*x*, QGIS includes support for scripting with Python. QGIS provides the following options for using Python:

- Use the Python console from which you can run scripts using the objects and methods in the QGIS API.
- Write plugins in Python instead of C++.
- Use PyQt[2] to build complete mapping applications using Python and the QGIS libraries.

Why would you want to do any of this? You'd be surprised at the things you might dream up. QGIS has been designed to make the libraries easily usable in your own plugins and applications. With the new Python bindings, this brings a whole new world of possibility—from simple plugins to complete applications. So far, some of the things that people have come up with are as follows:

- A map algebra plugin

1. http://grass.itc.it
2. http://www.riverbankcomputing.co.uk/pyqt/

- A geocoding plugin
- A region tool for drawing and getting map coordinates from the canvas
- An application to collect fisheries data

We're going to take a look at a simple plugin to help us get an idea of what can be done with the PyQGIS bindings. Using Python to get started is pretty easy, so don't be afraid if you aren't a programmer. Let's start by looking at the console.

The Python Console

The console is a bit like using Python from the command line. It lets you interactively enter bits of code and see the result. This is a good way to experiment with the interface and can actually be helpful when you are writing a plugin or application.

To bring up the Python console, go to the Plugins menu, and choose Python console. The console has a command-line entry area at the bottom and the result window above. Make sure you read the little tip at the top.

The console is not of much use if we don't know what to enter into it. Let's try a simple example and change the title of the main QGIS window. The iface object provides you with access to the QGIS API. Using it, we can reference the main window and set the title:

```
iface.getMainWindow().setWindowTitle('Hello from Desktop GIS!')
```

In Figure 13.1, on the following page, we can see the result of our little example, with the console in front and the new title showing on the QGIS window behind it.

Changing the title isn't all that useful, but it shows you how to get a hold of the interface into the QGIS internals. Like I said, the console isn't for doing anything overly useful but is a good exploratory tool. To manipulate the map canvas, we can try the command given as an example in the console:

```
iface.zoomFull()
```

This will zoom the map to its full extent. Now you're probably wondering how to find out what functions are available. The answer is the QGIS API documentation, available from the website.[3]

3. http://svn.qgis.org/api_doc/html

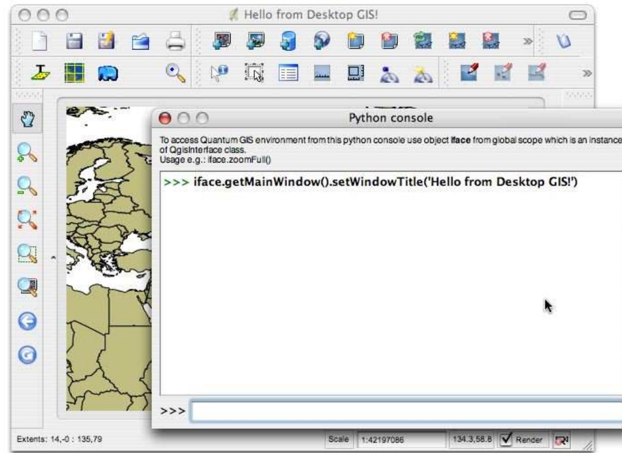


Figure 13.1: Changing the window title with Python

The API may be a bit intimidating at first, but it's very useful, in fact essential, to our Python exploits. When you use the iface object in the Python console, you are actually using an instance of the QgisInterface class.[4] If we look at the documentation for QgisInterface, we find functions such as the following:

- zoomFull(): Zoom to full extent of map layers.
- zoomPrevious(): Zoom to previous view extent.
- zoomActiveLayer(): Zoom to extent of the active layer.
- addVectorLayer(QString vectorLayerPath, QString baseName, QString providerKey): Add a vector layer.
- addRasterLayer(QString rasterLayerPath): Add a raster layer given a raster layer filename.
- addRasterLayer(QgsRasterLayer *theRasterLayer, bool theForceRender\ Flag=false): Add a raster layer given a QgsRasterLayer object.
- addProject(QString theProject): Add a project.
- newProject(bool thePromptToSaveFlag=false): Start a blank project.

We already used the zoomFull() method to zoom to the full extent of all the layers on our map. You can see there is a lot of potential here for manipulating the map, including adding layers and projects. We can use these same methods in our plugins and stand-alone applications as well. As you dive into PyQGIS, the documentation will be your friend.

4. http://svn.qgis.org/api_doc/html/classQgisInterface.html

Think of the Python console as a workbench for trying methods and using classes in the QGIS API. Once you get that under your belt, you're ready for some real programming. We'll start out by creating a little plugin using Python.

A PyQGIS Plugin

Writing plugins in Python is much simpler than using C++. Let's work up a little plugin that implements something missing from the QGIS interface. For this exercise, you'll need QGIS 0.9.*x*, Python, PyQt, and the Qt developer tools.

Harrison just received the latest *Birding Extraordinaire* magazine, and in it he finds an article that describes locations for the exotic Moose-Finch.[5] The locations are in latitude and longitude, which don't mean much to Harrison unless he's in his backyard. He fires up QGIS, adds his layer containing the world boundaries, and begins hunting for the coordinates. Sure, he can use the coordinate display in the status bar to eventually find what he wants, but wouldn't it be nice to be able to just zoom to the coordinates by entering them? Well, that's what our little plugin will do for us (and Harrison).

Before we get started, we need to learn a little bit about how the plugin mechanism works. When QGIS starts up, it scans certain directories looking for both C++ and Python plugins. For a file (shared library, DLL, or Python script) to be recognized as a plugin, it has to have a specific signature. For Python scripts, it's pretty simple. Take a look at your QGIS installation. By platform, here is where you'll find the Python plugin directory (we'll assume your top-level install directory is represented by .):

- *Linux and other Unix*: ./share/qgis/python/plugins
- *Mac OS X*: ./Contents/MacOS/share/qgis/python/plugins
- *Windows*: .\share\qgis\python\plugins

For QGIS to find our Python plugin, we have to place it in a subdirectory of the appropriate plugin directory for our platform. Each plugin is contained in its own directory. When QGIS starts up, it will scan each subdirectory in share/qgis/python/plugins and initialize any plugins it finds. Once that's done, our Python plugin will show up in the QGIS

5. A mythical creature

plugin manager where we can activate it just like the other plugins that come with QGIS. OK, enough of that, let's get started writing our plugin.

Setting Up the Structure

The first thing we need to do is set up the structure for our plugin. In this example, we'll be developing our plugin on Linux, but the method is the same. Just adapt some of the file system commands as appropriate for your platform. In our examples, QGIS is installed in a directory named qgis_09 in our home directory. Let's create the directory for the plugin:

```
mkdir ~/qgis_09/share/QGIS/python/plugins/zoom_to_point
```

That gives us a directory that QGIS will scan on start-up. Within that directory, we are going to create the following files to get us started (we'll need some additional files in a bit):

- __init__.py
- resources.py
- resources.qrc
- zoomtopoint.py

Making the Plugin Recognizable

To initialize our plugin and make it recognizable by QGIS, we use __init__.py. For our ZoomToPoint plugin, it looks like this:

```
# load ZoomToPoint class from file zoom_to_point.py
from zoomtopoint import ZoomToPoint
def name():
  return "Zoom to Point"
def description():
  return "Zooms the map canvas to the point you specify"
def version():
  return "Version 0.1"
def classFactory(iface):
  return ZoomToPoint(iface)
```

The mandatory things a plugin must return are a name, description, and version, all of which are implemented in our previous script. The other requirement is the classFactory() method that must return a reference to the plugin itself, after receiving the iface object as an argument. That's all there is to it to make QGIS think we're a plugin. But for it to work, we need to actually implement some logic to make it do something.

Defining Resources

For our plugin, we not only need the logic to zoom the map but also a dialog box to collect input from the user. We also need a resources file that will contain the icon for our tool. Let's get that out of the way first by creating our resources.qrc file that contains the definition of our icon:

```
 <RCC>
    <qresource prefix="/plugins/zoom_to_point" >
        <file>icon.png</file>
    </qresource>
</RCC>
```

This resource file uses a prefix to prevent naming clashes with other plugins. It's good to make sure your prefix will be unique—usually using the name of your plugin is adequate. We define one file, icon.png, in the resource file. This is just a PNG image that will be used in the toolbar when we activate our plugin. You can create your own PNG or use an existing one. The only real requirement is that it be 22-by-22 pixels so it will fit nicely on the toolbar. You can also use other formats (XPM for one), but PNG is convenient, and there are a lot of existing icons in that format.

Once we have the resource file built, we need to use the PyQt resource compiler to compile it:

```
pyrcc4 -o resources.py resources.qrc
```

The -o switch is used to define the output file. If you don't include it, the output of pyrcc4 will be written to the terminal, which is not really what we're after here. Now that we have the resources defined, we need to build the GUI to collect the information for ZoomToPoint.

Creating the GUI

To create the GUI, we'll use the same tool that C++ developers use: Qt Designer. This is a visual design tool that allows you to create dialog boxes and main windows by dragging and dropping widgets and defining their properties. Designer is installed along with Qt, so it should be already available on your machine.

Our dialog box is pretty simple. In Figure 13.2, on the next page, you can see the dialog box in Designer, along with the widget palette and the property editor. It's already complete, but let's take a look at what we had to do to build it. It's going to be a quick tour since we won't go into all the intricacies of Designer. If you want to get into the nitty-gritty, see

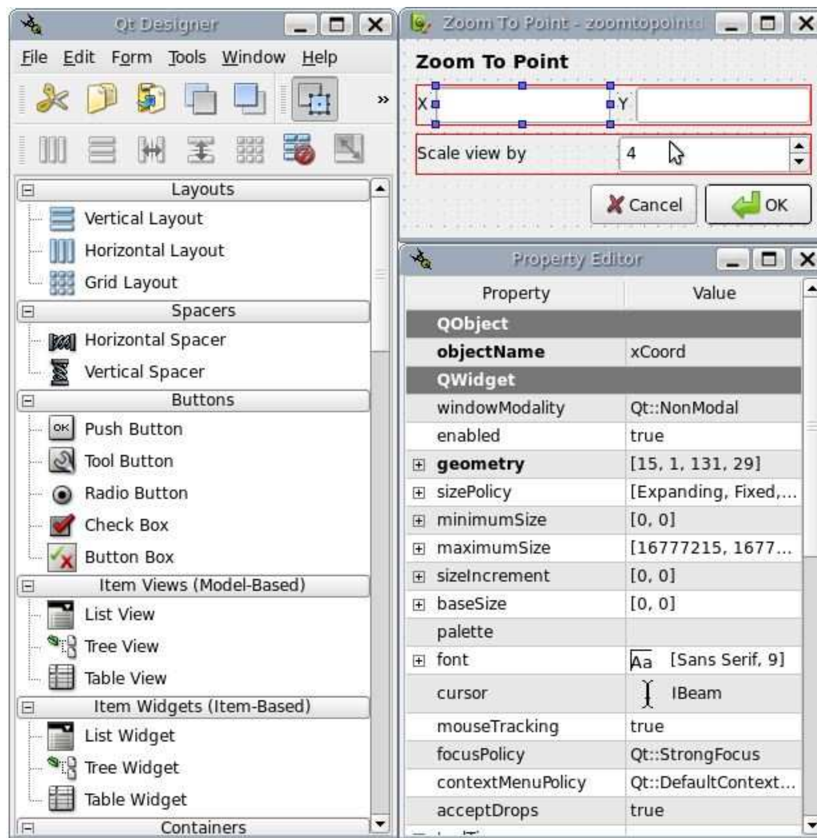


Figure 13.2: Plugin dialog box in Qt Designer

the excellent documentation on Designer on the Trolltech website[6] or in your Qt documentation directory.

We start by creating a new dialog box using the New form option on the File menu and selecting the dialog box with the button on the bottom. Then we add text labels and text edit controls, as shown in Figure 13.2. We also added a spin control for scaling the view. You don't have to set any properties of the text edit controls, but it can be convenient to name them something other than the default. In this case, I named them xCoord, yCoord, and spinBoxScale. This makes it easier to reference them in the code (for those of us with short memories). For our dialog box, we

6. http://trolltech.com

don't need to change the default actions of the OK and Cancel buttons. Once we have all the controls on the form, we're ready to generate some code from it.

To convert our dialog box (which we saved as zoomtopointdialog.ui) to Python, we use the PyQt pyuic4 command to compile it:

```
pyuic4 -o ui_zoomtopoint.py zoomtopointdialog.ui
```

This gives us ui_zoomtopoint.py containing the code necessary to create the dialog box when the plugin is launched. There is one more thing we need to get the dialog box up on the screen. We need a bit of code that imports the user interface and displays the form. For this we create zoomtopointdialog.py containing the following:

```
from PyQt4 import QtCore, QtGui
from ui_zoomtopoint import Ui_ZoomToPoint

class ZoomToPointDialog(QtGui.QDialog):
    def __init__(self):
        QtGui.QDialog.__init__(self)

        # Set up the user interface from Designer.
        self.ui = Ui_ZoomToPoint()
        self.ui.setupUi(self)
```

This bit of code uses the ui_zoomtopoint.py script in setupUi(self) to set things up. Our GUI is now ready for use. All we need to write now is the Python code to interact with the QGIS map canvas.

Getting to Zoom

We're now ready to write the actual code that does something with the map. Up to this point we've just been getting the plumbing put in. Now we'll write the code to actually zoom to the point we enter in our dialog box. As we go, we'll look at the code in chunks to make it a bit easier.

Let's start by looking at the things we need to import and the initialization of the plugin:

Download zoomtopoint.py

```
Line 1  # Import the PyQt and QGIS libraries
     -  from PyQt4.QtCore import *
     -  from PyQt4.QtGui import *
     -  from qgis.core import *
     5  # Initialize Qt resources from file resources.py
     -  import resources
     -  # Import the code for the dialog
     -  from zoomtopointdialog import ZoomToPointDialog
     -
```

```
10  class ZoomToPoint:
-
-     def __init__(self, iface):
-       # Save reference to the QGIS interface
-       self.iface = iface
15
-     def initGui(self):
-       # Create action that will start plugin configuration
-       self.action = QAction(QIcon(":/plugins/zoom_to_point/icon.png"), \
-           "Zoom To Point plugin", self.iface.getMainWindow())
20      self.action.setWhatsThis("Configuration for Zoom To Point plugin")
-       QObject.connect(self.action, SIGNAL("activated()"), self.run)
-
-       # Add toolbar button and menu item
-       self.iface.addToolBarIcon(self.action)
25      self.iface.addPluginMenu("&Zoom to Point...", self.action)
-
-     def unload(self):
-       # Remove the plugin menu item and icon
-       self.iface.removePluginMenu("&Zoom to Point...",self.action)
30      self.iface.removeToolBarIcon(self.action)
```

Every Python script that uses the QGIS libraries and PyQt needs to import the QtCore and QtGui libraries, in addition to the QGIS core library. This gives us access to the PyQt wrappers for our Qt objects (like our dialog box) and the QGIS core libraries. We do this in lines 2 through 8. Notice that not only did we import the PyQt and QGIS libraries, but we also brought in our resources file and, in line 8, the code for the dialog box.

If you noticed line 10, you probably realized that we're talking about a Python class. The implementation of our plugin all takes place within the ZoomToPoint class. The methods we are about to discuss are all members of ZoomToPoint.

When the class is first instantiated, we store the reference to the iface object using the __init__() method. This method gets called whenever we create a ZoomToPoint object. We store iface as a class member because we are going to use it later when we need access to the map canvas.

As far as QGIS is concerned, plugins must implement only two methods: initGui() and unload(). These two methods are used to initialize the user interface when the plugin is first loaded and clean up the interface when it's unloaded. Let's take a look at what we need to initialize our plugin GUI.

First we need to create what's called an *action*. This is a Qt object of type QAction. It's used to define an action that will later be used on a menu or a toolbar. On line 19, we create our action by supplying three arguments:

- The icon for the toolbar. This is a combination of the prefix (/plugins/zoom_to_point) and the icon file name (icon.png) as specified in our resources file.
- Some text that's used in the menu and tooltip, in this case "Zoom To Point plugin."
- A reference to the parent for the plugin, in this case the main window of QGIS.

Once we have created the action, we set some descriptive text to be used with the WhatsThis function (that's the little arrow you find on a lot of applications that you click to activate and then click a toolbar item or other GUI element to get a description). On line 21, we do one last thing with the action to connect it to the run() method. This basically connects things so that when the OK button on the dialog box is clicked the run() method in our ZoomToPoint class is called.

Next we need to actually put our nicely configured action on the menu and toolbar in the GUI. The QgisInterface class that we played with in the Python console contains the methods we need. On line 24, we use addToolBarIcon() to add the icon for our tool to the plugin toolbar in QGIS. To add it to the menu, we use addPluginMenu() method, as shown on line 25. Now our GUI is set up and ready to use.

The unload() method is pretty simple. It uses the removePluginMenu() and removeToolBarIcon() methods to remove the menu item and the icon from the toolbar. Remember this method is called only when you unload the plugin from QGIS using the Plugin Manager.

Finally, we are ready to add the bit of code that does the real work. Like most GUI applications, the bulk of the code has to do with the user interface while a few bytes do the actual work. In our case, the actual work is done by the run() method that is defined on line 1 in the following listing:

Download zoomtopoint.py

```
Line 1  def run(self):
     -    # create and show the ZoomToPoint dialog
     -    print "Creating ZoomToPoint Dialog"
     -    dlg = ZoomToPointDialog()
```

```
5   dlg.show()
-   result = dlg.exec_()
-   # See if OK was pressed
-   if result == 1:
-     # Get the coordinates and scale factor from the dialog
10    x = dlg.ui.xCoord.text()
-     y = dlg.ui.yCoord.text()
-     scale = dlg.ui.spinBoxScale.value()
-     # Create a rectangle to cover the new extent
-     rect = QgsRect(float(x)-scale,float(y)-scale,float(x)+scale,float(y)+scale)
15    # Get the map canvas
-     mc=self.iface.getMapCanvas()
-     # Set the extent to our new rectangle
-     mc.setExtent(rect)
-     # Refresh the map
20    mc.refresh()
```

Let's step through the run() method to see how it works. On line 3, we print a message to the terminal to let us know what's happening. If you start QGIS from a command shell in either Linux or OS X, you can see messages that are sent to the terminal. In this case, we use it just to ease our paranoia about whether the run() method is getting called.

The next step is to create the dialog box (line 4) and then display it using exec_(). This causes the dialog box to show itself and then wait for some user interaction. The dialog box remains up until either the OK or Cancel button is clicked. Once a button is clicked, we test to see whether it was the OK button on line 8. If so, we are then ready to zoom the map.

First we have to retrieve the x and y coordinates and the scale that you entered on the dialog box (lines 10 through 12). We store these in local variables just to make the next step a bit more readable in the code. Once we have the user inputs, we need to create a rectangle that we can use in setting the map extent. The QGIS API has a QgsRect class that is used for this purpose. On line 14, we create the rectangle by simply expanding the x and y values by the scale value. Once we have the rectangle, we are ready to zoom the map. First we get a reference to the QGIS map canvas on line 16, using the iface reference we saved in the __init__() method. Then it's simply a matter of calling the map canvas setExtent() method. To actually get the map to zoom, we call the map canvas refresh() method and the map zooms to the rectangle we specified. Once complete, our plugin stands by ready for the next request.

Let's summarize the process of creating a plugin. First we have a bit of work to do to get the GUI in order. This includes setting up our resources file, designing the dialog box, and writing the class needed to bootstrap it. Then we create the actual plugin code, including the methods needed to initialize the GUI when the plugin is activated, and clean up after itself when it is unloaded. Finally, we implement the run() method where the real work of showing the dialog box, collecting the input, and zooming the map takes place. While we stretched out the explanation, there really isn't all that much hand written code involved in making the plugin. In fact, for the ZoomToPoint plugin there are less than 80 lines of actual code.

Our plugin is pretty rough. We don't do any error checking—you can submit blank values for x and y. There are a number of enhancements you could add to the plugin, including the ability to "remember" the x, y, and scale values that you used the previous go. If you got really fancy, you could also figure out how to set a marker at the point after you zoom. Come to think of it, once you add those features, send them to me, and I'll include them in the next release of the plugin. Just to prove it works, you can see the plugin and the values we just entered in Figure 13.3, on the next page. Behind it you'll see the map zoomed to the coordinates we specified. Notice the magnifying glass icon with the blue dot in the middle on the upper left of the toolbar. That's the icon we specified for our plugin, and it indeed shows up on the toolbar. If we were to look in the Plugins menu, we would find an entry for Zoom to Point as well.

Writing a QGIS plugin in Python is pretty easy. Some plugins won't require a GUI at all. For example, you might write a plugin that returns the map coordinates for the point you click the map. Such a plugin wouldn't require any user input and could use a standard Qt MessageBox to display the result. You can also write plugins for QGIS in C++, but that's another story and one I'll let you write.[7]

A PyQGIS Application

A stand-alone application is a step beyond a QGIS plugin. In some ways, they are very similar. We need to create a GUI and use the same imports. On the other hand, we do not have to write all that code to

7. Actually, you can find information on writing QGIS plugins in C++ on the QGIS wiki at http://wiki.qgis.org.

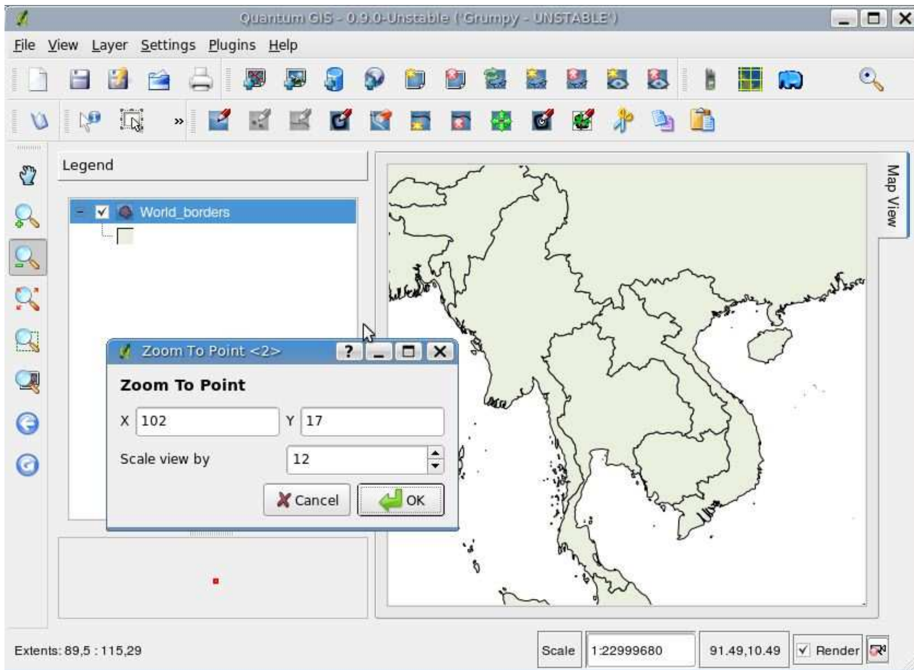


Figure 13.3: ZoomToPoint plugin in use

interface with the QGIS plugin mechanism. A stand-alone application does require a lot more GUI coding. Rather than build an application here, I'll point you at the QGIS blog for more information.[8] There you'll find five or six tutorials on creating stand-alone applications using QGIS and the Python bindings. For some real-world examples of PyQGIS applications, see Section 14.2, *Examples of Custom Applications*, on page 265.

13.3 GDAL and OGR

We already took a long look at the GDAL and OGR utilities in Chapter 11, *Using Command-Line Tools*, on page 174. Here we will take what we learned in that chapter and look at ways to automate our data conversion and loading tasks. Many, if not most, tasks can be handled with a shell script (bash for example) and don't require Ruby or Python. Of

8. http://blog.qgis.org

course, if you are more comfortable with one of those over shell scripting, it makes sense to use what you know. If you want to tap into the power of the GDAL/OGR bindings, you'll need to use Python, Ruby, Java, or one of the other supported languages.

For Windows users it will likely be easier to use Ruby or Python, unless you have access to a Unix-like shell through Cygwin or MSYS.

In this section, we'll take a look at a couple of examples, one using a shell script and the other using Python with the GDAL/OGR bindings.

Converting Data with a Shell Script

In the simplest case, we have a directory full of files, and we want to perform the same operation on each one. The basic application flow is as follows:

1. Get a list of the files.
2. Loop over the list.
3. Perform some operation on each file.
4. Repeat until all files are processed.

It can't get too much simpler than that. Let's take an example using bash and convert a batch of shapefiles from an Albers projection to geographic coordinates in WGS 84. First let's figure out what command and options we need to get the job done. We'll use ogr2ogr to convert the files. Fortunately, all the shapefiles have an associated projection file, so we don't have to worry about specifying that during the conversion. Here is the bash script to do the conversion:

Download convert_shapefiles.sh

```
Line 1  #!/bin/bash
     -  # Convert all shapefiles in the current directory to
     -  # WGS84 projection. The converted shapefiles are placed
     -  # in the geo subdirectory.
     5  for shp in *.shp
     -  do
     -    echo "Processing $shp"
     -    ogr2ogr -f "ESRI Shapefile" -t_srs EPSG:4326  geo/$shp $shp
     -  done
```

Notice that on line 7, we are going to print a little message for each shapefile that is processed. On line 8, we have the ogr2ogr command that does the actual work. As it's processed, each converted shapefile

is placed in a geo subdirectory. Other than that, the output from the script isn't that exciting:

```
$ . ./convert_shapefiles.sh
Processing adminbnd.shp
Processing admin_nps.shp
Processing admin_nra.shp
Processing admin_nwr.shp
Processing admin_state.shp
Processing admin_usfs.shp
Processing admin_wild.shp
Processing admin_wild_s.shp
Processing gnisalb.shp
Processing govt_emp.shp
Processing language.shp
Processing owner_fed.shp
```

Let's test one of the new shapefiles to make sure it did what we wanted:

```
$ ogrinfo -al -so adminbnd.shp
INFO: Open of `adminbnd.shp'
      using driver `ESRI Shapefile' successful.

Layer name: adminbnd
Geometry: Polygon
Feature Count: 11977
Extent: (-168.072393, 53.921043) - (-129.973606, 71.389543)
Layer SRS WKT:
GEOGCS["GCS_WGS_1984",
    DATUM["WGS_1984",
        SPHEROID["WGS_1984",6378137,298.257223563]],
    PRIMEM["Greenwich",0],
    UNIT["Degree",0.017453292519943295]]
AREA: Real (19.3)
PERIMETER: Real (19.3)
BNDS_: Integer (9.0)
BNDS_ID: Integer (9.0)
PARCEL_ID: Integer (9.0)
NAME: String (25.0)
LONGNAME: String (50.0)
ADMIN: Integer (9.0)
AGENCY_ITE: String (50.0)
PARCEL_TYP: String (50.0)
SCALE: Integer (9.0)
EFF_DATE: Date (10.0)
AMEND_DATE: Date (10.0)
DESCRIPTIO: String (50.0)
```

The spatial reference system (SRS) for the converted shapefile is indeed what we asked for—WGS 84.

Creating a Shapefile from Delimited Text

While the GDAL/OGR utilities provide you with a lot of capability, sometimes you may need to dig a little deeper. In this example, we'll use Python with OGR to create a shapefile from the volcanoes dataset. In Section 8.2, *Importing Data*, on page 122, we used the delimited text plugin to import the volcano data into QGIS and display it and then save it to a shapefile. That works well, but suppose you have a lot of data to process. In that case, writing a script to do the work is not only quicker but more flexible.

Before we get started, we need to make sure that the Python bindings for GDAL/OGR are present. This is easy to test using the Python Interpreter:

```
$ python
Python 2.5.1 (r251:54863, Oct  5 2007, 13:50:07)
[GCC 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import ogr
>>>
```

If you get the prompt back with no errors, you are good to go. If not, it means that your GDAL/OGR install doesn't include the Python bindings. If you built from source, you'll have to go back and recompile with the --with-python option. If you don't have them, a quick way to get the needed bindings for Linux or Windows is to use FWTools.[9]

Our script will take the following steps to get from delimited text to the shapefile:

1. Import the needed modules.
2. Open the delimited text file.
3. Create the shapefile.
4. Add the fields to the shapefile.
5. Read the text file and populate the attributes and geometry for each row.
6. Close the shapefile.

9. http://fwtools.maptools.org

Let's take a look at the code in chunks and go through it bit by bit:

Download import_volcanoes.py

```
# import the csv module
import csv
# import the OGR modules
import ogr
import osr

# use a dictionary reader so we can access by field name
reader = csv.DictReader(open("volcano_data.txt","rb"),
    delimiter='\t',
    quoting=csv.QUOTE_NONE)
```

Beginning with line 2, we import the modules needed for the script. The csv module is part of Python as of version 2.3. It provides a simple way to read a delimited text file and is well suited to our needs. The other imports we need are ogr, which provides access to the OGR functions needed to create and write features to a shapefile and osr, which provides the spatial reference functions.

Next we set up the csv reader in line 8. We are using the DictReader class to read the file and map the information into a dict. This allows us to reference the data using the field names in the header row of the input file. We'll use this capability to pick and choose which fields we want in our shapefile. When creating the DictReader, we need to specify the delimiter, in this case \t for the tab character. You might remember we used the same with the QGIS Delimited Text plugin. The last argument in setting up the reader is cvs.QUOTE_NONE, which specifies that our file has no quotes around the data.

Now we're ready to use the OGR bindings to create the shapefile:

Download import_volcanoes.py

```
# set up the shapefile driver
driver = ogr.GetDriverByName("ESRI Shapefile")

# create the data source
data_source = driver.CreateDataSource("volcanoes.shp")

# create the spatial reference
srs = osr.SpatialReference()
srs.ImportFromEPSG(4326)

# create the layer
layer = data_source.CreateLayer("volcanoes", srs, ogr.wkbPoint)

```

```
-   # Add the fields we're interested in
15  field_name = ogr.FieldDefn("Name", ogr.OFTString)
-   field_name.SetWidth(24)
-   layer.CreateField(field_name)
-   field_region = ogr.FieldDefn("Region", ogr.OFTString)
-   field_region.SetWidth(24)
20  layer.CreateField(field_region)
-   layer.CreateField(ogr.FieldDefn("Latitude", ogr.OFTReal))
-   layer.CreateField(ogr.FieldDefn("Longitude", ogr.OFTReal))
-   layer.CreateField(ogr.FieldDefn("Elevation", ogr.OFTInteger))
```

The first step is to create the driver in line 2. Once we have the driver, we can use it to create the data source. In OGR, a shapefile data source can be a directory of shapefiles, or it can be just a single file. In our case, we are creating just a single shapefile as the data source in line 5. In line 8, we create a SpatialReference object and then use the ImportFromEPSG() method to set the spatial reference to WGS84 using EPSG code 4326.

From the data source, we can create the layer as shown in line 12. The first argument is just the name of the shapefile without the extension. The second argument to CreateLayer() is the spatial reference. The final argument is the feature type—in our case a wkbPoint.

With the layer created, we can add the field definitions in lines 15 through 23. For the text fields, Name and Region, we create the field object, set an arbitrary width of 24, and then use the CreateField() to add it to our layer. For the numeric fields, we can create the field all in one step as in line 21.

The layer is now ready for some data:

Download import_volcanoes.py

```
Line 1  # Process the text file and add the attributes and features to the shapefile
-       for row in reader:
-         # create the feature
-         feature = ogr.Feature(layer.GetLayerDefn())
5         # Set the attributes using the values from the delimited text file
-         feature.SetField("Name", row['Name'])
-         feature.SetField("Region", row['Region'])
-         feature.SetField("Latitude", row['Latitude'])
-         feature.SetField("Longitude", row['Longitude'])
10        feature.SetField("Elevation", row['Elev'])
-
-         # create the WKT for the feature using Python string formatting
-         wkt = "POINT(%f %f)" %  (float(row['Longitude']) , float(row['Latitude']))
-
15        # Create the point from the Well Known Txt
-         point = ogr.CreateGeometryFromWkt(wkt)
-
```

```
-       # Set the feature geometry using the point
-       feature.SetGeometry(point)
20      # Create the feature in the layer (shapefile)
-       layer.CreateFeature(feature)
-       # Destroy the feature to free resources
-       feature.Destroy()
-
25  # Destroy the data source to free resources
-   data_source.Destroy()
```

In line 2, we began reading the text file to create the features, one for each line in the file. For each line we must create a feature object (line 4) and then set the values for each of the fields in lines 6 through 10. Since we chose to use the DictReader, we can access the values for each field by name to set the values. This takes care of the attributes—all that's left is to create the geometry from the latitude and longitude values.

To create the geometry, we create a WKT representation of the point using the values of the latitude and longitude fields. In line 13, the Python string-formatting feature is used to easily create the WKT for the point in the form of POINT(x y). Using the WKT, the point feature is created in line 16. The last step to get the entire feature ready is to set the geometry, as shown in line 19. The feature (attributes and geometry) is now complete and can be added to the shapefile using CreateFeature(), as shown in line 21. The last step before moving on to the next line in the text file is to destroy the local feature object to free up resources (line 23).

Once all the lines in the text file are processed, the data source is "destroyed" to close everything up cleanly. When it's run, the script produces the following files:

```
volcanoes.dbf
volcanoes.prj
volcanoes.shp
volcanoes.shx
```

If you look at the .prj file, you'll see that it contains the projection information for WGS84. Just to prove it works, the volcano shapefile displayed over the world_borders shapefile is shown in Figure 13.4, on the next page.

You can probably envision even more clever and perhaps complicated scripts using both the GDAL/OGR utilities and bindings. There are GDAL/OGR bindings for Perl, Python, Ruby, Java, and C#/.NET, plus a couple of other languages. This gives you the option of writing scripts

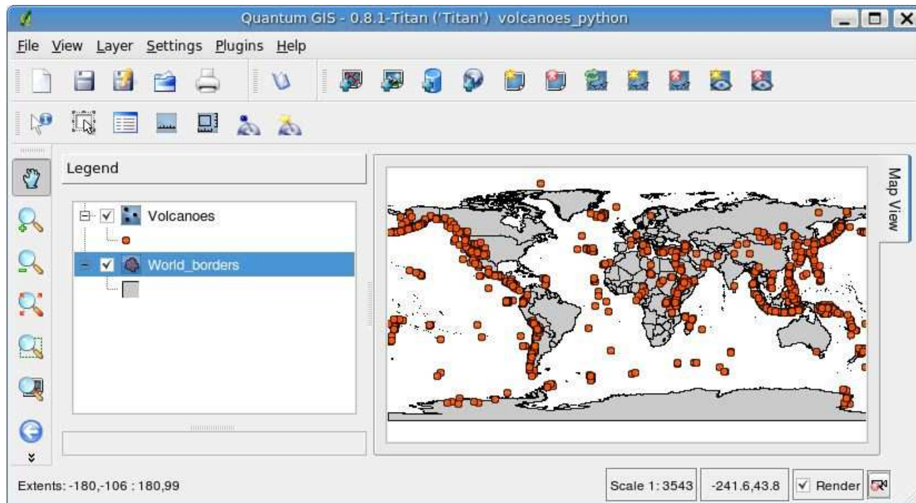


Figure 13.4: Volcanoes shapefile created with Python script

in these languages directly against the GDAL/OGR API to work with both vector and raster formats.

13.4 PostGIS

Although PostGIS is really on the server side of things, it's an important component of many of our desktop applications. In this section, we will look at using Ruby to access and work with geometries stored in PostgreSQL. Fortunately, the components to unite Ruby, PostgreSQL, and PostGIS already exist. We're using Ruby, but you could just as easily use Python or another language that provides PostgreSQL bindings.

Installing the Gems

First we need to get a few things installed in order to work with the database. Here's what you need:

- Ruby
- RubyGems
- postgres-pr
- GeoRuby

Obviously we need Ruby—I'll let you figure out how to get that—or just visit the website at http://www.ruby-lang.org. To access PostgreSQL, we

will use the postgres-pr library, and for working with geometries, we will use GeoRuby. We can install these using gem. To install RubyGems, download the distribution file for your platform, and unpack it. Make sure you have rdoc installed first. Then install Rubygems by changing to the distribution directory and running the setup.rb file:

```
$ ruby setup.rb
```

Install the remaining dependencies using gem:

```
# gem install postgres-pr
Bulk updating Gem source index for: http://gems.rubyforge.org
Successfully installed postgres-pr-0.4.0
# gem install GeoRuby
Successfully installed GeoRuby-1.2.1
Installing ri documentation for GeoRuby-1.2.1...
Installing RDoc documentation for GeoRuby-1.2.1...
```

Now we have the tools installed and are ready to give it a go. We already have a PostgreSQL database with some layers loaded. First let's just run a simple script to check the connectivity and make sure everything is working correctly. We can test this with the following code:

```
require 'rubygems'
require 'postgres-pr/connection'
c = PostgresPR::Connection.new('gis_data', 'gsherman', '', 'tcp://madison:5432')
res =c.query('select * from geometry_columns')
res.rows.each{|r|
  puts r
  puts "-------------"
}
```

The first couple of records from the script are shown next, just to prove it worked. We have connectivity to our database and were able to print out all the records in the geometry_columns table.

```
public
country
shape
2
4326
MULTIPOLYGON
-------------

public
edit_test
shape
2
4326
POINT
-------------
```

Now let's quickly test the GeoRuby install using the following script to create a point and print it out in WKT format:

```
require 'rubygems'
require 'geo_ruby'
include GeoRuby::SimpleFeatures
p = Point.from_x_y(-151.25,61.75)
puts p.as_wkt
```

The output from the script is a whopping:

```
POINT(-151.25 61.75)
```

Although not very earth-shattering, it proves we have the environment set up right and now can move on to doing something useful with our PostgreSQL data and Ruby.

Remember in Section 8.2, *Importing Data*, on page 122, where we wrote a little script to prepare the historic earthquake data for import? Well, we're going to modify that a bit and use it to load the earthquake data directly into PostGIS. Here's the script with the new bits added in:

Download load_earthquakes.rb

```
Line 1  #!/usr/local/bin/ruby
     -  require 'rubygems'
     -  require 'postgres-pr/connection'
     -  require 'geo_ruby'
     5  include GeoRuby::SimpleFeatures
     -  # load earthquake data into PostGIS
     -  conn = PostgresPR::Connection.new('gis_data',
     -                                     'gsherman',
     -                                     '',
    10                                     'tcp://madison:5432')
     -  # create the database table
     -  conn.query('create table quake_demo (id int4 primary key, event_date date,' +
     -    'event_time varchar(10), latitude float, longitude float, depth float,' +
     -    'magnitude float, geom geometry)')
    15  # Open the file
     -  f = File.open("db_search2291")
     -  # Skip the first two header records
     -  2.times{f.gets}
     -  # Member for the primary key
    20  key = 0
     -  # process the earthquake records
     -  while not f.eof
     -    record = f.gets
     -    # used a fixed length approach to get the fields we want since
    25    # splitting on white space isn't feasible
     -    event_date = record[1..10]
     -    event_time = record[13..22]
     -    latitude = record[26..32]
```

```
-     longitude = record[37..44]
30    longitude_direction = record[46..46]
-     depth = record[50..54]
-     magnitude = record[66..69]
-     # if the longitude is in the western hemisphere, it must be
-     # negative
35    longitude = -1 * longitude.to_f if longitude_direction == 'W'
-     magnitude = '0' if magnitude.strip.length == 0
-     key = key + 1
-     # create a point using geo_ruby
-     pt = Point.from_x_y(longitude, latitude)
40    # insert the row
-     result = conn.query("insert into quake_demo values(#{key}, " +
-       "'#{event_date}', '#{event_time}',#{latitude}, #{longitude}, " +
-       "#{depth}, #{magnitude}, GeometryFromText('#{pt.as_wkt}',4326))")
-
45  end
-     # create the spatial index
-     result=conn.query("create index sidx_quake_demo on quake_demo " +
-       "using GIST (geom GIST_GEOMETRY_OPS)")
-
50    # analyze the table
-     result = conn.query("vacuum analyze quake_demo")
-
-   conn.close
-   f.close
```

Let's take a look at how this works. First we need to pull in the needed dependencies; then in line 7, we make connection to the database. If you want to try this, just change the parameters to match your setup (database name, username, password, and connection information).

Once we have a connection, the next step is to create the table. We do this with a simple DDL[10] query beginning with line 12. The rest of the script is pretty much the same as the one we used to prepare the delimited data, until you get down to line 39 where we use one of the GeoRuby classes to simplify things a bit. By using the Point object, we can easily get the WKT needed for our insert statement. Sure, we could have cobbled it together from the latitude and longitude, but that wouldn't be any fun. Once we have the point, we create the insert statement (41). As you can see from the code, this is done for each line in the input file, resulting in more than 12,000 records in the database.

Once all the records are loaded, we have a couple of things left to do. First we need a spatial index on our new table. Without it, retrieving

10. Data Definition Language

features for a specific area will be slow. Again, we use a bit of DDL in line 47, taken straight from the PostGIS manual, to create the index.

The last step is to get out the vacuum and sweep up the dust. Well, not really, but we do need to run VACUUM ANALYZE on our new table. In the PostgreSQL world, this command reclaims space for use and updates the statistics for a table or tables to ensure that query execution is planned efficiently. Even if you don't quite understand that explanation, just run it anyway—your data will be happy, and you will be happy.

Our new table is now ready to use. We can view it by any application that supports PostGIS layers, assuming it doesn't rely on our old friend the geometry_columns table. Take a look back at Section 7.3, *The geometry_columns Table*, on page 104 if you need a refresher. QGIS doesn't require an entry in the geometry_columns—it can scan the database and find all tables that have a spatial column. If your client (or server) software relies on the geometry_columns table, you can easily add a record using something similar to the following:

```
insert into geometry_columns
  values('', 'public', 'quakes_demo', 'geom', 2, 4326, 'POINT');
```

Now every application that supports PostGIS should be able to find the layer. In Figure 13.5, on the following page, you can see the results of our scripting work, loaded into QGIS.

The earthquakes are rendered in graduated symbols, with the worst being big red dots. I also added the NASA JPL world mosaic as a backdrop. The big red dot you see is the site of the March 1964 Alaska earthquake, which registered approximately 9.2 on the Richter scale.

You may be wondering why we added an integer primary key when we created the table in line 12. Well, first, it's always good to have a primary key in your tables. And second, if we want to use the table in QGIS, it requires a primary key in order to manage selection sets and other bookkeeping tasks behind the scenes.

Transforming Coordinates

For our last example, we take a look at a simple way to transform coordinates using a PostGIS-enabled database. Why would you want to do this? Well, let's look at the process first and then revisit that question.

Harrison has some old musty topographic maps he used to mark bird sightings back in the old days before he had a GPS. Now he wants to

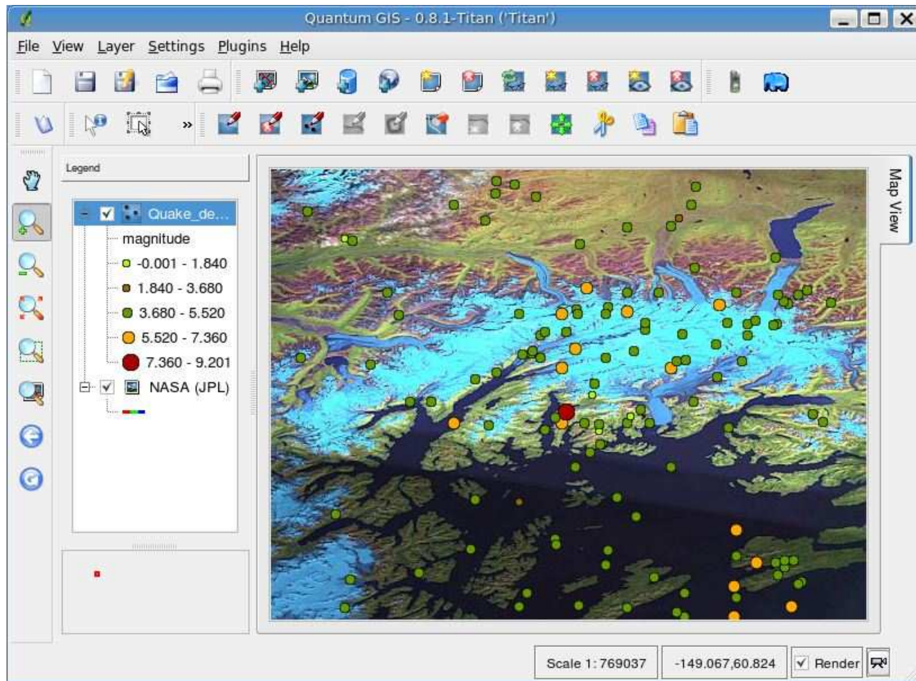


Figure 13.5: Results of loading earthquakes into PostGIS

add them back into his GPS and integrate them into his bird database (Harrison likes to revisit the exact spots year after year). No problem, you say—just put the latitude and longitudes from the paper map into the GPS. But Harrison, being an up-and-coming GIS savant, realizes that his paper maps are in a different datum than his GPS uses. The old maps were created using the NAD27 datum, where his GPS now uses WGS84. If he just plugs the latitudes and longitudes into the GPS, the locations could be off by as much as 400 feet or more. Since Harrison has a really accurate tool for determining latitude and longitude on his paper maps, he's not happy with a datum problem.

It turns out it's a simple matter to transform coordinates using PostGIS and GeoRuby. You don't even have to store anything in the database.

Here's the code:

Download transform_point.rb

```
Line 1  #!/usr/local/bin/ruby
     -  require 'rubygems'
     -  require 'postgres-pr/connection'
     -  require 'geo_ruby'
     5  include GeoRuby::SimpleFeatures
     -  # Connect to the database
     -  conn = PostgresPR::Connection.new('gis_data',
     -                                    'gsherman',
     -                                    '',
    10                                    'tcp://madison:5432')
     -  # Create a point using geo_ruby from the command line arguments
     -  pt = Point.from_x_y(ARGV[0], ARGV[1])
     -  # transform the point
     -  wkt = "'POINT(#{pt.x} #{pt.y})'"
    15  sql = "select transform(GeometryFromText(#{wkt},4267), 4326)"
     -  result = conn.query(sql)
     -  # Create a WGS84 point from the transform result
     -  wgs84_pt = Point.from_hex_ewkb(result.rows[0][0])
     -  # print the results
    20  print "Input NAD27 point : #{ARGV[0]}, #{ARGV[1]}\n"
     -  print "Output WGS84 point: #{wgs84_pt.x}, #{wgs84_pt.y}\n"
     -  # Close the database connection
     -  conn.close
```

Let's take a look at a couple of things about the code. First, Harrison must specify the latitude and longitude on the command line, since the NAD27 point from his map is created on line 12 using the first two elements of the ARGV array. Since I like to specify coordinates as X and then Y, that's the way Harrison did it. This means we have to put the longitude on the command line first, followed by the latitude. The second thing to note is there is absolutely no error checking—this is a no-frills script.

When Harrison runs the script, here's what he gets:

```
$ ruby code/transform_point.rb -151 61
Input NAD27 point : -151, 61
Output WGS84 point: -151.002235480186, 60.9994441802801
```

He now has his first point in converted to WGS84. He could just as easily transform the points to a projected coordinate system by substituting the appropriate SRID in the query. For example, if we wanted to convert from our NAD27 latitude and longitude coordinates to UTM Zone 6, NAD83 datum, the query on line 15 would look like this:

```
select transform(GeometryFromText(#{wkt},4267), 26906)
```

A Better Way to Transform?

You may be wondering if there is a better way to transform coordinates. In fact, there are a bunch of ways to do it. For simple command-line transformations, you can use the cs2cs utility that is part of the PROJ4 Cartographic Projections Library. This library is used by virtually all OSGIS projects for doing transformations or "projecting on the fly."

You can also transform entire datasets using ogr2ogr. Many of the OSGIS applications provide a means to transform a dataset to new coordinate system.

So, why use Ruby and PostGIS to do it? The point was to gently introduce you to the possibilities of scripting using both PostgreSQL and the GeoRuby classes. Let's hope we succeeded in whetting your appetite for further scripting endeavors.

We just need to know the proper EPSG code to specify in the query, in this case 26906. If we run the program now, we get different-looking coordinates—let's hope ones that make sense for a point in UTM Zone 6, NAD83:

```
$ ruby code/transform_point.rb -151 61
Input NAD27 point : -151, 61
Output UTM Zone 6 NAD83 point: 283625.288353973, 6769338.97358209
```

One other point before we leave this topic: note how we created a point from the results of a query. We used the from_hex_ewkb(WKB) method when creating the new point. This is the format that is returned when we query a geometry column in a PostGIS layer. This handy method makes it easy to create objects from the results of a query.

The GeoRuby classes can also be used with Ruby on Rails, opening up a world of possibilities for integrating PostGIS and your web applications.

Finally, if you are thinking this example was a bit contrived, you are right. However, there is a practical application. For example, suppose you are writing an application (either web or desktop) that accepts user input in one coordinate system but must store it in PostGIS in another. The simple solution presented here is one way to do it. Of course, the possibilities are endless. . . .

Here be dragons!
► Mapmakers of old

Chapter 14
Writing Your Own GIS Applications

Most GIS users have ventured into the realm of programming—whether it be writing scripts or full-blown applications. Scripting is quite useful for automating GIS tasks, as we saw in Chapter 13, *GIS Scripting*, on page 235.

Sometimes you find yourself in a position where you need a customized application. The full version of your favorite OSGIS application is perhaps overkill or doesn't provide the features you need. Often trying to twist the application into the form you need results in a system that is not user-friendly and is difficult to use. Many disciplines can benefit from a lightweight custom application that serves a specific need. These are the reasons for writing such an application; let's look at some of the specifics.

14.1 Options for Writing Your Application

If we are going to write an application, there are some things we *don't* want to build from scratch:

- Low-level drawing routines for displaying raster and vector data
- Read/write access to our data stores
- Renderers such as unique value and graduated symbol
- Legend generation

We want the API to handle all the hard stuff for us so we can concentrate on the custom functionality. That said, let's see what APIs are available for the task.

Tools for Building a Custom Application

You can build a custom application in many ways. You can program at a low-level against, for example, the GDAL/OGR or GRASS libraries. One of the most time-consuming aspects of creating an application is the user interface. If you've never done it before, you'll find that it takes nearly as much code for the GUI as it does for the logic. If you are going to write a desktop application, the first thing you should decide is which GUI toolkit you will use.

Let's take a look at the GIS toolkits (APIs) available for us to work with. If you're like me, cross-platform support is an important consideration, although depending on the scope and target for the application, building on a single OS may be perfectly acceptable. In the list of toolkits that follow, we'll point out the level of cross-platform support for each. This is not a comprehensive list—there are likely other toolkits out there in the wild.

Mapnik
: Mapnik[1] is a toolkit for developing mapping applications using C++ or Python. As of version 0.4, Mapnik runs on Linux, Mac OS X, and Windows. Mapnik renders its output as an image. You can likely integrate this with whatever GUI toolkit you desire, based on your platform.

MapWinGIS
: MapWinGIS[2] is an ActiveX control that you can use with any programming language that supports ActiveX on Windows. This includes Visual Basic, Delphi, VB .NET, and C# and the GUI elements that go along with them.

PyWPS
: PyWPS[3] is the "Python Web Processing Service," an implementation of the Web Processing Standard from the Open Geospatial Consortium. PyWPS allows you to write applications using Python and GRASS that work over the Web. Although it isn't a way to

1. http://mapnik.org
2. http://www.mapwindow.com
3. http://pywps.wald.intevation.org

write desktop applications, it does use GRASS on the back end to provide powerful geoprocessing capabilities via the Web and is an option depending on your needs.

QGIS libraries
: The QGIS[4] libraries support stand-alone application development using both C++ and Python. Operating system support includes Linux, *BSD, Mac OS X, and Windows. If you write an application using the QGIS libraries, you'll also be using the Qt GUI toolkit.

uDig framework
: The uDig[5] framework support cross-platform customization using Java and the Eclipse Rich Client Platform (RCP). The framework can be extended through the use of plugins and GUI customization. Since uDig is based on Eclipse, you'll use the SWT toolkit when developing your own customizations.

Now for a caveat when choosing a toolkit: Make sure it can support the data stores you want to use. uDig has good format support as does QGIS, since it relies on GDAL/OGR for reading and writing spatial data. Mapnik support shapefiles, PostGIS, and TIFFs. MapWinGIS supports shapefiles and ASCII grids and rasters in the form of GIF, TIFF, JPEG, and BMP.

14.2 Examples of Custom Applications

Here we'll provide a couple examples of applications that have been written using some of the toolkits we mentioned previously. This serves to illustrate the kinds of things people are doing with custom applications and serves to show that when you develop it, you are in control of the function and appearance.

Quantum Navigator

Quantum Navigator is an example of a complete application developed with the QGIS libraries. The application is written in Python using the bindings available for the QGIS libraries. The user interface is designed in Qt and made possible through the PyQt bindings. The components used in the construction of this application are as follows:

- QGIS libraries (version 0.9 or later)

4. http://qgis.org
5. http://udig.refractions.net

- Python
- Qt toolkit
- PyQt (Python bindings for the Qt toolkit)

Now you know what it's made of, let's see what it does. Quantum Navigator provides basic routing and navigation capabilities on a road map. With the proper road data (a shapefile), you can select the start and end points of your route and the application will calculate a route based on your criteria, whether it be the shortest, fastest, or most economic path. The route obeys all rules of the road, such as one-way streets and turn restrictions.

Quantum Navigator additionally includes a GPS simulator that can be programmed by creating waypoints on the map by clicking at the appropriate places and specifying a speed value. When in navigation mode, the application provides information about upcoming turns as you approach them.

In Figure 14.1, on the following page, you can see Quantum Navigator with an economic route outlined in blue with waypoints visible along the way. In case you're wondering, this application was not developed to provide actual in-car navigation with a GPS. It was done as part of a thesis and, per the author,[6] serves as a good example of creating a custom application with the Quantum GIS API and Python.

OpenOceanMap

Faced with deploying a means to collect fisheries data in the field, Ecotrust[7] developed OpenOceanMap, a decision support tool based on open source software. Rather than deploy a fleet of commercial software, Ecotrust developed the custom application using QGIS, Qt, and Python. This low-cost approach provided a lean interface that simplifies data collection in the field. It also avoids the complication and confusion that can result when you embed your functionality inside a full-blown desktop GIS. This stripped-down approach provides a workable cross-platform solution that can be efficiently deployed in the field. In Figure 14.2, on page 268, you can see the OpenOceanMap application.

6. Martin Dobias, http://mapserver.sk/~wonder/qnavigator/
7. http://ecotrust.org

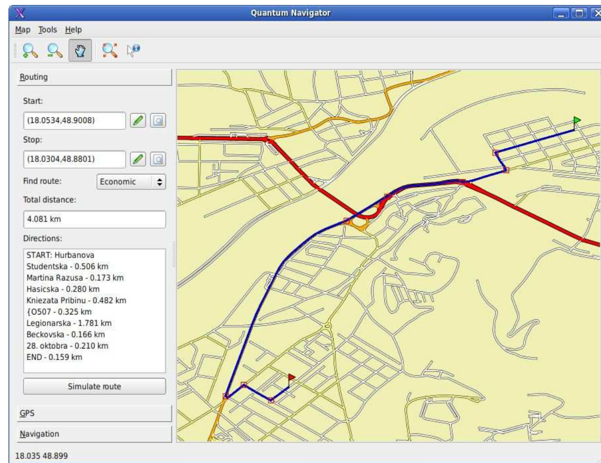


Figure 14.1: Quantum Navigator

uDig Examples

uDig has been used in a number of projects that created a custom application using the framework. Here are some examples:

- Populations at Risk, a disaster planning and response tool
- GeoVista, a system for management of parks and reserves
- Diva GIS, a tool used by UN Food and Agriculture to map and analyze potato species distributions
- ArboGIS, a tool for forest resource data management

14.3 How to Approach Your Own Project

We've seen some examples of custom applications using a couple of different technologies. You may be wondering at this point what the best approach is when starting your own project. Obviously you have choices to make: programming language, GIS toolkit, and GUI toolkit.

You may have noticed that the toolkits used in our examples (QGIS and uDig) are tightly coupled with both the programming language and the GUI toolkit. In other words, the GIS toolkit you choose is going to dictate that other two components. Regardless of the approach you

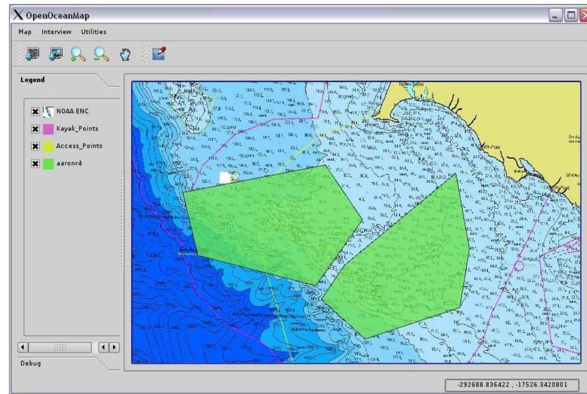


Figure 14.2: OpenOceanMap

take, the key thing is to leverage all the hard work others have done before you, so you can concentrate on implementing your logic, rather than redesigning the wheel from scratch.

A man's feet should be planted in his country, but his eyes should survey the world.

► George Santayana

Appendix A

Survey of Desktop Mapping Software

There are a lot of applications in the OSGIS desktop world. In this chapter, we'll explore some of the major choices available. In our survey, we'll classify applications based on both capability and the underlying language. The programming language behind an application is important because it affects how the application is distributed, is installed, and how easy it is for us to customize.

Sometimes we open source enthusiasts are a bit odd. Many of us choose our software based on the language in which it is written. When you begin to look into OSGIS applications, you will find them divided into what have been termed *tribes* based on the programming language. While we mention the programming language, our focus will center around the features of each application. For reference, in Figure A.1, on the following page, you can see the language behind each of the applications in our survey, including the potential for using a scripting language with each. If you are a programmer, you'll be interested in the underlying language since it will give you an idea of how easily you might customize or extend the application. After all, that's a big part of what open source is all about.

Bindings consist of interface code (stored in a shared library or DLL) that allow you to access the features of an application or library from a scripting language.

For most people, the words *Desktop GIS* generally conjure up visions of a GUI interface. Although that's largely true, it's clear there are command-line applications that deserve a place in our toolkit. In the survey, we'll divide the applications into two primary groups—those with a GUI and those that are command line only.

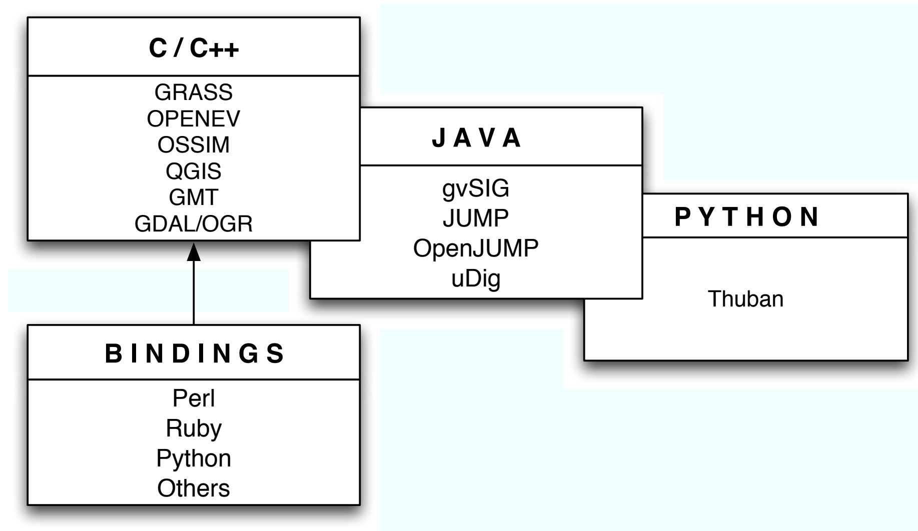


Figure A.1: Applications grouped by underlying programming language

A.1 GUI Applications

Let's start with the GUI applications. A lot of OSGIS GUI applications exist. If you don't believe me, just take a look at the FreeGIS website.[1] We're going to hit some of the major ones. Our survey includes the following apps in alphabetical order:

- GRASS
- gvSIG
- Jump/OpenJump
- OSSIM
- OpenEV
- Quantum GIS
- Thuban
- uDig

GRASS

GRASS is written in C with the GUIs implemented using TCL/Tk and Python.

Let's start with the patriarch of the OSGIS world—GRASS. The Geographic Resources Analysis Support System, or GRASS as its commonly called, is a GIS that supports analysis, modeling, visualization, raster processing, and many other operations. It is the "heavyweight" of the OSGIS world.

GRASS was originally developed by the U.S. Army Corps of Engineers Construction Engineering Research Laboratories (USA-CERL) for use in environmental research, assessments, monitoring, and management of U.S. Department of Defense lands. The last release was in 1992 and by 1996, USA-CERL was no longer supporting the public in its use of GRASS. This began a transition period that in the long run gave us the open source version of GRASS we have today.

Today GRASS has an international team of developers and users throughout the world, including academia, government, and consulting companies. If you are interested in more of the history, visit the GRASS home page.[2]

In Figure A.2, on the next page, you can see a simple example of GRASS with the countries of the world layer displayed on a raster background.

GUI or Not?

Is GRASS a GUI program or a command-line program? The answer is both. Seriously, though, GRASS has a GUI component, but the real work is done by a suite of command-line programs, or *modules*, that do everything from import data to combining grids. The GUI side provides both a means to view your data and to perform the many functions that GRASS provides. So in reality, you can think of GRASS as a bit of a hybrid with the power of the command line and the convenience of a GUI. The individual programs can be glued together with a scripting language (shell, Perl, Python, Ruby, your choice) to perform complex operations.

The GUI is currently undergoing a bit of change with a new interface being developed using Python and wxWidgets.[3] In addition, Quantum GIS supports viewing of GRASS layers, giving you another option for visualizing your data.

1. http://freegis.org
2. http://grass.itc.it
3. http://www.wxwidgets.org

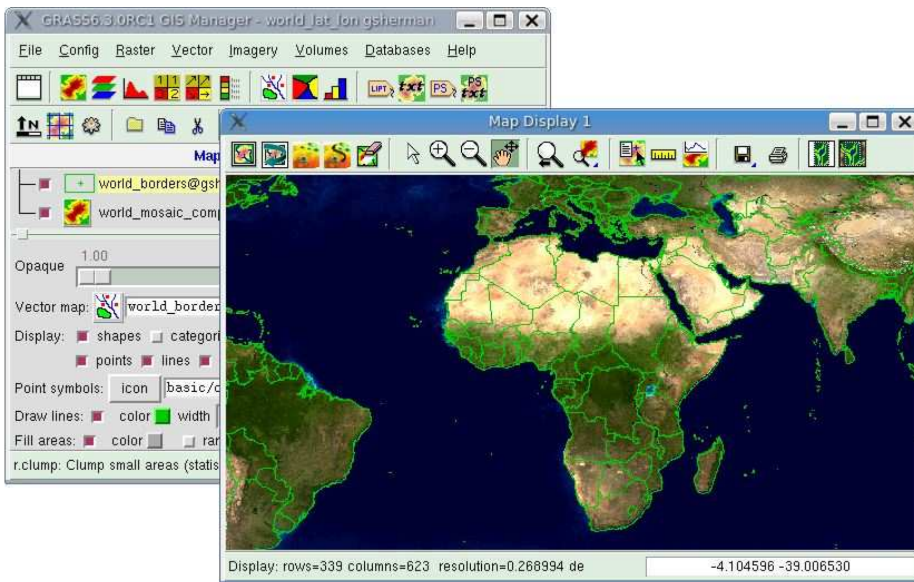


Figure A.2: GRASS on Linux

Pros and Cons

Let's look at the pros and cons of using GRASS. First here are the pros:

- Mature and stable implementation
- Huge feature set for visualization and analysis of both raster and vector data
- Supports wide array of data formats
- Good vector digitizing tools
- Good community support from mailing lists, Wiki, and IRC
- 3D visualization
- Can be automated and scripted using common languages
- Choice of GUIs
- Good documentation
- Packages available for most supported platforms
- Raster map algebra and simulation models

And now here are the cons:

- Rather steep learning curve
- For analysis, data must be converted to GRASS format
- "Nonstandard" GUI

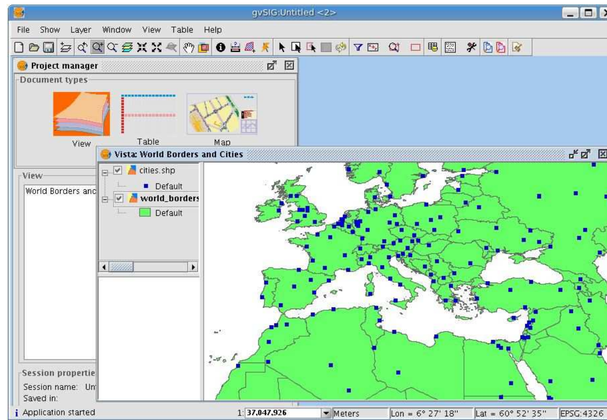


Figure A.3: world_borders and cities layers in gvSig

What kind of user would want to use GRASS? Although it's definitely not for Clive, our casual user (he should use the QGIS-GRASS integration), it's a good choice for our advanced user Alyssa. Intermediate users will find parts of it that may be useful and worth a test drive. Only you can tell whether it's for you.

gvSIG

gvSIG is written in Java.

gvSIG is an open source project that allows you to work with a variety of vector and raster data formats, including shapefiles, GeoTIFF, ECW, JPEG, WMS, WFS, and WCS. gvSIG provides a set of editing tools for maintaining your data. gvSIG is multiplatform, running on Windows, Linux, and Mac OS X. Plugins can be used to extend the functionality and provide access to additional data formats. In Figure A.3, you can see gvSIG with the world_borders and cities layers.

Pros and Cons

The pros for gvSIG include the following:

- Good format support including web-deliverable data
- Extensible through plugins
- Editing and drawing tools
- Map layouts
- Geoprocessing tools (buffer, intersection, union, and so on)

For gvSIG there are also a few cons:

- Incomplete internationalization of the text in menus and dialog boxes (gvSIG's first language is Spanish)
- Still maturing, especially regarding additional language support
- Minor GUI issues on some platforms[4]

JUMP and OpenJUMP

JUMP and OpenJUMP are written in Java.

JUMP stands for the Java Unified Mapping Platform. OpenJUMP is based on JUMP, created by Vivid Solutions,[5] with code contributions from Refractions Research.[6]

As development slowed, a group of JUMP users decided to continue development under the name OpenJUMP. This group created a "fork" of the JUMP source code and continued development apart from the JUMP development team. As a result, the two programs are similar, but different and incompatibilities have been introduced. For example, you can't use all OpenJUMP plugins in JUMP, and vice versa. In addition, there are no less than four other derivatives based on the original JUMP work at various stages in its development. These include DeeJUMP, SkyJUMP, PirolJUMP, and KOSMO.

The latest version of JUMP (1.2) released in November 2006 added support for rasters, spatial databases (PostGIS), and enhanced query capability. Most of these features were subsequently ported to OpenJUMP. While development on JUMP seems to have come to a standstill, the development of OpenJUMP continues with volunteer efforts.

As far as the fork goes, generally speaking it's considered undesirable for a project in the open source world. The end result in this case is two or more projects with similar goals and a common root, all under active development. Which you might use is up to you, based on the features and stability you desire. When it comes to the JUMP family, you have six choices, so you'll have a bit of homework to do if you decide the JUMP lineage is for you.

While not as strong on the analysis end of the spectrum as GRASS, it does have a lot of useful features for all classes of users. Its support of GIS standards and good feature set make it attractive to a lot of folks.

---

4. The issues were noted during a review of the 1.1 version and are being addressed by the development team.
5. http://www.vividsolutions.com
6. http://refractions.net

Pros and Cons

Let's look at the pros and cons of JUMP. First here are the pros:

- Easy install
- Editing tools
- Support for a good range of data formats
- WMS support
- Basic analysis functions like buffering and spatial operations
- Geometry validation: checks to see whether your features are valid
- Good set of visualization options
- Extensible framework for creating customizations
- Supports industry standards

And here are the cons:

- Six versions from which to choose from with differing levels of support, goals, and features

To give you a sample of the interface, OpenJUMP with some worldwide data loaded is shown in Figure A.4, on the next page.

OSSIM

OSSIM is written in C++ and uses the Qt class library.

OSSIM stands for Open Source Software Image Map and is pronounced as "awesome." The core of OSSIM is a library that provides remote sensing, image processing, and geospatial functionality. Some of the things you can do with OSSIM include the following:

- Ortho rectification[7]
- Terrain correction
- Create mosaics from individual images

OSSIM supports a wide range of projections as well as a lot of data formats. As you can guess from the name, OSSIM is focused on raster processing and display rather than vector data.

You might be wondering why are we talking about a software library in the survey of desktop applications. The reason is because OSSIM also comes with a selection of command-line utilities, as well as three GUI applications (ImageLinker, iview, and ossimPlanet).

7. *Ortho rectification* is a process where an image (photo) is registered to map coordinates (a projection) and correction is made for distortions because of terrain. The result is an image that can be used in your GIS software that will "line up" with your other data.

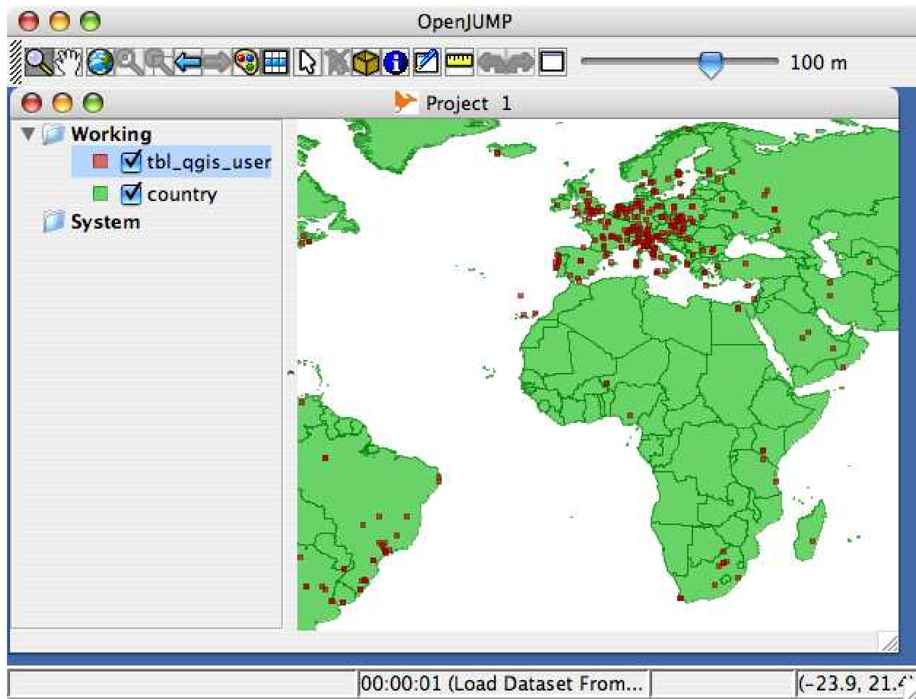


Figure A.4: OpenJUMP

ossimPlanet is a globe-style viewer that provides support for a number of formats. There are a number of reasons why you might want to use it instead of, or in addition to, Google Earth, including access to Worldwind[8] data, the ability to add your own data without having to convert it first, and its support for WMS layers. Plus, it's open source, and you can drag and drop your data right onto the globe.

In Figure A.5, on the following page, you can see ossimPlanet zoomed into a local airport. Since ossimPlanet can access Worldwind layers, you get the full access to the data on platforms where Worldwind doesn't run (everything but Windows).

Whether the OSSIM suite of programs is right for you depends on how much you play with raster data. In general, the new ossimPlanet is a good viewer of raster imagery available from Worldwind sources and

8. http://worldwind.arc.nasa.gov/index.html

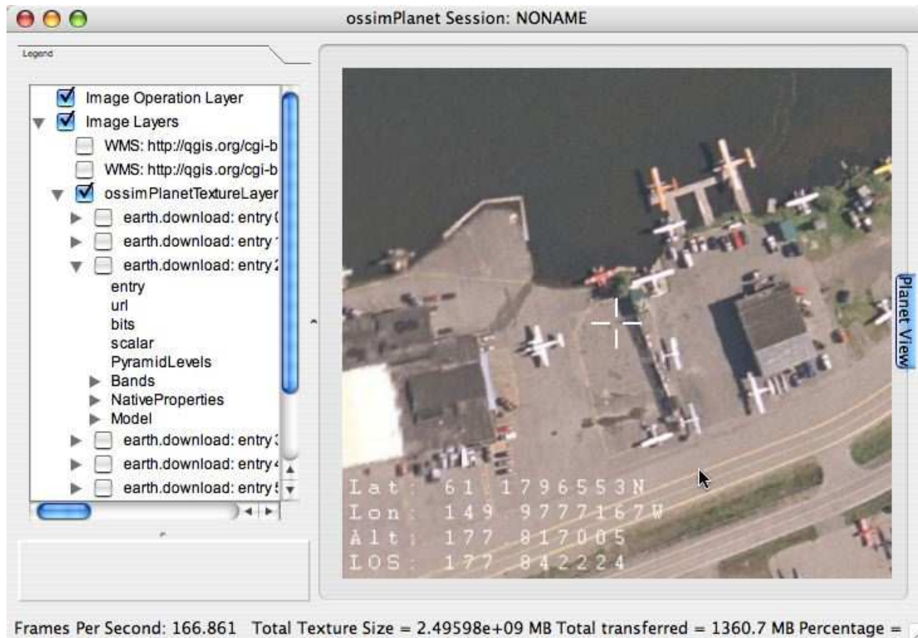


Figure A.5: ossimPlanet viewing hi-res imagery

other WMS servers around the Internet, including NASA JPL. At this point I'd say the raster processing capabilities are for the advanced user. But casual and intermediate users might find it a great tool for viewing raster data from a number of sources across the Internet. The fact that you can drag and drop your own shapefiles right into ossim-Planet is a great advantage too. Just keep in mind that your shapefiles have to be in geographic (read latitude/longitude) coordinates in order to display them in ossimPlanet.

Pros and Cons

The pros for the OSSIM suite are as follows:

- Impressive raster processing capabilities
- Access to a huge repository of data on the Internet through using ossimPlanet
- Runs on most platforms
- Community support via website, mailing list, and IRC

There are just a few cons:

- Still evolving (especially with regard to ossimPlanet)
- Has a bit of a learning curve
- Advanced raster processing functions not well documented

OpenEV

OpenEv is written in C and makes extensive use of Python.

OpenEv allows you to view vector, raster, and WMS data sources. OpenEv has been around for a while (since 2000), and recently development on it seems to have slowed. The latest version (1.8) was released in 2004, and the project is currently in maintenance mode.[9] There is an effort underway to produce version 2 of OpenEv, using the Gtk2 toolkit to give it a more modern look and feel. Currently, version 2 is still under wraps and can't be tested. If you are interested in OpenEV 1.*x*, it is distributed as part of the excellent FWtools suite.[10] Note that OpenEv is available only for Linux and Windows.

Should you consider using OpenEv? Frankly, there are probably better choices for you to use in visualizing your data. While there is nothing wrong with OpenEv, the facts that it's in maintenance mode and that version 2 hasn't surfaced yet makes it less attractive.

Quantum GIS

Quantum GIS is written in C++ and uses the Qt class library.

I'll try to maintain some objectivity is this section.[11] The Quantum GIS project was founded in early 2002 with the original goal of building a GIS data viewer for Linux that was fast and supported a wide range of data stores, in particular the PostGIS spatial database. Since then, QGIS, as it's known, has grown to support a large array of data types and runs on many platforms, including Mac OS X, Windows, BSD, and of course Linux. The project has a strong developer community, and a glimpse at the user map (see Figure 11.4, on page 185) shows that it is being used widely around the world.

In Figure A.6, on the following page, you can see a simple example of QGIS with the countries of the world layer displayed on the map canvas.

QGIS provides viewing for both vector and raster data sets. Support for the majority of these is provided through the GDAL/OGR libraries (see Section 11.2, *Using GDAL and OGR*, on page 186).

---

9. See http://openev.sourceforge.net for details
10. See http://fwtools.maptools.org.
11. I founded the Quantum GIS project.

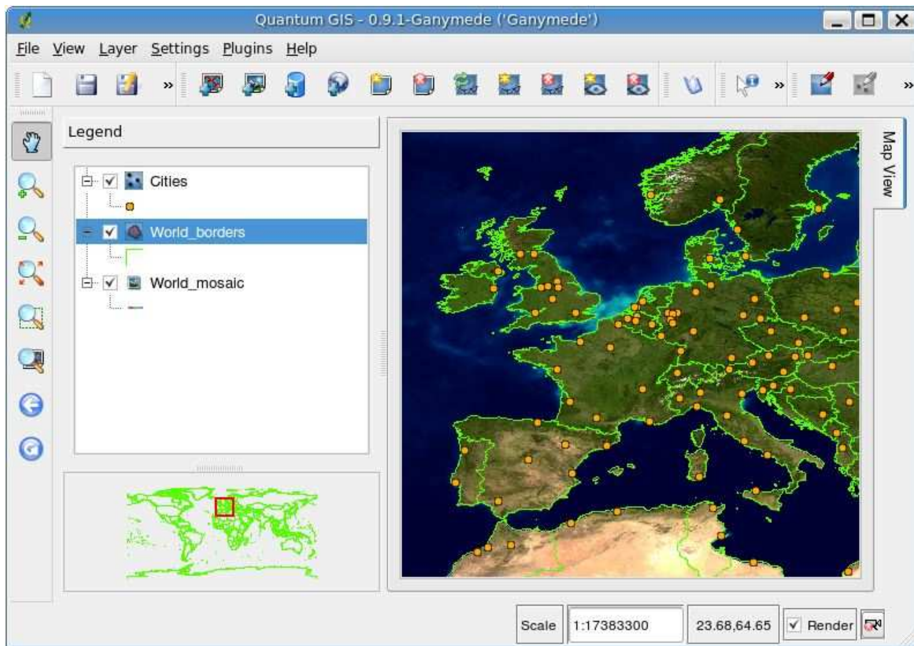


Figure A.6: Quantum GIS

Additionally, it supports PostGIS layers (stored in a PostgreSQL database), delimited text, GPS tracks, routes, and waypoints, and GRASS layers. With QGIS you can edit PostGIS and GRASS layers and do heads-up digitizing.

QGIS and GRASS

QGIS is not a full-fledged GIS application (the only one I consider in that category throughout our survey is GRASS). To enhance its capabilities, QGIS supports viewing, editing, and manipulation of GRASS data through its plugin facility. This allows you to create data from the GRASS command line and view it in QGIS. The GRASS toolbox provided by the plugin allows you to perform many common functions without leaving QGIS.

QGIS is an application that has something for every class of user. Casual users will find it a handy tool for visualizing data and working with your GPS data. Irving and his cohorts (that's you intermediate users) can use QGIS to create and edit data in a number of formats.

For the advanced crowd (Alyssa and friends), you can perform analysis using the GRASS plugin.

Pros and Cons

Let's look at some of the pros of QGIS:

- Support for a wide range of both vector and raster data
- Editing capabilities
- Good set of tools for symbolizing and visualizing your data
- Good documentation
- Strong community support through forum, mailing lists, and IRC
- Extensible through plugins
- Includes a plugin for working with GPS units
- Good integration with GRASS visualization, editing, and analysis functions
- Customization using Python in order to write new tools and plug-ins (requires version 0.9.0 or higher)

Here are a couple of cons:

- Simple feature labeling (no collision detection)
- Limited map composition and printing capability
- Doesn't include all features of a desktop GIS
- Still maturing

QGIS likely has a place in your visualization and editing toolbox, especially when you consider the wide range of formats supported, the integration with GRASS, as well as the extensibility provided by the Python bindings.

Thuban

Thuban is written in Python and uses the wxWidgets toolkit.

Thuban has been around since 2002 and is developed and maintained by Intevation GMBH. Thuban provides viewing of GIS data stored in shapefiles, GeoTIFF, and PostGIS. It has projection support and can do table queries and joins. Figure A.7, on the following page, which comes from the Thuban website,[12] shows the main application window with several vector layers loaded.

12. http://thuban.intevation.org/

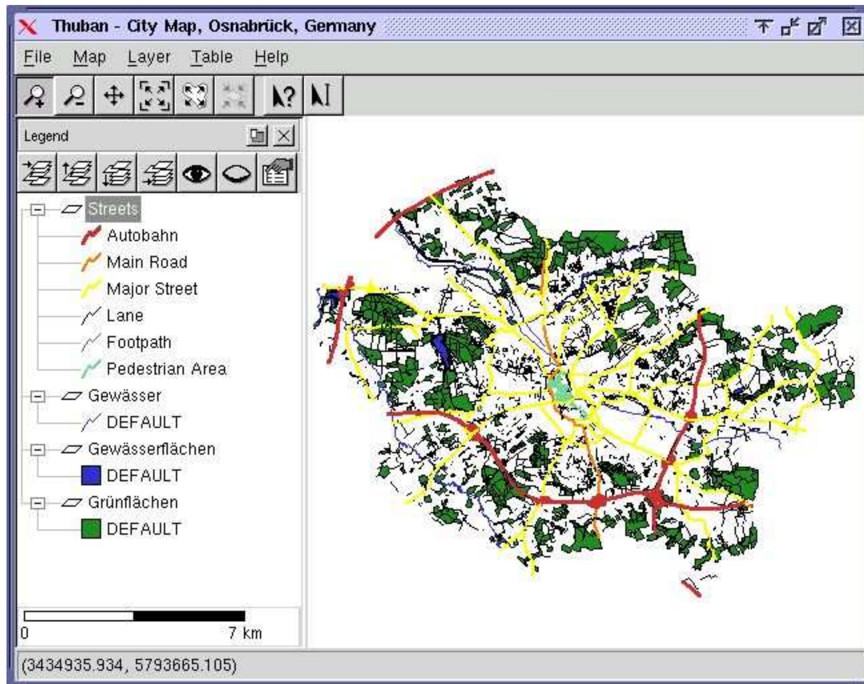


Figure A.7: Thuban

Thuban is a nice application with a pretty complete feature set for visualizing data. Since it doesn't provide much in the way of analysis capability, it's not going to fulfill all the needs of an advanced user.

Pros and Cons

Some of the pros for Thuban are as follows:

- Lightweight viewer
- Good feature set
- Support for a good range of data formats

For Thuban we do not find much in the way of cons other than the following:

- Install can be bit tricky
- Development seems to have slowed

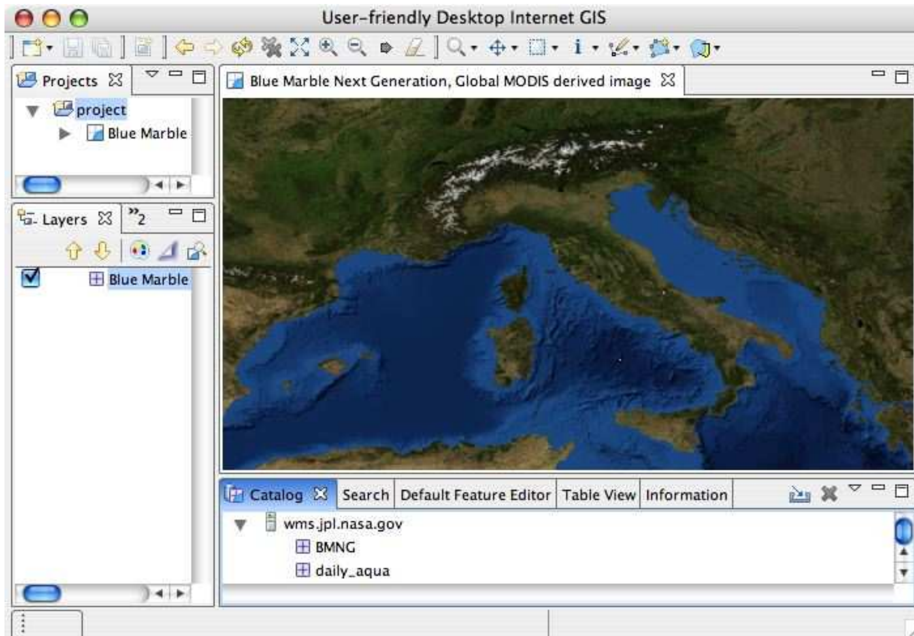


Figure A.8: uDig displaying NASA Blue Marble WMS

uDig

uDig is written in Java and uses the Eclipse framework.

uDig is the User Friendly Desktop Internet GIS, created by Refractions Research (http://udig.refractions.net). uDig provides viewing and editing for a variety of data formats, including the usual file-based layers (shapefiles and rasters), PostGIS layers, WMS, WFS, Oracle Spatial, and DB2. That should cover most of your data needs.

uDig provides a fairly complete set of viewing and editing tools. Since it supports WMS, we can pull in a wide array of free data from the Internet. In Figure A.8, you can see uDig displaying the NASA Blue Marble WMS layer, from the NASA JPL WMS site.[13]

uDig is another one of those Swiss Army knife–type applications that provides a lot of features. You may not use all of them, but there is something there for pretty much everybody. If you are heavy into analysis like Alyssa, you'll find it a bit light on that end of things. Otherwise, it's an easy-to-install and easy-to-use viewer/editor.

13. http://wms.jpl.nasa.gov/wms.cgi

Pros and Cons

Some of the pros for uDig are as follows:

- Support for a wide range of data formats
- Easy to install
- Both viewing and editing capabilities
- Extensible

And here are the cons:

- Interface is a bit nonstandard
- Some learning curve in getting up to speed on loading, symbolizing, and visualizing data
- No analysis functions; however, work is underway to integrate JGrass[14] into uDig

A.2 Command-Line Applications

Now it's time to take a look at some of the command-line applications you will find useful in your OSGIS ventures. With the advent of "modern" GIS software, most people want to point and click their way through life. That's good, but there is a tremendous amount of flexibility and power waiting for you with the command line. Many times you can do something on the command line in a fraction of the time you can do it with a GUI. The applications we'll look at next are definitely worthy of consideration when you start stuffing gadgets into your toolbox.

GDAL/OGR

GDAL and OGR are written in C and C++.

Let's take a look at GDAL and OGR. These two are used under the hood in a large number of GIS applications, both open source and proprietary. GDAL and OGR are really libraries that provide support for a vast number of raster and vector formats. Along with the libraries are a suite of command-line tools to work with these formats.

Raster support is provided by the GDAL library. Most popular raster formats are supported, including TIFF, PNG, JPEG2000, GRASS raster, ArcInfo grid, DEM, and ECW. Some of these formats require external libraries that are not included with GDAL (an example is ECW). If you want to have support for one of these, you will have to download and possibly build the dependent libraries and then compile GDAL. You

14. http://www.jgrass.org

can find a complete list of supported formats on the GDAL home page at http://www.gdal.org.

Vector support is provided by the OGR library. It too supports a large number of formats, including shapefiles, MapInfo (tab and mid/mif), PostGIS, GML, DGN, Oracle Spatial, and SQLite. The OGR library has similar constraints as GDAL. You may have to provide your own version of nonfree libraries needed to compile OGR (for example, Oracle Spatial).

GDAL-Supported Formats

As I said earlier, GDAL provides a number of command-line utilities for manipulating common raster formats. Operations such as warping, converting, and merging are all supported. You can also do coordinate transformation (that means changing projections) when converting between formats.

The list of supported formats in quite long. Some of the more popular raster formats supported by GDAL, and ones you are likely to encounter, are shown in the following list. We haven't listed them all—to get the full list, see the GDAL website.

- Arc/Info ASCII Grid
- Arc/Info Binary Grid (.adf)
- First Generation USGS DOQ (.doq)
- New Labeled USGS DOQ (.doq)
- ERMapper Compressed Wavelets (.ecw)
- ESRI .hdr Labelled
- ENVI .hdr Labelled Raster
- GMT Compatible netCDF
- GRASS Rasters
- TIFF/GeoTIFF (.tif)
- GXF—Grid eXchange File
- Hierarchical Data Format Release 4 (HDF4)
- Hierarchical Data Format Release 5 (HDF5)
- Erdas Imagine (.img)
- JPEG JFIF (.jpg)
- JPEG2000 (.jp2, .j2k)
- MrSID
- NetCDF
- Portable Network Graphics (.png)

- ArcSDE Raster
- USGS SDTS DEM (*CATD.DDF)
- USGS ASCII DEM (.dem)

The list of formats may seem a bit daunting—don't worry if you don't recognize all of them; the point is to illustrate the wide range available. At last count, GDAL included support for seventy-three different raster formats.

GDAL Utilities

Let's take a brief look at the command-line utilities that are part of GDAL. We won't cover them all, just the ones that you're likely to find useful from the outset. For a complete list and documentation, see the GDAL website.

gdalinfo
: This handy utility reports information about a raster, including, if applicable, the coordinate system, color palette, extents, and probably more than you want to know about your raster. This is a quick way to examine a raster and get some information on it without having to load it into a desktop application.

gdal_translate
: This command allows you to copy a raster file and convert it to another format. You can also add coordinate system information to the output or create a subset of the image by specifying a sub-window in pixel coordinates. Another neat trick is setting a certain value in the output to nodata,, making it transparent (depending of course on the software you use to view it).

gdaladdo
: This utility adds overviews (commonly called *pyramids*) to a raster to improve the display speed at smaller scales. On an image with pyramids, as you zoom in, more detail will appear. One note of caution, when using gdaladdo, your original image may be modified. For a GeoTIFF the pyramids are stored right in the original image. It's a good idea to create a backup before running gdaladdo.

gdalwarp
: With gdalwarp, you can "warp," in other words, transform, a raster from one coordinate system to another. This comes in handy if you want to convert a raster into a local projection to match your other

data. You can also specify multiple input files to create a mosaic from a group of rasters.

gdaltindex
: If you use MapServer, you'll find this utility handy for creating an index of the area covered by a group of rasters. Using gdaltindex you can create a tile index (a shapefile) that can be used with MapServer. You could also use the tile index as a coverage map to see where your rasters are located.

gdal_contour
: This utility will create contour lines from a Digital Elevation Model (DEM), which is essentially a raster composed of cells of a given size. Each cell has one attribute: an elevation. The smaller the cells, the more accurately the elevation is depicted. With gdal_contour, you specify the contour interval, and it kindly creates a shapefile with contour lines. There are other ways to do this, but gdal_contour is a quick way to make contour lines.

gdal_merge.py
: If you have a bunch of adjacent images, gdal_merge.py allows you to mosaic them together to create a single seamless image. This can be handy, for example, when piecing together DRGs for a local area so you can load just the one image along with your other data.

gdal-config
: This utility is used to print the configuration options and other information about the installed version of GDAL, including which raster formats are supported. If you end up getting crazy and compiling your own OSGIS software, gdal-config is almost always used during the configuration process to set things up for compiling with the GDAL/OGR libraries.

OGR-Supported Formats

While GDAL is used with rasters, OGR provides tools to manipulate vector GIS layers. OGR supports a wide variety of formats. In some cases, the OGR library can create many of the formats it supports while others are read-only. Many of the formats can also be georeferenced[15] using the library or the OGR utilities. OGR also has an impressive list

---

15. In this case, georeferencing means the format can contain information about the coordinate system.

of supported formats, some of which I've listed for you here. For the complete list, see the GDAL website.

- Arc/Info Binary Coverage
- Comma-Separated Value (.csv)
- DWG
- DXF
- ESRI Personal GeoDatabase
- ESRI ArcSDE
- ESRI Shapefile
- GML
- GMT
- GPX
- GRASS
- KML
- Mapinfo File
- MySQL
- ODBC
- Oracle Spatial
- PostgreSQL
- SDTS
- SQLite
- U.S. Census TIGER/Line
- VRT—Virtual Datasource
- Informix DataBlade

OGR Utilities

Three utilities come with the OGR library, two of which you are likely to find very useful when working with vector data.

ogrinfo
: This utility displays information about a layer, including the coordinate system, attributes, and number of features. Given the right command-line switches, it will even print out the coordinates of every feature. You'll find ogrinfo to be very handy, especially with data you download or are handed from a stranger. It's even helpful with your own data, when you go back to it months after first creating it and can't remember anything about it.[16]

16. Or maybe I'm the only one with this problem.

ogr2ogr

With ogr2ogr, you can convert a vector layer from one format to another, optionally translating the coordinate system along the way. This can be especially handy when you get some new piece of data and want to get it into your favorite format or coordinate system.

ogrtindex

Unless you're a MapServer user, you likely won't use ogrtindex. It creates a tile index from a group of files (for example, vectors) that can be used with MapServer.

Here's a tip that you'll find useful: both the gdalinfo and the ogrinfo commands accept a --formats switch.[17] This is a quick way to find out which formats are supported for a given installation of GDAL/OGR. This is important because both GDAL and OGR can be compiled and distributed with support for a number of optional features. If in doubt, using --formats is a quick way to see whether the magic you are about to attempt is supported.

For a more comprehensive look at both the GDAL and OGR utilities, see Section 11.2, *Using GDAL and OGR*, on page 186.

Generic Mapping Tools

GMT is written in C

This next set of command-line tools can create some really impressive output. In fact, that's its whole aim—to create quality output that can be printed or included in other documents. The Generic Mapping Tools (GMT) has been around a long time. This is a testament to both its utility and acceptance by the user community. GMT was originally developed in 1988 by Paul Wessel and Walter H. F. Smith and is currently hosted at the University of Hawaii.

GMT allows you to create cartographic-quality maps from the command line. This sounds simple, but in fact it has quite sophisticated features including base map creation, plotting x-y values, lines, and polygons, coordinate transformations, gridding, contouring, and 3D illuminated surfaces.

Now I know you are probably thinking that GMT doesn't exactly fit your idea of a desktop GIS application. In fact, it can be a valuable addition

17. In fact, almost all of the GDAL and OGR utilities accept the --formats switch.

to your toolkit. For an introduction to GMT and its capabilities, see Section 11.1, *GMT*, on page 174.

A.3 Other Tools

As I said earlier, there is a huge selection of OSGIS applications to choose from. It's also impossible to discuss each of these in detail. Our survey included some of the major applications available today—and those that together in some combination form a productive toolkit.

Our survey should have got you started thinking about some of the tools available. Now it's up to you to carry on the survey if you see fit. To get you started, here are some links to lists of OSGIS software and tools that you may want to peruse:

- http://www.osgeo.org
- http://www.freegis.org
- http://opensourcegis.org
- http://en.wikipedia.org/wiki/List_of_GIS_software#Open_source_software
- http://maptools.org

For a good survey that includes both open source desktop and web mapping tools, see "The State of Open Source GIS" by Paul Ramsey.[18]

18. http://www.foss4g2007.org/presentations/viewattachment.php?attachment_id=8

Appendix B

Installing Software

In this appendix, you will find brief information on installing most of the applications we have discussed. As always, it helps to read the installation instructions provided with the software. The following information is of the quick-start variety and will help you get up and running.

We provide information for each platform, assuming of course that the application is supported on each.

B.1 GRASS

The good news is you can get a binary distribution for most major platforms. And if you can't, GRASS builds quite readily on most platforms. In this section, we'll look at the options for each of the major platforms so you can get up and running quickly. All binary and source packages are available from the GRASS website.[1]

Linux

Binaries are readily available for Linux, including a generic tar.gz package. Typically, you will find binaries for the following distributions:

- Generic GNU/Linux
- Debian
- Fedora Core
- Gentoo
- Mandriva
- OpenSuSE

1. http://grass.itc.it

- SuSe 10.2
- Ubuntu

To install, just use the package management tool(s) provided with your distribution (for example, rpm, apt, yum). The package for Generic GNU/Linux is easily installed using the provided installation script for your distribution. For all packages, GRASS depends on a number of supporting libraries that you will also have to install using your package management system.

In the event you can't find a package for your distribution or you just want to live on the edge, you will have to compile GRASS from source. There is a complete "Compiling source code" manual available on the website. Compiling from source is not difficult and may be your best option if you want to stay current with new developments.

Unix

If you are using Solaris, Irix, HP-UX, DEC-Alpha, AIX, or one of the BSD variants, compiling from source is your only option. GRASS should build on POSIX systems using the GNU C compiler.

OS X

For Mac OS X a binary distribution is available from the GRASS website as a disk image (.dmg). Currently, a number of other frameworks are required along with the GRASS image. These are also available from the link on the website.

Installation is standard Mac fare—open the disk image, and drag the application to your Applications folder. Make sure you get all the required frameworks as described on the website.

Windows

For GRASS 6.2.*x*, installation on Windows requires the use of Cygwin (http://cygwin.org). Detailed instructions on installing Cygwin and GRASS are available on the GRASS website.

At version 6.3 of GRASS, there will be a native Windows version that does not require Cygwin. At the time of this writing, there was an experimental build available for download from the GRASS website.[2] If you are a Windows user and are pining for GRASS, it's on its way.

2. http://grass.itc.it/download/index.php

B.2 OpenJUMP

Installing OpenJUMP consists of the following:

1. Downloading the distribution ZIP file
2. Unzipping the distribution
3. Running the start-up script in the bin directory

In reality, there are some tweaks needed to get things working with your operating system. Your best bet is to refer to the install instructions on the OpenJUMP website[3] after unzipping the distribution for the latest information on getting things up and running.

B.3 Quantum GIS

For QGIS, you will find packages for the major platforms, including Linux, Mac OS X, and Windows on the download site.[4] To install on each platform, do the following:

- *Mac OS X*: Mount the disk image (.dmg), and drag QGIS to your Applications folder.
- *Windows*: Run the installer, and follow the instructions to install QGIS.
- *Linux*: If you find a package (rpm, deb) for your Linux distribution, just download and install it. If not, you will have to compile QGIS and its dependencies. Information on building from source is available on the QGIS website.[5]

QGIS depends on a number of other OSGIS packages. When installing on Mac OS X and Windows, these are bundled for you in the package. On Linux and other *nix platforms, you will need to install the dependencies prior to installing or building QGIS. For distributions that have a package manager, this is usually not a difficult task. It's best to look around before building dependencies from scratch to see whether packages or builds for your operating system are available.

3. http://openjump.org/wiki/show/Installation+Instructions
4. http://download.qgis.org
5. http://qgis.org

B.4 uDig

Installing uDig is pretty easy if you are running Windows, Linux, or Mac OS X. Since uDig is based on Eclipse,[6] you obviously need to run it on a platform supported by Eclipse. The uDig download site[7] provides binary releases for the three aforementioned platforms. The uDig folks tell me it should also run on Solaris and any other Unix platform that supports GTK.[8]

Installation is fairly simple for each platform:

- *Mac OS X*: Mount the disk image (.dmg), and drag uDig to your Applications folder.
- *Linux*: Unzip the distribution file.
- *Windows*: Run the downloaded .exe to install uDig.

Currently, the Windows and Linux binary distributions include a Java runtime and are ready to run. On Mac OS X, Java is already installed, and you are good to go. Since all the dependencies needed for uDig are packaged with it, you don't need to download and install anything else. This makes it quick and easy to get going.

B.5 GMT

As with most OS GIS applications, you have choices when it comes to installing GMT. If you are lucky enough to have a system that uses apt for package management, it can be as easy as finding out which packages are available:

```
root@madison:~ # apt-cache search "Generic Mapping Tools"
gmt - Generic Mapping Tools
gmt-coast-low - Low resolution coastlines for the Generic Mapping Tools
gmt-doc - HTML documentation for the Generic Mapping Tools
gmt-doc-pdf - PDF docs for the Generic Mapping Tools
gmt-doc-ps - PostScript docs for the Generic Mapping Tools
gmt-manpages - Manpages for the Generic Mapping Tools
gmt-tutorial-pdf - Tutorial for the Generic Mapping Tools (PDF)
gmt-tutorial-ps - Tutorial for the Generic Mapping Tools (PostScript)
```

---

6. Eclipse is an "open development platform comprised of extensible frameworks, tools and runtimes for building, deploying, and managing software." See http://www.eclipse.org.
7. http://udig.refractions.net
8. GTK is a multiplatform toolkit for creating graphical user interfaces. It was originally developed and used in the creation of the GNU Image Manipulation Program (GIMP). See http://gtk.org.

You might want to broaden your search a bit by searching for gmt to make sure you haven't missed any goodies. Once you have located them, you can easily install the components you want using apt-get:

```
root@madison:~ # apt-get install gmt gmt-coast-low gmt-doc-pdf gmt-examples
Reading package lists... Done
Building dependency tree... Done
gmt-examples is already the newest version.
The following NEW packages will be installed:
gmt gmt-coast-low gmt-doc-pdf
0 upgraded, 3 newly installed, 0 to remove and 246 not upgraded.
Need to get 0B/14.9MB of archives.
After unpacking 20.6MB of additional disk space will be used.
Selecting previously deselected package gmt.
(Reading database ... 160635 files and directories currently installed.)
Unpacking gmt (from .../gmt_4.0-2build1_i386.deb) ...
Selecting previously deselected package gmt-coast-low.
Unpacking gmt-coast-low (from .../gmt-coast-low_20020411-1_all.deb) ...
Selecting previously deselected package gmt-doc-pdf.
Unpacking gmt-doc-pdf (from .../gmt-doc-pdf_4.0-2build1_all.deb) ...
Setting up gmt (4.0-2build1) ...
Setting up gmt-coast-low (20020411-1) ...
Setting up gmt-doc-pdf (4.0-2build1) ...
root@madison:~ #
```

Operating systems that support apt include Debian and its variants (for example, Ubuntu) and Mac OS X with Fink.[9] If you aren't fortunate enough to have one of these systems, you may find packages available for you Linux variant. Check your package management tool to see whether GMT is available. If not, refer to the GMT home page to see what your options are. You always have the option to build from source on your platform, and the GMT website[10] can generate a configuration file based on your choices. You then feed the configuration file to the install_gmt script, and it handles downloading, building, and installing the software.

If you are a Windows user, your options are a bit more limited. The GMT folks suggest installing Cygwin and then building GMT as you would for Unix. Again, see the website for your options.

9. http://fink.sourceforge.org
10. http://gmt.soest.hawaii.edu

B.6 GDAL/OGR

- *Linux*: Binaries for GDAL and OGR are available from the GDAL website.[11] For Linux, check for the current packages, and install using your package manager. If you don't find current binaries for your Linux distribution, then you will have to build from source or install FWTools (Section B.7, *FWTools*).
- *Mac OS X*: Binaries frameworks are available from the Kyng Chaos website.[12] Just download and install the .dmg per the instructions found in the ReadMe.rtf.
- *Windows*: Binaries and plugins for Windows are available from the OSGeo download site.[13] Download the latest version and extract it to a directory in your path.

For Linux and Windows, you may find that FWTools is a better way to go—you get GDAL/OGR as well as a bunch of other handy applications.

B.7 FWTools

FWTools is an open source GIS binary kit for Linux and Windows. It includes GDAL/OGR, the projections library PROJ4, MapServer, and Python, among others. It's a complete runtime environment with no external dependencies. Install it, and you are ready to use all the included applications and utilities.

To install FWTools, do the following:

- *Linux*: Download and untar the distribution, change to the FWTools directory, and execute the install.sh script. Once it's complete, you'll need to add the bin_safe directory to your path.
- *Windows*: Download the installer and run it, and then follow the instructions to complete the installation. The installation creates a shortcut to start an FWTools shell from which you can use the applications.

11. http://www.gdal.org
12. http://www.kynchaos.com
13. http://download.osgeo.org/gdal/win32

Appendix C

GRASS Basics

Once you have GRASS installed, setting up GRASS and creating your locations is key to getting off the ground. This appendix will guide you through creating a location using both QGIS and the GRASS shell. From there you'll get a basic introduction to working with GRASS GIS. Once you've mastered the basics of GRASS, you might be interested in diving deeper with *Open Source GIS: A GRASS GIS Approach* by Markus Neteler and Helena Mitasova.

C.1 Location, Location, Location

Why are we repeating ourselves? Because it's real important. One of the key concepts in using GRASS is that of a *location*. Remember the old axiom about location used in both real estate and business? Well, the same holds true for GRASS. Once you get the concept down, you will have mastered what many have stumbled on.

Quick Start

We are going to "cheat" to get started with GRASS data by using the QGIS/GRASS plugin to create a new location and mapset. We'll then go back and look at alternative ways to do the same. This method makes it very easy to create a new location. We simply add one or more layers to QGIS that encompass the geographic region of interest and then use the plugin to do the dirty work. Before we do that, though, we need a GRASS database to get started with.

A GRASS database is simply a directory where we will store our GRASS locations and mapsets, along with the data. You can have as many

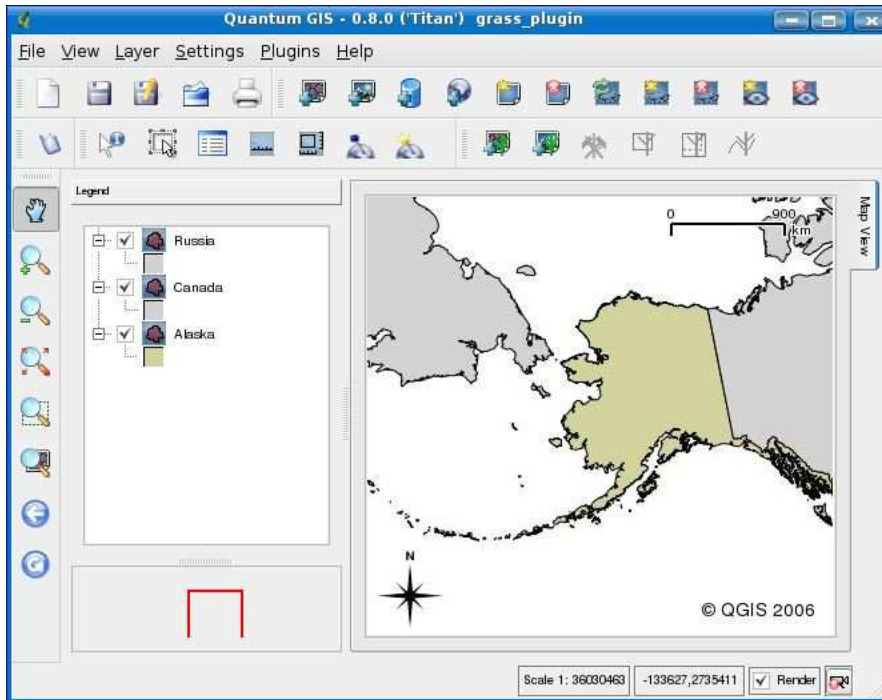


Figure C.1: QGIS with layers used to create a GRASS location

GRASS databases as you desire. For this example, we'll use a directory in our home directory named grassdata. The QGIS/GRASS plugin can create this directory as part of defining our new location and mapset.

Creating the Location

Let's start by running QGIS and loading the data that covers the area of interest. We'll use the standard Alaska sample data here and load just the coastline and the Russia/Canada boundaries. If you don't already have the sample data, you can get it from http://desktopgisbook.com. In Figure C.1, you can see QGIS with the data loaded and ready for us to use the GRASS plugin.

Now it's time to create the database, location, and mapset. Choose GRASS from the Plugins menu and choose New Mapset. Here you can choose an existing database directory or create a new one. The dialog box provides some visual cues as to how you should proceed. In Figure C.2, on the following page, you can see the database selection

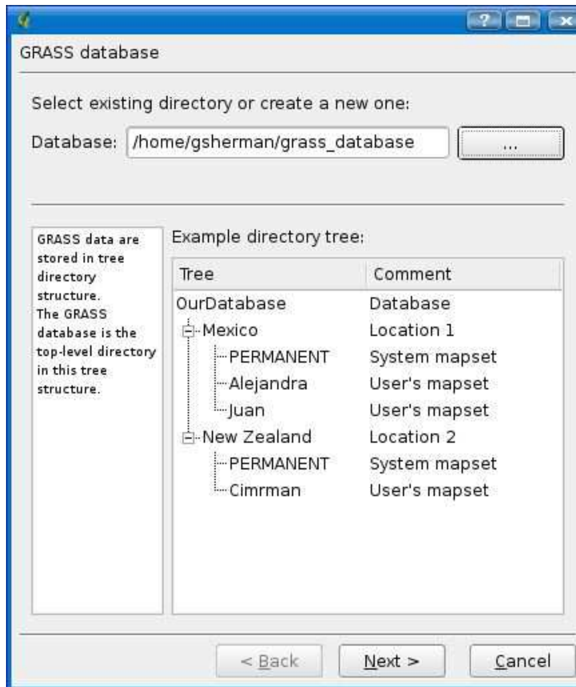


Figure C.2: GRASS database selection/creation dialog box

dialog box with values for our new database plumbed in (to create a new database, just create a new folder using the browse button to the right of the Database field).

Clicking the Next button brings us to the location screen. To create a location, we simply provide a name, in this case alaska_data. Note that the plugin will use not only the extent of the data loaded into QGIS but also the projection information when creating the new location. Clicking Next opens the projection page where we can customize the projection if we choose to do so. If the layers loaded in to QGIS have associated projection information, it will be preselected in the Projection list. In this case, "NAD 27 / Alaska Albers (meters)" is selected. We don't have to do anything further to define the coordinate system.

Moving to the next screen (by clicking the Next button) brings us to Default GRASS Region. Here you can customize the region (think extents) for the location. By default, it will be set to the same extent as the data loaded in QGIS. Usually you won't have to make any changes here.

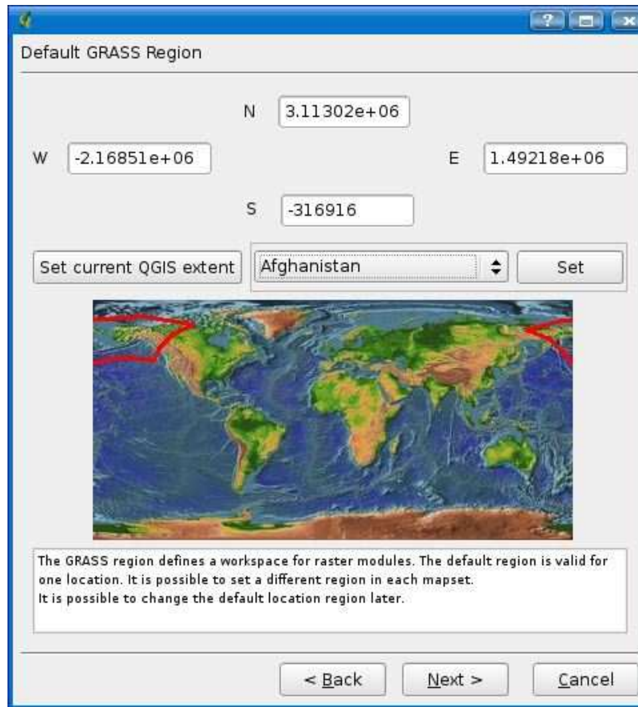


Figure C.3: Default region populated from layers loaded in QGIS

Note that this screen allows you to set the extent by choosing countries or regions from a drop-down list. This can be useful if you want to create a new location but don't have any layers loaded in QGIS. In this case you need to specify the region, as well as the projection information since it won't be automatically populated for you. In Figure C.3, you can see the region screen with the extents set automatically from our data loaded in QGIS. Notice the extents are shown graphically in red on the world map.

Keep in mind that we have defined a default region. The region can be changed for each mapset or for the entire location later.

Clicking the Next button brings us to the final screen in the process. Here we give the mapset a name. As explained in the dialog box, a mapset is simply a collection of maps (think layers here) used by a single user. You can name the mapset whatever you like. In a multiuser environment, a user ID might be a good choice. Enter a name, and click

Next. This brings up a summary of what we just did, telling us that we have created a database in our chosen directory, a named location, and a mapset.

Click Finish, and we are done. You now have a new location and mapset ready for your use. The plugin automatically opens it and sets it as the current location. You should be able to see the region delineated by a red rectangle on the map canvas.

Using the QGIS/GRASS plugin allowed us to create the new location without knowing the gory details about the extent of our data and projection parameters. Once you've become accustomed to the GRASS paradigm, you may choose to create your locations/mapsets using the GRASS shell.

Creating a Location with GRASS

You are probably noticing about now that we haven't run GRASS yet. So far everything we have done has been through QGIS using the GRASS plugin. Now it's time to show you how to start GRASS and create a location. Starting GRASS depends on your platform:

- *Linux*: From a terminal window, run the GRASS binary using grass combined with the version number. So to start GRASS 6.2, you would use grass62 from the command line or the menu.
- *Mac OS X*: Start by double-clicking the GRASS icon in your Applications folder.
- *Windows*: Start your Cygwin shell, and start GRASS by running grass62 or start it from the menu.

When you start GRASS, you are presented with either a text-based form or a GUI dialog box requesting the location and mapset you want to use for the session. Which you see depends on how you started GRASS and which mode you used last (GRASS remembers and starts up in the last-used mode). For the purpose of this example, we started GRASS from the command line and specified the -text switch to force start-up in text mode.

In Figure C.4, on the following page, you can see the start-up screen for GRASS 6.2.2. Starting GRASS requires three pieces of information: the database directory, location, and mapset. These requirements are nicely explained on the form.

```
                        GRASS 6.2.2

DATABASE: A directory (folder) on disk to contain all GRASS maps and data.

LOCATION: This is the name of a geographic location. It is defined by a
          co-ordinate system and a rectangular boundary.

MAPSET:   Each GRASS session runs under a particular MAPSET. This consists of
          a rectangular REGION and a set of maps. Every LOCATION contains at
          least a MAPSET called PERMANENT, which is readable by all sessions.

         The REGION defaults to the entire area of the chosen LOCATION.
         You may change it later with the command: g.region
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

LOCATION:   world_lat_lon___________   (enter list for a list of locations)
MAPSET:     gsherman________________   (or mapsets within a location)

DATABASE: /home/gsherman/grassdata_____________________________________________


         AFTER COMPLETING ALL ANSWERS, HIT <ESC><ENTER> TO CONTINUE
                        (OR <Ctrl-C> TO CANCEL)
```

Figure C.4: GRASS start-up form

If you don't have a directory for storing your GRASS locations in, create it before you start GRASS, or pop out to another shell and create it. You can store locations in any directory, but it's best to establish a structure for your data to keep it apart from other non-GRASS files and directories on your system.

In Figure C.4, we have entered the required information to get started. Since the location doesn't exist yet, we have to provide some additional information as we proceed. Hitting Esc+Enter tells us that the location we specified (world_lat_long) doesn't exist and asks whether we want to create it. It also shows a list of the current locations (empty in our case) in the database in case we made a mistake and want to select an existing one. To continue, press Enter or enter n and Enter to cancel the process. Next GRASS tells us what we need before we can proceed to create the location:

- The coordinate system for the database
- The zone for the UTM database and all the necessary parameters for projections other than Latitude-Longitude; x,y; and UTM

- The coordinates of the area to become the default region and the grid resolution of this region
- A short, one-line description or title for the location

We will be creating a worldwide location in geographic (latitude/longitude) coordinates so we know the answer to all the questions. The second requirement doesn't apply since we aren't using UTM or some other projection. Pressing Enter brings us to the coordinate system selection screen. The options are as follows:

- A x,y
- B Latitude-Longitude
- C UTM
- D Other Projection

Option B is what we want, so we press B and then Enter. If we had specified one of the others, we would have had to enter additional information that requires having some information about the projection and its parameters. GRASS asks us to confirm our selection by entering Y or just pressing Enter since yes is the default answer (indicated by the brackets around "y" in the question):

```
Latitude-Longitude coordinate system? (y/n) [y]
```

GRASS now asks for a one-line description of the location. We'll use something simple like "World wide location in latitude and longitude." After confirmation, we are asked whether we want to specify a datum. This is always a good idea—in fact, it's essential to ensure your maps are transformed properly should the need arise. For this location, we will use WGS84. We also have to set a datum transformation parameter. The transcript for this part of the process is as follows:

```
Please enter a one line description for location <world_lat_lon>

> World wide location in latitude and longitude
=====================================================
World wide location in latitude and longitude
=====================================================
ok? (y/n) [y]
Do you wish to specify a geodetic datum for this location?(y/n) [y] Y

Please specify datum name
Enter 'list' for the list of available datums
or 'custom' if you wish to enter custom parameters
Hit RETURN to cancel request
>WGS84
```

```
Now select Datum Transformation Parameters
Please think carefully about the area covered by your data
and the accuracy you require before making your selection.

Enter 'list' to see the list of available Parameter sets
Enter the corresponding number, or <RETURN> to cancel request
>list
Number  Details
---
1       Used in whole wgs84 region
        (PROJ.4 Params towgs84=0.000,0.000,0.000)
        Default 3-Parameter Transformation (May not be optimum for older datums;
 use this only if no more appropriate options are available.)
---

Now select Datum Transformation Parameters
Please think carefully about the area covered by your data
and the accuracy you require before making your selection.

Enter 'list' to see the list of available Parameter sets
Enter the corresponding number, or <RETURN> to cancel request
>1
```

Now GRASS pops up the screen to define the default region. Even though we know we are creating a worldwide location, GRASS doesn't yet. This screen provides a visual representation of the four required edges, as well as the default grid resolution. In the case of vector layers, grid resolution has no effect since the coordinates are stored at full precision, as are rasters that you import. The grid resolution applies to new raster maps you create in your mapset. In our example we'll just use a resolution of 1 degree since we don't have any particular information about rasters we might want to import. Since we want to cover the whole world, we enter the coordinates, as shown in Figure C.5, on the following page. When we are happy with the values, pressing Esc+Enter takes us to a summary screen where we can confirm everything before proceeding. Pressing Enter creates the new location:

```
projection: 3 (Latitude-Longitude)
zone: 0
  north:       90N
  south:       90S
  east:        180E
  west:        180W

  e-w res:     1
  n-s res:     1

total rows: 180
total cols: 360
total cells: 64,800
```

```
                    DEFINE THE DEFAULT REGION


                 ====== DEFAULT REGION =======
                 | NORTH EDGE:90N_______     |
                 |                           |
      WEST EDGE  |                           |EAST EDGE
      180W______ |                           |180E______
                 | SOUTH EDGE:90S_______     |
                 =============================


      PROJECTION: 3 (Latitude-Longitude)        ZONE: 0

                      GRID RESOLUTION
                          East-West:     1_________
                        North-South:     1_________


      AFTER COMPLETING ALL ANSWERS, HIT <ESC><ENTER> TO CONTINUE
                         (OR <Ctrl-C> TO CANCEL)
```

Figure C.5: Defining the default GRASS region

```
Do you accept this region? (y/n) [y] >y
LOCATION <world_lat_lon> created!

Hit RETURN -->
```

When we hit Enter, we end up back at a screen that looks exactly like Figure C.4, on page 301. But now the location exists, so pressing Esc+Enter actually gets us to the GRASS command prompt (this example is from a Linux session, so your output will look a bit different, depending on your operating system):

```
Welcome to GRASS 6.2.2 (2007)
GRASS homepage:                          http://grass.itc.it/
This version running thru:               Bash Shell (/bin/bash)
Help is available with the command:      g.manual -i
See the licence terms with:              g.version -c
Start the graphical user interface with: gis.m &
When ready to quit enter:                exit

GRASS 6.2.2 (world_lat_lon):~ >
```

The entire process of creating a location was done from a shell session because we started GRASS without any command-line switches. GRASS also provides a GUI start-up using grass62 -gui. Let's look at one more way to create a location, using information from one of our layers and the GRASS GUI.

Start GRASS using grass62 -gui. On some platforms you can get the same effect by double-clicking the GRASS desktop icon or selecting it from your application menu. Once the GUI starts, we are presented with a form that allows us to start working right away, create a new mapset, or create a new location. In Figure C.6, on the following page, we see that the location we created using the command line shows up by default and is ready to use. Our interest is in creating a new location, using the Georeferenced File button found under the "Define a new location with..." heading.

When you click the button to create a location from a georeferenced file, a small dialog box opens where you can specify the name for the new location, the path (by default this is your current GRASS database location), and importantly, the file to be used in creating the location. Using the browse button, we can browse to the location of our world_borders shapefile and use it to create the location. Once we have the path for it filled in, we click the Define Location button.

That's all there is to it. GRASS creates the new location and tells you it needs to restart in order to work with the new location. After it closes, start up the GRASS GUI again, and you will find the newly created location ready for use—with one exception. There are no mapsets yet, other than the PERMANENT mapset. The PERMANENT mapset is for shared layers. We need to create a mapset for us to use for loading our data and eventually doing some editing. From the GUI, it's easy to create a mapset by selecting the location you want to add it to and then filling in a name for the mapset. Click the Create New Mapset button, and it's done. You can now select it from the list and click the Enter GRASS button to continue. This brings up the grass shell, and it also starts the GRASS display manager. For now, just exit GRASS, unless you want to play around with your new mapset and do some exploration.

In this section, we saw how several ways to create a new location. We didn't look at them all but just enough to get us started. The GRASS GUI also provides the ability to create a location using an EPSG code or by entering projection values. Of all the methods available to us, using QGIS and the Georeferenced File method in the GRASS GUI are

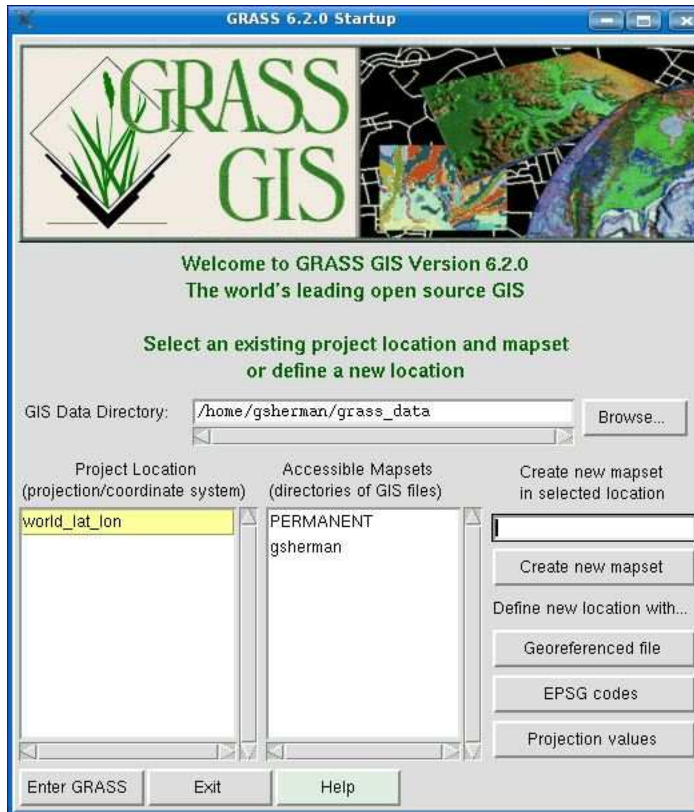


Figure C.6: GRASS GUI start-up screen

probably the quickest and easiest. If you are familiar with projections and EPSG codes, you may find the other methods just as easy.

C.2 Getting Some Data

A GRASS location and mapset is nice, but you obviously need some data to work with. GRASS uses its own format for storing data. Unlike the OSGIS viewers we have discussed (Thuban, uDig, QGIS) that read a number of formats, GRASS prefers its own format. That's not to say you can't use external data with GRASS because you can. To get full advantage of the capabilities, importing the data is required.

A Word About GRASS Commands

GRASS commands are arranged by function. For example, vector commands begin with v., raster commands begin with r., database commands with d., and general commands with g.. For a complete list of these commands and their function, refer to the online GRASS manual that is installed with GRASS. You can easily access the manual using the g.manual -i command from the GRASS shell.

This section will show you several ways to get some data into GRASS so you can work with it, including the following:

- Import the data using v.in.ogr and r.in.gdal GRASS commands.
- Use the GRASS plugin in QGIS to import a loaded vector layer.
- Use v.external to make an external layer (for example a shapefile) accessible to GRASS.

Using the Command Line

Let's start by importing some vector data into GRASS using v.in.ogr. This command allows you to import data sources supported by the OGR library (see Section A.2, *OGR Utilities*, on page 287 for information on supported formats). This means we can use it to import a shapefile, among others. The world_borders shape is stored in the desktop_gis_data folder (if you are following along, your location may vary). We will use it to import into GRASS.

Let's look at part of the syntax help for v.in.ogr. If you want to get the full picture, use g.manual v.in.ogr:

```
GRASS 6.2.2 (world_lat_lon):~ > v.in.ogr --help

Description:
 Convert OGR vectors to GRASS. Available drivers:
 ESRI Shapefile,MapInfo File,UK .NTF,SDTS,TIGER,S57,
 DGN,VRT,AVCBin,REC,Memory,CSV,GML,PostgreSQL,GRASS

Usage:
 v.in.ogr [-lfcztoe] dsn=string output=name [layer=string[,string,...]]
   [spatial=value[,value,...]] [where=sql_query] [min_area=value]
   [type=string[,string,...]] [snap=value] [location=string]
   [cnames=string[,string,...]] [--overwrite]
```

```
Flags:
  -l   List available layers in data source and exit
  -f   List available formats and exit
  -c   Do not clean polygons (not recommended)
  -z   Create 3D output
  -t   Do not create attribute table
  -o   Override projection (use location's projection)
  -e   Extend location extents based on new dataset
 --o   Force overwrite of output files

Parameters:
     dsn   OGR datasource name. Examples:
              ESRI Shapefile: directory containing shapefiles
              MapInfo File: directory containing mapinfo files
  output   Name for output vector map
  layer   OGR layer name. If not given, all available layers are imported.
          Examples:
              ESRI Shapefile: shapefile name
              MapInfo File: mapinfo file name
```

If you happen to invoke a GRASS command without any parameters, a GUI window will pop up, allowing you to set the options and execute the command. To import our layer, we'll use the command line rather than the GUI.

There are a lot of optional flags and parameters we can use with v.in.ogr. For our example, we are going to go the simple route and see whether it gives us the expected results. To import the layer, use the following:

```
v.in.ogr dsn=/home/gsherman/desktop_gis_data output=world_borders \
  layer=world_borders
```

The dsn is the data source name and, in the case of OGR vectors, refers to the directory where the layers are stored. In this case, that's the directory where the world_borders layer lives: /home/gsherman/desktop_gis_data. If you don't specify a layer name using the layer parameter, all layers in the dsn directory will be imported. This obviously can be pretty handy for bringing in a lot of layers all at once.

Importing the layer will take a while. GRASS does a number of things when importing a layer, including building topology as appropriate. Part of the output from the import process is shown next. We didn't include the entire output because it gets quite long and includes detailed information about the import process and building topology.

Joe Asks...

What Is Topology?

You probably noticed the word *topology* has cropped up a couple of times now. In simple terms, topology is the relationship between spatial features. For example, a polygon boundary consists of lines. Adjacent polygons share common boundaries. Data formats that are topological maintain this relationship when creating and editing data. The common boundaries are stored only once.

In a nontopological format, the common boundaries are duplicated, one for each polygon.

Apart from the storage difference, a topological GIS such as GRASS maintains the spatial relationships as you edit data. If you move a line, the polygon boundary or boundaries are adjusted accordingly. This provides consistency in your data and is essential when performing many geoprocessing tasks.

```
GRASS 6.2.2 (world_lat_lon):~ > v.in.ogr dsn=/home/gsherman/desktop_gis_data \
     output=world_borders layer=world_borders cnames=dog
A datum name wgs84 (WGS_1984) was specified without transformation parameters.
Note that the GRASS default for wgs84 is towgs84=0.000,0.000,0.000.
Projection of input dataset and current location appear to match.
Proceeding with import...
Layer: world_borders
Importing map 3784 features...
-----------------------------------------------------
Building topology ...
```

If the import's successful, you should be returned to the GRASS prompt without any error messages. You can quickly confirm that we now have a world_borders layer using the following:

```
GRASS 6.2.2 (world_lat_lon):~ > g.list vect
----------------------------------------------
vector files available in mapset gsherman:
world_borders
----------------------------------------------
```

Now let's bring in a raster using the r.in.gdal command. This command uses the GDAL library and therefore can import a wide range of formats. For this example, we will import the NASA world mosaic into GRASS.

The syntax of r.in.gdal is as follows:

```
Usage:
 r.in.gdal [-oefk] input=string output=name [band=value]
   [target=string] [title="phrase"] [location=string] [--overwrite]

Flags:
  -o   Override projection (use location's projection)
  -e   Extend location extents based on new dataset
  -f   List supported formats then exit
  -k   Keep band numbers instead of using band color names
 --o   Force overwrite of output files

Parameters:
     input   Raster file to be imported
    output   Name for output raster map
      band   Band to select (default is all bands)
    target   Name of location to read projection from for GCPs transformation
     title   Title for resultant raster map
  location   Name for new location to create
```

The options here are fewer than with v.in.ogr. To import the raster, use the following:

```
GRASS 6.2.2 (world_lat_lon):~ > r.in.gdal \
  input=./ev11612_land_ocean_ice_8192.tif output=nasa_world_mosaic
  ERROR: Projection of dataset does not appear to match current location.

       LOCATION PROJ_INFO is:
       name: Latitude-Longitude
       datum: wgs84
       towgs84: 0.000,0.000,0.000
       proj: ll
       ellps: wgs84

       cellhd.proj = 0 (unreferenced/unknown)

       You can use the -o flag to r.in.gdal to override this check and use
       the location definition for the dataset.
       Consider generating a new location from the input dataset using the
       'location' parameter.
GRASS 6.2.2 (world_lat_lon):~ >
```

Oops—what happened? GRASS doesn't like our raster because it thinks it may be in a different projection than our WGS84 location. We can override this using the -o switch if we're sure the projection is correct:

```
GRASS 6.2.2 (world_lat_lon):~ > r.in.gdal -o \
  input=./ev11612_land_ocean_ice_8192.tif output=nasa_world_mosaic
Over-riding projection check.
Proceeding with import...
WARNING: G_set_window(): Illegal latitude for North
GRASS 6.2.2 (world_lat_lon):~ >
```

Now what? It appears that GRASS isn't happy with the raster's northern latitude value(s). Let's use gdalinfo to examine the raster so we can determine what's going on:

```
GRASS 6.2.2 (world_lat_lon):~/desktop_gis_data > gdalinfo -nomd \
  ./ev11612_land_ocean_ice_8192.tif
Driver: GTiff/GeoTIFF
Size is 8192, 4096
Coordinate System is `'
Origin = (-180.021973,90.021973)
Pixel Size = (0.04394530,-0.04394530)
Corner Coordinates:
Upper Left  (-180.0219727,  90.0219726)
Lower Left  (-180.0219727, -89.9779762)
Upper Right ( 179.9779249,  90.0219726)
Lower Right ( 179.9779249, -89.9779762)
Center      (  -0.0220239,   0.0219982)
Band 1 Block=8192x1 Type=Byte, ColorInterp=Red
Band 2 Block=8192x1 Type=Byte, ColorInterp=Green
Band 3 Block=8192x1 Type=Byte, ColorInterp=Blue
Band 4 Block=8192x1 Type=Byte, ColorInterp=Undefined
GRASS 6.2.2 (world_lat_lon):~/desktop_gis_data >
```

This provides us with a lot of detail information about the raster. For now, we are interested in the coordinates. Notice that our image extends beyond the bound of the world (at least as we defined it in our GRASS location). The image appears to be shifted to the upper left. This is because a world file specifies the position of the *center* of the upper-left pixel. You can see that the shift is exactly half the pixel size of 0.04394530. So, how do we solve this problem so we can complete the import? The answer is to simply translate the bounding rectangle so it is correct. To do that, we will use the gdal_translate command:

```
  GRASS 6.2.2 (world_lat_lon):~/desktop_gis_data > gdal_translate -of GTiff \
  -a_ullr -180 90 180 -90 -co "COMPRESS=LZW" -a_srs EPSG:4326 \
  ./ev11612_land_ocean_ice_8192.tif world_mosaic.tif
Input file size is 8192, 4096
0...10...20...30...40...50...60...70...80...90...100 - done.
```

Using gdal_translate we translated the raster and created a new one with the proper bounding rectangle (-a_ullr -180 90 180 -90). We didn't change the data; all we did was remove the shift because of the way world files are designed. During the process, we specified the image should be compressed using LZW (-co "COMPRESS=LZW") and also assigned the WGS84 projection to it (-a_srs EPGS:4326). The new image is a true GeoTIFF. It doesn't have a world file but has the projection information encoded in the file itself.

Using gdalinfo on the new world_mosaic.tif yields the following:

```
GRASS 6.2.2 (world_lat_lon):~/desktop_gis_data > gdalinfo -nomd world_mosaic.tif
Driver: GTiff/GeoTIFF
Size is 8192, 4096
Coordinate System is:
GEOGCS["WGS 84",
    DATUM["WGS_1984",
        SPHEROID["WGS 84",6378137,298.2572235629972,
            AUTHORITY["EPSG","7030"]],
        AUTHORITY["EPSG","6326"]],
    PRIMEM["Greenwich",0],
    UNIT["degree",0.0174532925199433],
    AUTHORITY["EPSG","4326"]]
Origin = (-180.000000,90.000000)
Pixel Size = (0.04394531,-0.04394531)
Corner Coordinates:
Upper Left  (-180.0000000,  90.0000000) (180d 0'0.00"W, 90d 0'0.00"N)
Lower Left  (-180.0000000, -90.0000000) (180d 0'0.00"W, 90d 0'0.00"S)
Upper Right ( 180.0000000,  90.0000000) (180d 0'0.00"E, 90d 0'0.00"N)
Lower Right ( 180.0000000, -90.0000000) (180d 0'0.00"E, 90d 0'0.00"S)
Center      (   0.0000000,   0.0000000) (  0d 0'0.01"E,  0d 0'0.01"N)
Band 1 Block=8192x1 Type=Byte, ColorInterp=Red
Band 2 Block=8192x1 Type=Byte, ColorInterp=Green
Band 3 Block=8192x1 Type=Byte, ColorInterp=Blue
Band 4 Block=8192x1 Type=Byte, ColorInterp=Alpha
GRASS 6.2.2 (world_lat_lon):~/desktop_gis_data >
```

The important things to note in the output of gdalinfo are as follows:

- The coordinate system WGS84 has been assigned and encoded in the file.
- The pixel size hasn't changed.
- The bounding coordinates are now correct.

At last we can import the image into GRASS. Now that the image has been tortured into submission, we don't have to supply any flags to the r.in.gdal command:

```
GRASS 6.2.2 (world_lat_lon):~/desktop_gis_data > r.in.gdal \
  input=./world_mosaic.tif output=world_mosaic
A datum name wgs84 (WGS_1984) was specified without transformation parameters.
Note that the GRASS default for wgs84 is towgs84=0.000,0.000,0.000.
Projection of input dataset and current location appear to match.
Proceeding with import...
 ...
r.in.gdal complete.
GRASS 6.2.2 (world_lat_lon):~/desktop_gis_data >
```

Success! We stumbled across a bit of a problem initially, but this served to illustrate some problem-solving tools and techniques at our disposal. You should note that you won't encounter the offset issue with every raster you import. If you are importing a GeoTIFF that contains coordinate information, odds are it will be processed just fine. If not, you are now prepared to deal with the problem.

The last thing we want to do is color-composite the image for display purposes. The import processed each band of the image separately, creating separate outputs for red, green, blue, and the alpha channel. We can combine these together so the image looks like we expect using r.composite:

```
GRASS 6.2.2 (world_lat_lon):~ > r.composite red=world_mosaic.red \
  green=world_mosaic.green blue=world_mosaic.blue output=world_mosaic
 ...
GRASS 6.2.2 (world_lat_lon):~ >
```

In Figure C.7, on the following page, you can see the world mosaic displayed in GRASS. You might notice it looks a little "blocky." If you zoom in, you will find it loses the detail we had in the original TIFF image. We should have set the region to that of the raster before creating the composite; otherwise, GRASS uses the default region specified when we created the location. To get the results we want, we can composite the image again, after setting the region using one of the rasters (world_mosaic.red) created during the import:

```
GRASS 6.2.2 (world_lat_lon):~ > g.region rast=world_mosaic.red
GRASS 6.2.2 (world_lat_lon):~ > r.composite red=world_mosaic.red \
  green=world_mosaic.green blue=world_mosaic.blue output=world_mosaic_better
 ...
GRASS 6.2.2 (world_lat_lon):~ >
```

Now we have a proper looking raster that preserves the resolution of the original TIFF file.

Importing with QGIS

Now let's look at using QGIS to import a layer into GRASS. After QGIS is fired up, open the mapset we want to use for storing our data. To open a mapset, choose Open mapset from the GRASS menu under the main Plugins menu. Make the appropriate selections for location and mapset, and press OK. Nothing much happens when you do this, other than the GRASS toolbox icon will now be enabled. Now that the mapset is open, we need to load the shapefile we want to import. To import a shapefile

Figure C.7: World mosaic in GRASS

into GRASS, it must first be loaded into QGIS using the Add a Vector Layer menu or toolbar icon. In our case, we will use the cities layer.

Once the cities layer is loaded into QGIS, open the GRASS toolbox by clicking the tool in the GRASS toolbar or by choosing Open GRASS tools from the GRASS plugin menu. The toolbox contains a wealth of GRASS tools and functions we can run from within QGIS. Right at the top of the list you will find Import OGR/PostGIS Vector Layer. This tool will give us the opportunity to import any loaded vector layer into GRASS, assuming it's an OGR or PostGIS layer.

Once we click the import tool, a new tab page is opened, and a drop-down box of all the eligible layers is available. We just pick the one we want to import from the list and then fill in a name for the GRASS layer (output vector map). Clicking the Run button starts the import process. The output from the command will be displayed as the import proceeds. If all goes well, a "Successfully finished" message will be displayed at the end of the output. If you scroll back through the output window, you will find that the command used to do the import is v.in.ogr. The GRASS plugin in QGIS just provided a convenient front end for importing the layer. We could easily have accomplished the same thing using the GRASS command line.

If you click the Add GRASS vector layer tool you will see that our GRASS mapset now contains both the world_borders layer and the cities layer. We can use the same method to import rasters into GRASS, using the Import GDAL Raster Layer tool in the toolbox. If we encounter difficulties with the raster as we did the NASA world mosaic, some prep work using gdalinfo and gdal_translate may be required.

Importing External Data

The last means of getting vector data into GRASS involves importing external data. Actually, it's not really an import but more of a link. When working with external data, you have to be aware that a number of GRASS tools and operations won't work. But it is a handy way to display data without converting to a GRASS layer. For information on which OGR formats can be linked, see the manual for v.external.

The syntax for v.external is almost exactly like the simplest form of v.in.ogr:

```
v.external dsn=string [output=name] [layer=string] [--overwrite]
```

To link the world_borders shapefile, we would use the following:

```
v.external dsn=./desktop_gis_data output=world_borders_external \
    layer=world_borders.shp
```

We can now use the layer in GRASS—just remember it is read-only and may yield incorrect results when used in some GRASS operations. This is because an external layer is fundamentally different from a true GRASS layer. For example, an external layer doesn't have true topology but a pseudo-topology is created to allow it to act like a complete GRASS layer. Again, if you want to use a layer for operations other than display, it's best to import it into GRASS using one of the methods we have discussed in this chapter.

C.3 Working with Data

Now that we know how to get data into GRASS, let's work with it a bit and learn how to use the display manager. You may have gathered by now that GRASS has a couple of start-up modes—GUI and text. GRASS remembers which mode you used last and will attempt to start up using the same the next time around. Not only are there two modes, there are also two GUIs to choose from. By default if you start GRASS using grass62 -gui, you get the "official" display manager gis.m. The other, older display manage is d.m. If you prefer it, start GRASS in text mode

GRASS GUIs

GRASS itself has two GUI interfaces currently: gis.m and d.m. There is currently work being done on a new interface using Python and wxWindows.* That interface isn't far enough along for us to use in our examples, but it's something to keep an eye on. Oh, and don't forget the other GRASS GUI option—Quantum GIS.

*. http://wxwindows.org

using grass62 -text, and then enter the command to bring it up. So far we have been using GRASS 6.2 in our examples. Now we will switch to the bleeding edge a bit and use GRASS 6.3. Everything we have done so far in terms of creating locations and mapsets is the same between 6.2 an 6.3, so you don't have to start over and learn something new. In fact, our locations and mapsets all work with 6.3, so we are in good shape.

To view GIS data, obviously you need a GUI, so we will start up GRASS 6.3 in that mode, using the -gui switch. This gives us the new and improved display manager. The first thing you will notice is that three windows pop up. GRASS requires a bit of screen real estate in which to operate. You can use it on a 12-inch laptop display, but that's not ideal. The first step is to rearrange the windows so you can get a look at all of them at the same time. With a bit of resizing and moving, you will be able to fit them all nicely on your display. You should now have the GIS Manager, Map Display 1, and Output windows arranged so each is visible. Let's add a vector layer and see what happens.

Everything starts with the GIS Manager. Its toolbars are organized by function, in particular the raster and vector functions are those we are interested in at the moment. Like other GIS applications, GRASS has tooltip text for each button on the toolbar(s). Hover the mouse to learn what each tool is for, or consult the manual. The button to add a vector layer looks like a stream with a green polygon and red markers on a white background. Clicking it adds a generic vector layer to the manager window. To actually display something, we have to add a layer to this vector "object." To do this, click the newly added Vector 1 item. This opens up a new form on the lower half of the manager window where we can set up the display parameters and options. To specify the layer to use, click the Vector Map to Display button just to the left of

the first textbox. This will display a list of layers in the mapset that you can choose from. For our example, we'll just add the world_borders layer. We could save a step and just type in the layer name if we happen to know it.

Entering the layer name is just the start. GRASS can display a number of features from the layer, including the following:

- Shapes
- Categories
- Topology
- Line directions
- Points
- Lines
- Boundaries
- Centroids
- Areas
- Faces

This gives us a lot of options. Don't worry if you don't understand what all of them mean. For now we just want to produce a map showing the polygons that make up the countries of the world. To do this, all we really need to select is shapes and areas. To draw the layer now, switch to the map window, and find the Zoom to button located near the middle of the toolbar (it looks like a magnifying glass next to a map). Notice the tool is actually a drop-down tool button. Click and hold the button to display the options. Click Zoom to Default Region, and the world_borders map will be drawn to fill the display window. Notice we didn't set any options other than the layer name and what features to draw so the countries are drawn in the default color (gray). If we want to symbolize the map by population or some other attribute, it turns out you can't do that by adding a vector layer to the map—you have to use a thematic map layer.

Adding a thematic map layer is done in a similar fashion as a regular vector layer. In fact, the button to add a thematic layer is just to the right of the button we just used—if in doubt, hover the mouse. Clicking the Add thematic map layer button adds a "thematic 1" object to the manager window. Clicking it brings up the options panel. First we specify world_borders as the vector map. You will notice the options for a thematic map are different than for the vector map. The critical thing is we need a numeric field to classify the map with. Fortunately,

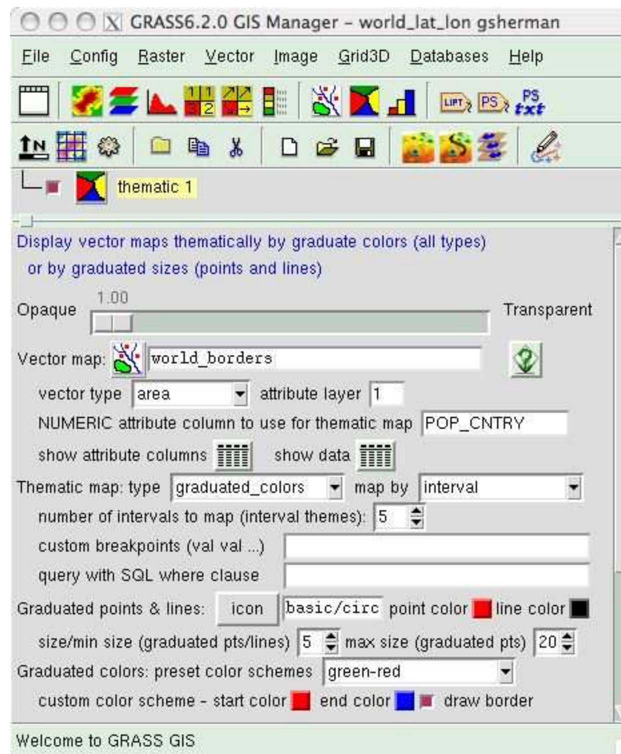


Figure C.8: Settings for a thematic map of the world by population

the population field is precisely that. If you can't remember the name of the fields in the layer, GRASS lets you list those using the Show Attribute Columns button. Clicking it causes the field names and types to be printed in the Output window. Doing that we quickly see that POP_CNTRY is what we want to use to classify the map so it gets entered in the textbox for the numeric attribute column.

Next we need to set the type of thematic map from the drop-down list. In the case of the population map, we want graduated colors. You can also select the Map By option, choosing interval, standard deviation, quartiles, or custom_breaks. Interval will produce the results we need. We set the number of intervals (classes) to 5 and choose a preset color scheme of Green-Red. That's all we have to do to set up the map. In Figure C.8, you can see the completed setup in the GIS Manager for the thematic map. There are more options you can't see in the figure,

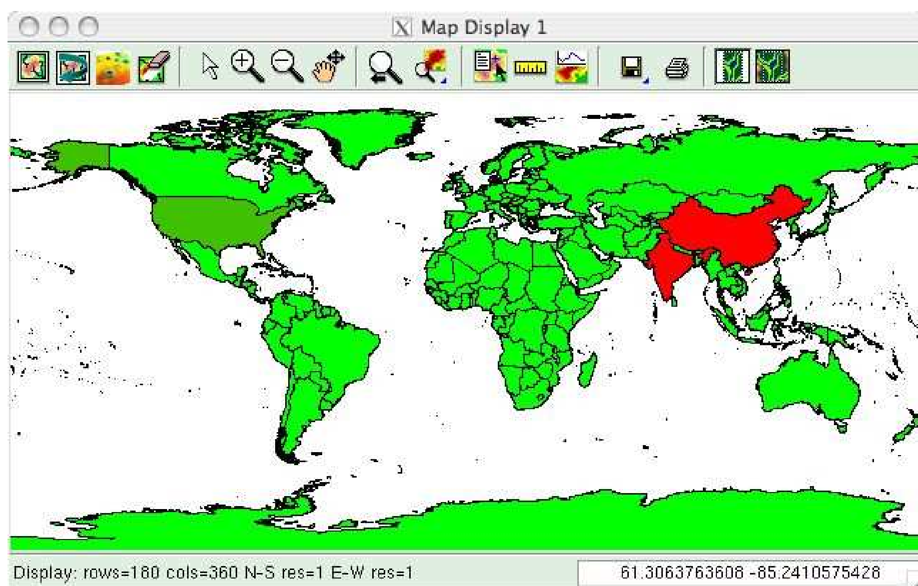


Figure C.9: Thematic map of population by country

such as font sizes for the legend and so forth, but the defaults are fine for generating the map (fire up GRASS and look if you want to see what they are). The thematic map can be drawn using the same method as for the regular vector map. As you can see in Figure C.9, the countries are rendered by population, with the most populous in red. The legend is displayed separately from the map window in GRASS and shows the colors and class intervals.

You've probably noticed that the GRASS GUI uses a different paradigm than the other OSGIS applications we surveyed in Appendix A, on page 269. As such, it takes some getting used to. If you are going to use GRASS, take some time to familiarize yourself with the interface. In fact, let's do some of that right now.

C.4 Getting to Know the GUI

One of the barriers to using GRASS is psychological. Many people dismiss it as being too difficult, arcane, or archaic. That's unfortunate because GRASS has a lot to offer. That's not to say there isn't a learning curve with GRASS. Just like any powerful application, it takes time to learn and master. That said, let's take a look at the map interface

MDI or SDI?

As you may have guessed by now, GRASS lets you have multiple windows, which essentially makes it a Multiple Document Interface (MDI) application as opposed to a Single Document Interface (SDI) application. Some folks prefer MDI, although modern user interface design thought seems to indicate that SDI is preferable. Using an MDI application on a small display can be quite painful—plan accordingly.

in Figure C.9, on the preceding page and examine both the toolbar and the other features.

To acquaint us with the tools available on the map interface, take a look at the following list of each tool and a brief description of its function. The name for each is the same tooltip text you see when you hover the mouse over a tool.

Display active layers
: This tool displays all the active layers using the current region extents (that is, view extent) for the map interface.

Redraw all layers
: Redraws the map, displaying all active layers in the current view extent.

Start NVIZ using active layers in current region
: This starts NVIZ, a tool for viewing data in three dimensions. NVIZ supports vertical exaggeration, rotation, and fly-through effects.

Erase to white
: Erases the map to a white background color.

Pointer
: Switches to the pointer. When the pointer is active, its location is displayed in the status bar at the bottom of the map.

Zoom In
: This provides an interactive zoom-in tool. Dragging a box zooms in to the outlined area. A single click of the tool zooms in a fixed amount.

Zoom Out
: This provides an interactive zoom-out tool that works like that

of QGIS. The display will be zoomed so that it is contained in the rectangle created by dragging the mouse. Clicking the map results in a fixed zoom out.

Pan
: Pans the map when you hold down the mouse and drag it.

Return to previous zoom
: This should be pretty obvious, but it returns you to the previous view. Unlike QGIS, this tool maintains a history of zoom levels, allowing you to click your way back to where you initially started.

Zoom to
: This tool is really a collection of one-shot zoom actions allowing you to zoom to the selected map, a saved region, the current region, and the default region. There are a couple of other choices as well.

Query
: This tool is equivalent to the identify tools we saw in uDig and QGIS. It operates on the active layer (map) and returns information about the layer at the location you click. For a raster this will be the coordinates of the mouse click and the cell value. For a vector layer, it displays the coordinates as well as the attributes for the feature.

Measure
: Measures distances on the map, using the current measurement units.

Create profile of raster map
: This button actually opens another window that allows you to draw a profile line on a raster and then plot a cross section. This is useful for example in creating elevation cross sections from a Digital Elevation Model (DEM).

Export display to graphics file
: Exports the current map view to a graphics file. PPM/PNM, TIFF, JPEG, and BMP are supported.

Print raster and vector maps to eps file
: This tool does more than the tooltip implies. It allows you to send the map to a printer or create a PostScript, PDF, or EPS file.

Constrain map to region geometry
: This causes the map display to be constrained to the actual region

(extent) you asked for in your zoom or other map navigation action. When using this mode, the map likely won't fill the display.

Map fills display window

This mode causes the map to fill the entire display area. This mode is the same as that used by uDig and QGIS.

C.5 Digitizing and Editing

To complete our introduction to GRASS, let's look at how to digitize and edit a layer. We'll create a layer and digitize some lakes off a world mosaic raster.

The GRASS Vector Model

Before we start digitizing, we need to go over a few facts about how GRASS stores and works with vector features. The GRASS vector model is all new at version 6.*x*. Rather than go into all the gory the details now, just be aware of the following:

- Each vector feature has a "category" number that serves as an identifier.
- Vector features can have 0..n category numbers.
- A GRASS layer consists of link(s) from a set of geographic objects to attribute table(s).
- Layers don't contain any geographic features but are linked to them.
- For a feature to be part of a layer, one or more of its category values must appear in the attribute table.

You're probably wondering what the story is with this confusing sounding category and attribute table thing. Basically, it's pretty powerful in that it allows you to link multiple attribute tables to a single feature. Why would we want to do that? Well, in a contrived example, suppose for our lakes layer we want to store physical characteristics such as volume and maximum depth, width, and length. We also want to store information about fish species and their percentage of the total fish population. As you can see, these two goals don't really translate into one attribute table. In the first case, it's a one-to-one relationship, and in the second (fish), it's a one-to-many relationship. Besides that, the two types of data just don't belong together. In GRASS, we can create two attribute tables and link them to our vector layer. Then we can add records to each, using the vector category as an identifier in the attribute tables. If a lake has no fish, it won't have any records in the

fish table. If we draw the "fish" layer, that lake won't show up. Although this may not be the best example, you get the idea.

In fact, you can combine totally unrelated features (datawise) that are topologically related in a single map and break it out into layers using this scheme. The GRASS documentation uses the example of forests and lakes, but any mapping of land parcels that includes water bodies is a good example. The lakes and parcels are topologically related (share common boundaries) but are different types of features. In conventional GIS, we might represent these with two separate layers (for example, shapefiles). In GRASS, they can live in the same map and be distinguished using categories and attributes.

Digitizing and Editing in GRASS

GRASS digitizing is a bit different from what you may have seen with other applications. First, of course, we need to create a new map in which to store our vector features. We can do this and start digitizing all in one step using v.digit. When you use v.digit from the GRASS shell, it fires up the digitizing tools and a map display window. But for this to work, we have to open a map monitor first using d.mon.[1] Instead of going that route, we'll just start up GRASS in GUI mode and let it do some of the behind-the-scenes work for us.

Once you have the GRASS GIS Manager up and running using the world latitude/longitude location, choosing Digitize from the Develop map item on the Vector main menu brings up the v.digit dialog box. This allows us to enter the name for the new layer and tick the box to allow creating it since it doesn't exist yet. In Figure C.10, on the next page, you can see the v.digit dialog box with the options we need filled in. You'll notice that we specified a display command of d.rast map=world_mosaic. This becomes the background image that we will digitize from. We could specify more than one background map if needed. The command d.rast simply tells GRASS to draw the raster specified by the map option, in this case our world mosaic.

You've perhaps noticed that the entire v.digit command required to accomplish the same result is displayed at the bottom of the dialog box, next to a little "copy" icon. This allows you to copy the command to the clipboard and use it elsewhere—for example in script. This is also

1. In GRASS 6.3 the use of d.mon is no longer required.

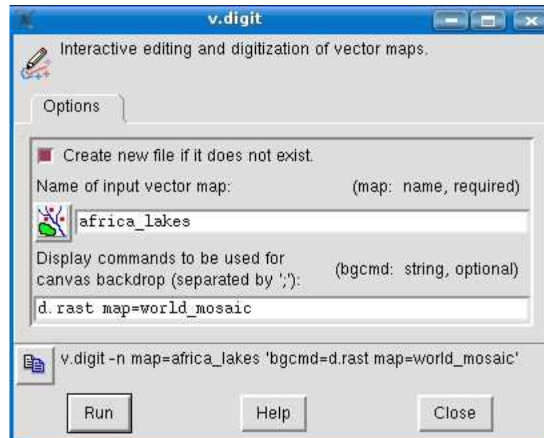


Figure C.10: Setting up to digitize a new map in GRASS

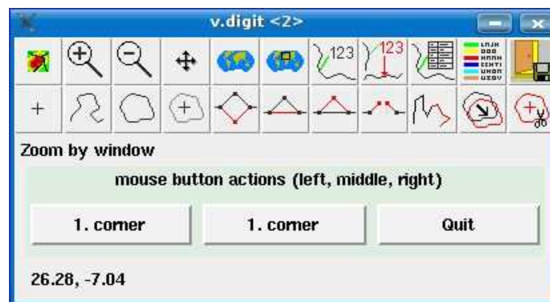


Figure C.11: GRASS digitizing tools

helpful from a learning standpoint so you can see what commands are used to accomplish a task in GRASS.

Once we click the Run button, two things happen. First, a Monitor window is opened, and the layer(s) we specified as background are drawn. Second, the v.digit tool window appears, as shown in Figure C.11. Together these two windows provide us with the capability to digitize features. The first step is to zoom into the area where we want to work. Let's start with Lake Tanganyika, located in the Great Rift Valley near 29.57 degrees N latitude, 6.16 degrees W longitude (the v.digit tools dialog box show latitude/longitude at the bottom left of the window).

We are almost ready to digitize, but first we need to add at least one more column to the attribute table associated with our vector map. To do this, click the Open settings tool in the v.digit toolbar. This brings up the settings dialog box where we can configure a number of things for our digitizing session, including the symbology, background map, and snapping thresholds, as well as the attribute table. To add a column to the attribute table, click the Table tab. Click the Add New Column button, enter the field name, and then choose the field type from the drop-down list. For the Name field for our lakes, we'll use a varchar field with a length of 24.

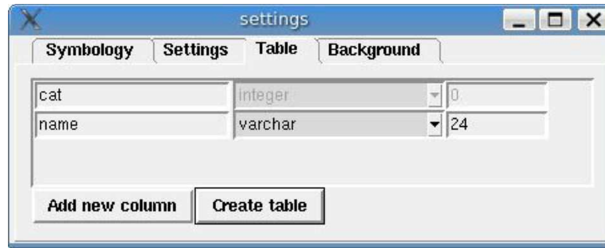


Figure C.12: Adding a Column to a GRASS Vector Map

In Figure C.12, you can see the settings filled out for the name field. If you wanted to add more columns, you can do so using the same process. To create the column(s), click the Create Table button, and then close the settings dialog box. Now we have a vector map and associated attribute table ready for digitizing.

Let's take a look at the tools available for digitizing (see Figure C.11, on page 324). The first four tools are for refreshing and navigating the map and by now should be pretty familiar. The first button just redraws the map. The others allow you to zoom in, out, and pan the map. I need to point out a key feature of this dialog box: the three wide buttons below the toolbars represent the buttons on your mouse (yes, you need a three-button mouse or some way to emulate one). When you select a tool, the labels on the buttons change to give you a visual cue as to the function of each mouse button. Get used to looking at the labels; it will help you as you digitize. If we click the Zoom in by window tool, we see that the mouse button functions are as follows:

- 1. corner
- 1. corner
- Quit

So, this means we can use buttons 1 or 2 (left or middle) to mark the first corner of our zoom window. When we click the map with either button, the button labels in the dialog box change yet again:

- 1. corner
- 2. corner
- Quit

If we click the left button, we reset the corner and start over. If we click the middle button, it establishes the second corner of our zoom window and the display changes, zooming in as we expect. In both cases, the third (right) button cancels the zoom function and resets all the button labels to blank.

This takes a bit of getting used to because you are probably accustomed to just dragging a rectangle, letting go, and watching the zoom happen. The shortcut in this case is to use the middle button for setting both corners of the zoom window—not that is saves you any effort, just a bit of finger movement for those of us that are uncoordinated.

Now we know how to navigate the map, and the tools we need to get started digitizing are on the second toolbar. The first two are for points and lines, and the next two are the ones we want.

First we select the Digitize new boundary tool and click it. The mouse button cues now indicate left button to create a new point and right button to quit. Now all we have to do is click our way around the lake, following the shoreline. When you click the first point, the mouse cues change again. The first remains New Point, but the second changes to Undo Last Point, and the third changes to Close Line. Now we can just click around the lake and use undo if we need to delete a bad point. Once we get to the origin, we use the right mouse button to close the boundary, clicking over the first point. If we click within the tolerance, the boundary will close; if not, we'll have to do it manually.

As soon as you right-click, a form pops up to allow you to enter the name. At this point you are naming the boundary line. If it's not important to you to put a name on the boundary, you can skip it; otherwise, you can enter the name of the lake now. Once we close the boundary, we'll see how to make it into a polygon.

If the boundary turns green when you close it, you are in luck. The start and end points coincide. If not and it's red, we can use the Move vertex tool to easily move the last point to close things up. To move it, select the tool and click the last point, and then move the mouse until the cursor is over the first point and click again. If you're lucky (or good), the border will turn red to green, indicating closure.

Now that we have a closed boundary, we can make it a polygon by adding a centroid. Click the Digitize new centroid tool, and then click somewhere in the center of the lake polygon. Again the attribute form

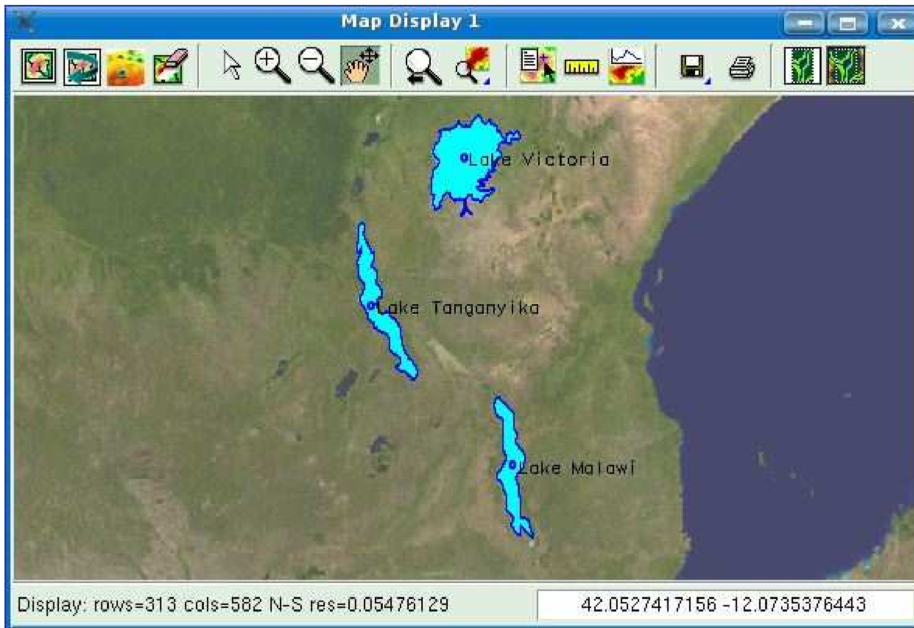


Figure C.13: Results of digitizing lakes in GRASS

will pop up, and we can enter Lake Tanganyika for the name. When we submit, the attributes are stored. Now when we click the centroid of the lake in the GRASS map display using the Query tool, we get the polygon attributes. In fact, now our africa_lakes map contains two feature types: 1_line and 1_polygon. We can draw boundaries to view the outline of the lake. If we also turn on areas in the GIS Manager, we see the polygon. In Figure C.13, you can see the results of our digitizing the lakes. We've rendered them in cyan with a three-pixel dark blue border and added labels to the centroids, all using the GIS Manager to set things up.

That completes our simple example of digitizing in GRASS. We didn't get too fancy with things but illustrated a fairly typical session for creating polygons. Of course, lines and points work in a similar fashion. The other tools on the v.digit dialog box allow us to add and delete a vertex, split a line, and move or delete an entire feature.

On the upper toolbar, you'll find tools to both display categories for a feature as well as copy them. Why would you want to copy categories from one feature to another? Well, imagine we have two fea-

tures that belong to the same overall "thing." An example is a parcel of land divided by a river. When you digitize, you will have two polygons, but they both are the same parcel of land—they have the same owner, tax ID, or what have you. We can digitize the first polygon and set its attributes. Then when we digitize the second, rather than assigning a new category number and duplicating all the attributes, we just copy them from the first parcel using the Copy categories tool. So, this tool is good when we are working with multipoint, multilinestring, or multipolygon features.

This section is about digitizing and editing with GRASS, and you're thinking "all we did was talk about digitizing." Well, as with most of the editing-enabled applications, the operations are pretty much the same. How do we edit the features in GRASS? Just start up v.digit from the GIS Manager or from the GRASS shell. We can also manipulate the attribute data using the db. suite of commands.

Appendix D

Quantum GIS Basics

Quantum GIS has a lot of functionality and many areas to explore. Here you will find an introduction to the basics of using QGIS, including map navigation and other essential features.

At first glance, QGIS has a feature set similar to other GIS viewers/editors. QGIS can view vector and raster data, including data stored in a PostGIS-enabled PostgreSQL database. QGIS also supports WMS layers and has preliminary support for WFS layers.

We used QGIS to view and render data in Chapter 3, *Working with Vector Data*, on page 37, so we won't repeat that here. Instead, we'll take a look at some of the features you may not have seen yet.

D.1 Vector Properties and Symbology Options

Each layer you add in QGIS has properties associated with it. You can access the vector properties dialog box by double-clicking the layer name in the legend or by right-clicking its name and choosing Properties from the pop-up menu.

Symbology

First let's look at the options on the Symbology tab:

Legend Type

QGIS supports a number of legend types. These are actually ways to symbolize your data. Currently you can choose from Single Symbol, Graduated Symbol, Continuous Color, and Unique Value renderers. For examples of using these to render your data, refer to Section 3.4, *Advanced Viewing and Rendering*, on page 45.

Transparency
: The transparency slider allows you to make the layer transparent. This allows you to "see through" the layer to reveal layers lower in the map stack.

Label
: The Label field allows you to specify the name that appears next to the symbol in the legend. So for the world_borders layer, we might name it "World Borders."

Point Symbol
: This drop-down contains the available symbols for a point. When you select one from the list, it will be used to render the points on the layer.

Point Size
: This allows you to change the size of the symbol. The size is specified in points just as font sizes are in your word processor.

Outline Style
: For line and polygon layers, you can specify the style of the line used to draw the feature or its outline. This also works with the basic point symbols (box, triangle, circle, and diamond). QGIS provides a number of style choices, including solid, dashed, dotted, and various combinations of dot-dash.

Outline Color
: You can specify the color of the outline for the basic point symbols as well as for line and polygon layers. Just click the colored box to the right of the option to open the color selector. The color selector is platform specific, so its appearance will depend on whether you are running Windows, OS X, or in the case of Linux, your Linux windows manager (for example, KDE vs. Gnome vs. XFCE, etc.).

Outline Width
: This option lets you specify the width of the line in pixels (used for line and polygon layers).

Fill Color
: For a polygon layer, you can specify the color used to fill the polygons or basic point symbols. Again, just click the colored box to the right of the option to open the color selector.

Fill Patterns

For polygon and basic point symbol fills, the default is a solid. QGIS has a range of other fill patterns you can choose from (including hollow). Just click the pattern you want to use to select it.

More Properties

Let's finish up looking at a few more features of the vector Layer Properties dialog box before moving on to more advanced consideration of QGIS. Some of the things we'll see here may not make too much sense yet, depending on what class of user your are. Don't worry, we'll explain as we go along and draw it all together.

General Tab

The General tab contains options and settings that are of all things—"general." First off we see that we can define a display name for the layer. We did this in the previous example by right-clicking the layer name and choosing Rename. That's the quick way to do it, but you can also set it here if you like.

Scale-Dependent Rendering

QGIS supports *scale-dependent rendering*. Scale-dependent rendering allows you to control when a layer is visible. There are a couple of primary reasons why would we would want to do that:

- The layer is meaningless at small scales (remember, a small scale covers a large area) because you wouldn't be able to see the information. An example is displaying streets at the scale of our world_borders layer. You wouldn't gain much information from such a display.
- You have two versions of the data, one for small scales and one for large. An example of this is a coastline. At a small scale (zoomed way out), we don't need to display a lot of detail. In fact, such detail is wasted and slows only the rendering process. You can cram only so much information into a single pixel. As you zoom in, you want the low resolution layer to switch off and another higher resolution (more detail) layer to switch on.

If you choose to use scale-dependent rendering, you have to set the minimum and maximum scales for the layer. This controls the visibility of the layer. The scale for each setting is specified as a ratio of 1:[some

number]. How do you determine what that number should be? Fortunately, QGIS displays the current scale in map units on the status bar. If you look back to Figure 3.8, on page 49, you will see that the scale is 1:464 (look at the bottom of the QGIS frame, right under the map display). You can use this information to determine what scales you want for the minimum and maximum settings. Use the zoom tools to get the view you want at each level and then use the displayed scale to set up the scale-dependent rendering.

What if we want the layer to always be visible when zoomed out (small scales) but turn off as we zoom in? Set the scale you want it to turn off at in the Minimum field, and set the Maximum field to a very large number. If you want the inverse to be true, set the Minimum field to zero and the Maximum field to the scale at which you want the layer to turn off.

Note that the terms *minimum* and *maximum* as used in the dialog box seem to be opposite of our definition of small vs. large scale. As long as you are aware of this, you'll be able to set things up to meet your needs.

Spatial Indexes

The next area of interest on the General tab is the Spatial Index section. From here you can create a spatial index, assuming the layer type supports such an option. A spatial index speeds up drawing, selecting, and identifying features. For example, when zooming in, QGIS uses the spatial index to select only the features in the view window for drawing. This is much quicker than marching through each feature in the layer and testing it to see whether it should be drawn. The same holds true for selecting or identifying (which really involves a select of sorts) features. The spatial index helps quickly locate the feature(s). When you click the button to create a spatial index for a shapefile, a new file with a .qix extension is created. For our world_borders layer, the spatial index file is world_borders.qix. Always make sure to create a spatial index for layers that support it. It makes things much snappier.

Spatial Reference System

This is just a fancy name for projection. The *spatial reference system* defines the projection and coordinate system of the layer. This is what makes it possible to draw data in real-world coordinates and have differing layer's "line up." The projection information for the layer is displayed in PROJ.4 format (see Chapter 9, *Projections and Coordinate*

Systems, on page 138 for details). Using the Change button, you can set the projection for the layer.

Usually QGIS sets the parameters you see here based on the projection information associated with the layer. If there is no projection information, then QGIS makes an assumption based on your preference setting for projections. You can set QGIS to do the following when a layer is loaded that has no projection information:

- Prompt for projection
- Use the projectwide default projection as set in the Project Properties
- Use the global default projection specified in your user preferences

If you want to take a look at these options now, open the Preferences dialog box, and click the Projection tab.

Subset

The last piece of the General tab concerns itself with the subset of the data. This feature lets you create a subset of your layer on the fly and display only the data matching the criteria you specify. Currently this works only with PostGIS layers. For an example of how to use this feature, take a look at Section 7.4, *Using PostGIS and Quantum GIS*, on page 110.

Metadata Tab

The Metadata tab provides detailed information about the layer, including its location, type, number of features, and projection. In Figure D.1, on the next page, we can see the Metadata tab contents for our world_borders layer.

The General information provided shows us not only information about the layers physical location but also the type—in this case an ESRI shapefile. We can also see that our layer contains polygon features and has a number of editing capabilities. Well, these aren't really capabilities of the layer but show you what QGIS can do with the layer in terms of editing. With the world_borders layer, we see that we can add and delete features, create a spatial index, and change the values for attributes in the layer. This is useful information, especially in the case where you just downloaded a new layer and it's still a mystery to you.

The metadata also includes information about the extent of the layer. It shows the extent in both the coordinate system (spatial reference system) of the layer and the project. You might notice that they are the

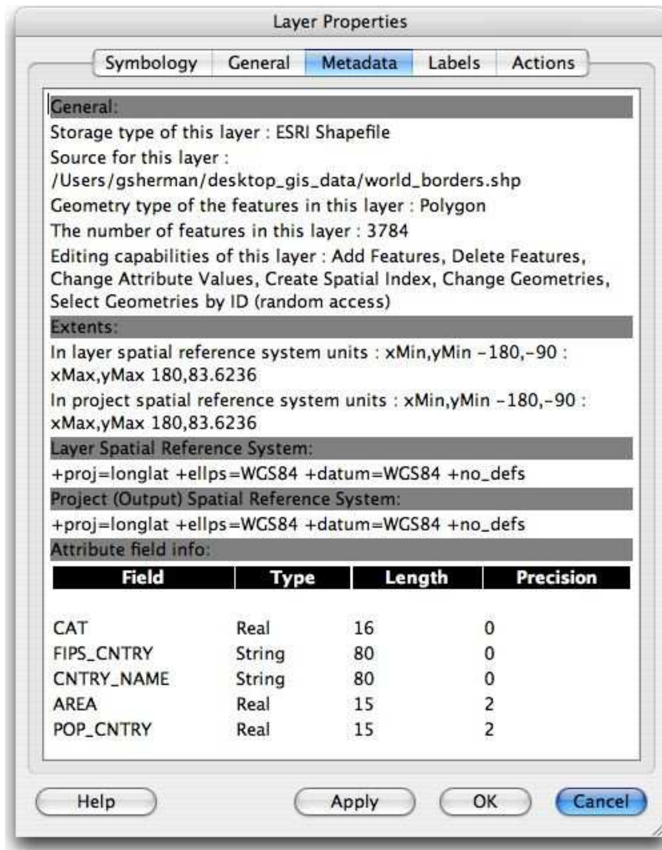


Figure D.1: Metadata for the world_borders layer

same in our example. This is because we haven't changed the coordinate system of the map canvas. We are displaying the world_borders features in the same coordinate system in which they are stored.

The layer and project coordinate systems are displayed in the next section of the metadata, using PROJ.4 format. In this case, it's +proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs. QGIS isn't so friendly in describing the projection in terms that we can easily understand. If you are familiar with GIS and coordinate system, you can probably guess that the coordinates are latitude and longitude in the WGS84 datum. Refer to Chapter 9, *Projections and Coordinate Systems*, on page 138 for hints on how to decipher what projection you're working with.

The last bit of information we can glean from the metadata is information about the attributes of the layer. If you look at Attribute Field Info in Figure D.1, on the previous page, you will see it lists all the fields in the layer, along with information about the type, length, and precision. In the case of world_borders, we have three numeric fields (real) and two string fields. This gives us a quick overview of the attributes for the layer and also the field names we can use in labeling and identifying features.

You might be wondering about the remaining tabs on the vector Layer Properties dialog box—Labels and Actions. QGIS has a lot of options for labeling features. You can set the font, shading, alignment, and other options to control how features are labeled on your map. You can even use fields in the layers attribute table to control the style, alignment, and buffer (shading) for the labels. Attribute actions are handy things that allow you to call an external program and pass it values from the layer's attribute table. You can get an in-depth look at them in Section 3.5, *Using Attribute Actions*, on page 64.

D.2 Project Properties

You can customize a number of properties for a QGIS project. For example, QGIS allows you to set a background color for the map canvas. Often, this can be used to improve the appearance of the map, especially when you have large areas of whitespace. To set a background color, choose Project Properties from the Settings menu. To set the color, activate the General tab, and then click the color box to the right of the Background Color label.

D.3 Map Navigation and Bookmarks

QGIS supports the typical map navigation toolset with zoom and pan. We can also create spatial bookmarks that allow us to store a view extent and return to it later.

Zooming

Load a layer, click the Zoom In tool (looks like a magnifying glass with a plus sign in the middle), and then drag a rectangle on the map canvas to zoom in. If you can't find the zoom tool, hover the mouse over the tools in the toolbar, and check out the tooltip help that pops up for each one. You can also zoom in by a fixed amount by just clicking the

map canvas. Zooming out works the same way but with one nuance. When you drag a rectangle to zoom out, the current view will be scaled to fit in the rectangle.

You can also use the mouse wheel to zoom in and out on the map. First, make sure the mouse cursor is over the map canvas. To zoom in, roll the mouse away from you (think of it as flying toward the map). To zoom out, roll it toward you (flying away from the map). The focal point of the zoom is the location of the mouse cursor. After a mouse zoom, the map will be centered at the cursor location. You can change this behavior on the Map Tools tab of the Options dialog box. If you like, you can set it to just zoom in and not re-center the map. You can also set the zoom factor used with each "roll" of the wheel. Using the mouse wheel to zoom can be very useful when digitizing data.

We need to mention a several other zoom tools here:

Zoom Full
: This tool zooms to the full extent of the layers on the map canvas. This means you will see all the features in all the layers on the map. This is a quick way to reset the view after you have zoomed in and panned around.

Zoom to Layer
: This zooms to the extent of the currently selected a layer. The selected layer is the one highlighted in the legend. You can invoke this by using the tool button or by right-clicking the legend and choosing Zoom to layer extent from the pop-up menu.

Zoom to Selection
: In QGIS (as in most GIS applications), you can select features in a layer either interactively on the map using a selection tool or from the attribute table. If you have a selection set, you can zoom to just the area it covers using this tool.

Zoom Previous
: This is like a back button. It zooms to the previous view extent. If you hit the button again, it will take you to where you were before. In other words, it's a one-shot deal. There is no zoom history that you can navigate forward and back through. But it's still a quick way to back out of a zoom or pan operation.

Panning

Panning is done using the Pan tool on the toolbar. It looks like a hand and is located in the same toolbar as the zoom tools. Again, use the mouse hover and tooltips to help determine the function of any tool on the toolbars. To pan, select the tool, and drag it on the map canvas to move the view.

You can also pan using the mouse and without clicking the map canvas. This works only if the map canvas has the focus, meaning you haven't just worked with some other component of the interface, such as the legend. To pan, hold down the spacebar, and move the mouse. This method of panning is very useful when digitizing data, since switching to the Pan tool may have undesired consequences.

Spatial Bookmarks

Bookmarks are everywhere—in your web browser, text editors, and even books. QGIS supports spatial bookmarks, a way to save a location and return to it later. Spatial bookmarks are stored globally, meaning they are available in QGIS regardless if you have the same layers loaded as when the bookmark was created.

To create a bookmark for an area you are viewing, simply click the New Bookmark tool (its the half-globe with star above it), or choose New Bookmark from the View menu. This opens a simple one-line dialog box that allows you to enter a bookmark name. Click OK, and the bookmark is created. Make sure the name you create is descriptive. You might also want to append the coordinate system to the name as a reminder. The bookmark tool doesn't store the projection (it just stores coordinates), so it's up to you to remember or annotate it if it's important.

We can manage our bookmarks using the Geospatial Bookmarks dialog box shown in Figure D.2, on the next page. Note we have four bookmarks stored. The name and coordinates (Extent) are visible. Since bookmarks are global, QGIS doesn't currently store anything in the Project column. To zoom to a bookmark, select it from the list, and click the Zoom To button. You can also zoom to a bookmark by double-clicking it in the list.

You can't do much in the way of managing bookmarks, other than delete them. There is no way to rename a bookmark or edit the extents. Be sure to take a look at the context help (click the Help button) for more information.

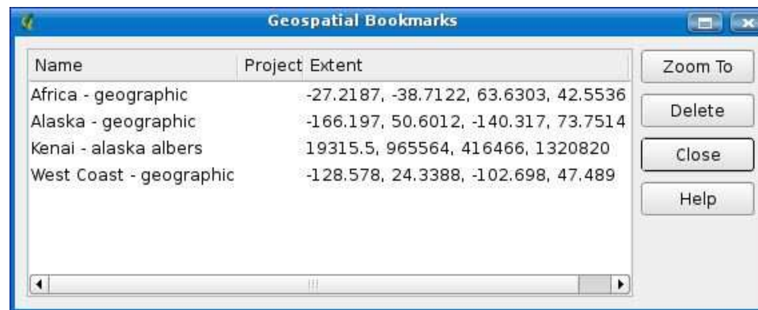


Figure D.2: QGIS Geospatial Bookmarks dialog box

D.4 Plugins

As you may have gathered already, QGIS supports the use of plugins to add new capabilities and tools. Basically, a plugin is a loadable module that can be added and removed from QGIS. With the release of version 0.9, QGIS supports writing plugins in both C++ and Python.

Let's get a brief description of each of the twelve plugins distributed with QGIS.

Add Delimited Text Layer
: Loads and displays data from a delimited text file. The text file must contain x and y coordinates for each feature. Only point data is supported by the plugin. You can find an example of using this plugin in Section 8.2, *Importing Data with QGIS*, on page 123.

Copyright Label
: Displays copyright information on the map canvas. You can customize the text, style, and placement of the label.

GPS Tools
: Tools to load and import GPS data, as well as download to your GPS unit. For details on using this plugin, see Section 8.4, *Using GPS Data with QGIS*, on page 130.

GRASS
: A full suite of GRASS tools for loading vector and raster maps, digitizing features, and using modules to import, export, and process data. We cover this plugin in a fair bit of detail in Chapter 12, *Getting the Most Out of QGIS and GRASS Integration*, on page 208.

Georeferencer
: Georeference rasters by interactively setting control points to create a world file. See Section 8.5, *Georeferencing with QGIS*, on page 136.

Graticule Creator
: Create a graticule (grid) shapefile by specifying the extent and interval between latitude and longitude lines. You can create a point, line, or polygon graticule with this plugin.

Launcher
: Launches a program or script from QGIS and captures the output. Commands are stored in a drop-down list for future use.

North Arrow
: Displays a customizable north arrow on the map canvas. You can adjust the placement of the arrow, as well as the angle. Or you can just let QGIS determine the angle direction automatically.

PostgreSQL Geoprocessing
: Tools for processing PostGIS layers. At present, this plugin is rather limited and contains only a Buffer tool.

SPIT
: The Shapefile to PostGIS Import Tool (SPIT) allows you to import shapefiles into PostGIS. The use of SPIT is covered in Section 7.3, *Using QGIS to Load Data*, on page 107.

Scale Bar
: Displays a scale bar on the map canvas. You can customize the placement, style, color, and size of the scale bar.

WFS
: Experimental plugin for consuming WFS services on the Internet and displaying the data in QGIS. This plugin is included as part of the QGIS package, but at the time of this writing probably isn't ready for prime time.

QGIS plugins are managed by the Plugin Manager. You access the Plugin Manager from the Plugins menu. In Figure D.3, on the next page, you can see the Plugin Manager and the twelve plugins that are distributed with QGIS.

The Plugin Manager is easy to use. Basically, you just check the plugins you want loaded and click the OK button. If you want to unload a

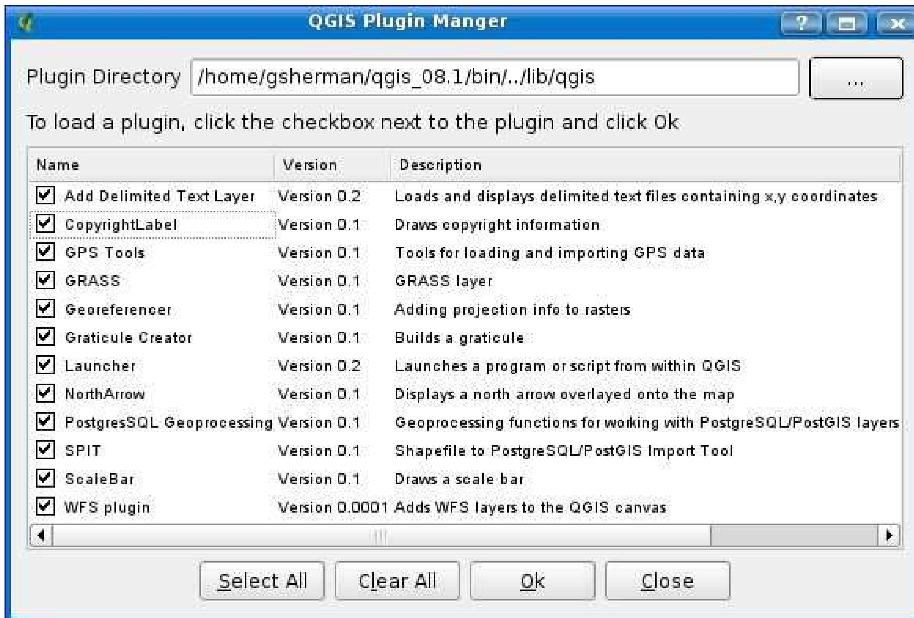


Figure D.3: QGIS Plugin Manager

plugin, uncheck it before you click OK. That's pretty much it when it comes to managing plugins. You probably noticed the Plugin Directory text box at the top of the manager dialog box. This allows you to specify a different directory to be searched for QGIS plugins. In practice, it's best to leave the directory alone, since by default it points to the location where your plugins are installed. If you were developing a plugin, you might find it necessary to change the directory so you could load your new creation for testing purposes.

When you load a plugin, the plugin's menu and toolbar icons are added to the QGIS GUI. With one exception all plugin icons are added to the Plugins toolbar. The exception is the GRASS plugin. It has ten tools and therefore its own toolbar. Once you have loaded one or more plugins, you'll find them listed under the Plugins menu on the QGIS main menu. You can access a plugin's functions from either the menu or the appropriate tool on the toolbar.

Creating a Plugin

Creating a plugin in QGIS is a bit of a technical affair. Currently, it requires the use of C++ or Python and Qt. Full support for writing plugins in Python is available in version 0.9. If you really want to write a C++ plugin, check out the resources available on the QGIS website, blog, and wiki. For an example of a QGIS plugin written in Python, see the handy Zoom To Point plugin we created in Section 13.2, *A PyQGIS Plugin*, on page 239.

With the advent of the Python bindings for QGIS, a number of Python plugins have already emerged. For more information, check out the QGIS Python Plugin repository.[1]

1. http://spatialserver.net/pyqgis_repository.html

Index

A

Actions, attribute, 66f, 64–66
Alaska Albers Equal Area Conic projection, 142
Algebra, 233f, 231–233
Alignment problems, 142
American Society for Photogrammetry & Remote Sensing, 148
Analyze, 18–20
  see also Geoprocessing
Applications, writing, 263–268
Attribute data, 44–45, 46f, 56–66
  editing, 89–90
  features and, 84f
Attribute tables, quick search of, 60–62
Automating tasks, *see* Scripting
Azimuthal projection, 139

B

Beach Lake example, 84
Beaudette, Dylan, 186
Bindings, 235
Bird sightings example, 16f, 55, 56f
Bleeding-Edge Syndrome (BES), 30
Borders, 49f
Browser tab, 219–221
Buffering, 19
  GRASS and, 223f, 224f, 222–224
  lake example, 19f
  tools and, 81

C

Clipping feature, 167f, 171f, 172f, 173f, 166–173
Collar, 68
Collars, 168
Command-line interface (CLI), 235, 236
Command-line tools, 174–207
  GDAL and OGR, 186–201
    data conversion, 197f, 199f, 190–201
    raster information, 189–190
    vector information, 186–189
  generic mapping tools, 175f, 178f, 174–186
  PostGIS and, 203–207
  spatial index for shapefiles, 202–203
Conic projection, 139
Continuous colors (QGIS), 51f, 50–51
Contour maps, 229f, 230f, 228–231
Control points, 136
Converting, data, 96–97, 128–130, 190–201, 249
Coordinate system, 138–148
  changing, 198
  control points and, 136
  conversion of, 194
  datum, 140
  generic mapping tools and, 175
  projections and, 139–145
  raster data, 69
  transforming, 259–262
Costs, platform and, 26
Cygwin, 236
Cylindrical projection, 139

D

Daley Bay Quandrangle, Minnesota, 82
Data
  classification of, 46–47
  conversion of, 96–97, 190–201, 249
  converting, 128–130
  datum and, 140
  as drug, 93n
  formats of, 91–97
  freeing of, 34
  generic mapping tool, 179

GPS, 131f, 133f, 130–135
images, georeferenced, 137f, 136–137
importing, 124f, 128f, 122–128
management, 93
projection problems, 142–145
QGIS and GRASS integration and, 209–211
safety of, 86
sample dataset, 37
sources of, 34–35
symbolizing, 15
viewing, 37–42, 67–73
web deliverable, 93
see also Digitizing; Geoprocessing
Data Definition Language (DDL), 258
Data types, 17
Databases
Post-GIS enabled, 102–105
tables vs layers, 114
Databases, spatial, 94, 98–119
features of, 98
index, 105–106
open source, 99–101
PostGIS and, 101–110
PostGIS and QGIS, 111f, 113f, 115f, 116f, 110–117
PostGIS and uDIG, 118f, 118–119
spatial query, 99
structure of, 98
Datum, 140
Desktop mapping
described, 13–21
GIS functions, 14f
vs. server mapping, 20–21
Digital elevation model (DEM), 76–77, 153, 154
contour maps and, 229, 230f
hillshades, 161f, 163f, 159–165
hydrologic model, 156–159
merging, 166f, 164–166
units, converting, 233f
Digital raster graphic (DRG)
clipping, 171f, 166–173
collar overlap, 167f
Digitize, 16–18
Digitizing, vector data, 82f, 83f, 84f, 85f, 81–89, 120–122
mistakes, 88f, 89f, 85–89
safety of, 86
subdivision plat, 121f
tools for, 81
Distortion, 138

E

E00 interchange files, 129–130
Eagle nest example, 222, 223f
Earthquake data (Alaska), 125, 128f, 259, 260f
Editing
attribute data, 89–90
with QGIS, 211–218
Editing, with QGIS, 212, 213f
Encapsulated PostScript (EPS) output, 174
EPSG notation, 141, 143, 144
European Petroleum Survey Group (EPSG), 195, 198

F

Formats, data, 91–97
conversion of, 96–97
raster, 92
selecting, 93–95
standardizing, 93
vector, 91
web deliverable, 93
Functionality, 94
FWTools, 44, 70, 202, 251

G

GDAL and OGR, 186–201
data conversion, 197f, 199f, 190–201
documentation, 190
driver support, 187
raster information, 189–190
scripting, 249–255
vector information, 186–189
GDAL library, 74
gdaladdo, 75
Generic mapping tools (GMT), 174–186
-B switch, 183
coordinate systems supported by, 175
data sources, 179
Earth hemisphere view, 175f
EPS output and, 174
fill colors, 182
flat example, 180–182, 183f
Globe centered, 178f
grid lines, 181
multiple commands and, 184

overlay, 183
parameters and, 177
printing maps, 234
projection, 181
pscoast command, 176, 178
QGIS users plotted, 185f
rivers, 181
scale bar, 182
Geocaching, 130
Geodata.gov, 21, 35
Geographic coordinates (GEOGCS), 141, 144
Geographic information system (GIS)
analyze, 18–20
data types, 17, 91–97
defined, 12
desktop vs. server mapping, 20–21
digitize, 16–18
as drug, 93n
functions, 14f
options and, 22
requirements, 23–29, 97
software for, 27–28
support for, 32–33
toolkit for, 21, 26
viewing data, 37–42, 67–73
visualize, 14–15
writing applications, 263–268
Geographic Markup Language (GML), 191
Geographic Names Information System (GNIS), 114
geometry_columns table, 104–105
Geoprocessing, 149–173
clipping feature, 167f, 171f, 172f, 173f, 166–173
data, projecting, 150–153
defined, 149
DEMs, merging, 166f, 164–166
hillshades, creating, 161f, 163f, 159–165
hydrologic modeling, 159f, 156–159
line-of-sight analysis, 153–156, 157f
Georeferenced TIFF (GeoTIFF), 69
Georeferencer plugin, 136
GeoRuby gem, 255, 257
GMane, 32
Google Earth, 22
GPS data
downloading, 132–133
loading and viewing, 133–134
QGIS and, 131f, 133f, 130–135
uploading, 135f, 134–135
GPS exchange format, 132
GPS plugin, 131, 135
gpsbabel, 130, 134, 135
GPX, 132
Graduated symbols (QGIS), 53f, 52–53
GRASS
browser, activating, 219–221
buffers and, 223f, 224f, 222–224
capabilities of, 95
clipping feature, 167f, 171f, 172f, 173f, 166–173
contour maps, 229f, 230f, 233f, 228–233
conversion, 96, 97
coordinate systems, changing, 150
deleting maps, 221
DEMS, merging, 166f, 164–166
E00 interchange files and, 129–130
FAQ, 137
geoprocessing and, 149
georeferencing, 137
hillshades, creating, 161f, 163f, 159–165
hydrologic modeling, 159f, 156–159
importing with, 126–127
line-of-sight analysis, 153–156, 157f
maps, creating new, 216f, 215–216
mapset, opening in, 209
multiple layers in, 214
QGIS integration, 208–234
attributes, adding, 213f, 217f
browser, 220f
city map with water wells, 218f
data, loading and viewing, 209–211
editing and, 212f, 211–218
mapset, opening in, 209f
settings, customizing, 217–218
tools, 210f, 218–233
raster data, importing, 211
rasters, exporting, 165
saving in, 213
scripting and, 235–236
vector overlays, 226f, 227f, 228f, 225–228
wiki, 234
Windows support, 236n
see also Appendix C
GTOPO30 DEMs, 164

H

Hillshades, 161f, 163f, 159–165
Hydrologic modeling, 159f, 156–159

I

Images
georeferencing, 137f, 136–137
warping, 199
Importing, data, 124f, 128f, 122–128
The Impossible Map (film), 148
Index, spatial, 202–203
Installation, *see* Appendix B
Integer primary key, 259
Integration, 29
Intelligent rasters, 78f, 79f, 76–80

K

KML, coordinates for, 194

L

Lake buffer, 19f
Large scale, 42
Latitude, 62
Layers
editing in GRASS, 212
multiple, creating, 214
vs tables, 114
Libre Map Project, 15
Line-of-sight analysis, 153–156, 157f
LiveCDs, 29
Logging example, 225, 226, 227f
Longitude, 62

M

Map algebra, 233f, 231–233
Mapnik, 264
MapServer, 202
MapWinGIS, 264
Minimum bounding rectangles (MBRs), 100f, 101
MSYS, 236
MySQL, 99, 100

N

Nabble, 32
NASA Visible Earth, 71, 71n
National Elevation Dataset (NED), 153
National Film Board of Canada, 148

O

Open Source Geospatial Consortium (OSGeo), 93
Open Source Geospatial Foundation (OSGeo), 33
OpenOceanMap, 266, 268f

P

Performance, spatial indexes and, 204
Platforms, 25–26
Population, 43, 44f, 63
PostGIS, 97, 101–110
cities features, 194
command-line tools and, 203–207
commands, 103
coordinates, transforming, 259–262
data loading, 192
data unloading, 192
database, 102
documentation for, 110n
gems, 255
limiting features, 114
loading data, 106, 260f
vs MySQL, 100f, 99–101
OGR and connection strings, 188
QGIS and, 111f, 113f, 115f, 116f, 110–117
scripting and, 255–262
shapefiles, exporting, 206
shapefiles, importing, 203
shapefiles, loading, 108f
spatial reference ID and, 204
templates, 103
uDig and, 118f, 118–119
PostgreSQL/PostGIS, 99, 100
Preprocessing data, 125–126
Primary key, 259
Printing maps, 234
"Producing Press-Ready Maps with GRASS and GMT" (Beaudette), 186
PROJ.4 Projections Library, 145–148
Projected layers (PROJCS), 142
Projections, 138–148
described, 138
determining, 141–142
EPSG notation, 143
focus of, 138
geoprocessing, 150–153
problems, data, 142–145
PROJ.4 library, 145–148

resources for, 148
types, 139–140
utilities for, 145
pscoast command, 176, 178
Public Geo Data effort, 34
PyQt website, 236n
Pyramids (raster data), 74f, 74–75
Python, scripting, 236–248
console, 238f, 237–239
PyQGIS plugin, 242f, 239–247, 248f
resources, defining, 241
shapefile, volcanoes, 255f
PyWPS, 21, 265

Q

QGIS, 46–47
attribute data and, 56–66
attribute actions, 66f, 64–66
attribute table, 60f, 63f, 59–64
identifying features, 57–58
selecting features, 59
attributes, entering, 84f
blog, 248
continuous colors, 51f, 50–51
data, importing, 124f, 123–125
data, loading, 107
delimited text plugin, 124f
digitizing, 82, 85f
earthquakes in, 128f
generic mapping tools and, 185f
geometry collections in, 214
georeferencing, 137f, 136–137
GPS data and, 131f, 133f, 130–135
GPS plugin, 131
graduated symbols, 53f, 52–53
GRASS integration, 208–234
browser, 220f
data, loading and viewing, 209–211
editing and, 212f, 213f, 211–218
mapset in, 209f
tools, 210f, 218–233
identifying features, 59f
layers in, 83f
libraries, 265
NASA World Mosaic, 71f
options, 48–50
plugins for, 208
PostGIS and, 111f, 113f, 115f, 116f, 110–117
pyramids and, 73, 74f
Python, scripting in, 236–248
raster properties, 72f
rasters, loading, 68f, 68
sample data loaded, 47f
shapefiles in, 82f
SQL and, 62
tolerance, 87
unique values, 55f, 54–55, 56–58f
vector layer properties, 48f
world borders layer, 49f
see also Appendix D
Qt designer, 242f, 241–243
Quantum GIS, *see* QGIS
Quantum Navigator, 265, 267f

R

Raster data, 17, 67–80
collar, 68
conversion of, 196
coordinate system for, 69
digital elevation models, 76–77
exporting from GRASS, 165
formats of, 92
GDAL and, 189
GDAL/OGR and, 189–190
GRASS, importing into, 211
intelligent, 78f, 79f, 76–80
properties, 72f, 72
pyramids, 74f, 74–75
silver grid, 77, 78f
transparency and, 200
transparency model, 73f
viewing, 68f, 67–73
Raster properties, 72f, 72
Refractions research, 118
Relief maps, shaded, 161f, 163f, 159–165
Requirements, 23–29, 97

S

Safety, of data, 86
Scale, 42
Scaling, 75
Scripting, 235–262
data transformation, 262
GDAL and OGR, 249–255
GRASS and, 235–236
PostGIS and, 255–262
Python and, 236–248
console, 238f, 237–239

PyQGIS plugin, 242f, 239–247, 248f
Searching, attribute tables, 63f, 60–64
Server vs. desktop mapping, 20–21
Shaded relief maps, 161f, 163f, 159–165
Shapefiles
conversion of, 250
creating, from delimited text, 255f, 251–255
described, 38
loading, 108f
PostGIS, exporting, 206
PostGIS, importing into, 203
in QGIS, 82, 83f
spatial index for, 202–203
vector data, getting, 186
volcanoes and, 193
Shell, 236, 249
Small scale, 42
Software, 27–28
change and, 30–31
support for, 32–33
see also Appendix A
Spatial databases, 94, 98–119
features of, 98
index, 105–106
open source, 99–101
PostGIS and, 101–110
PostGIS and QGIS, 111f, 113f, 115f, 116f, 110–117
PostGIS and uDig, 118f, 118–119
spatial query and, 99
structure of, 98
Spatial index, 105–106, 202–203
Spatial queries, 99, 109–110
Spatial reference ID (SRID), 204
Spatial Reference website, 146
Spatial view, creating, 117
SPIT plugin, 107
SQL, 104
data loading and, 107
QGIS and, 62
virtual layers, 117
Support, 32–33
Symbolizing data, 15, 43f, 44f, 42–44

T

Tables vs layers, 114
Templates, PostGIS and, 103
TIFF, georeferenced, 69
Tolerance, 87
Toolkit, 21, 26
for analysis and conversion, 218–233
custom applications, 264–265
digitizing, 81
for GRASS in QGIS, 210f
see also Command-line tools
TopoGrafix, 132
Transparency, 200
Trolltech, 242

U

uDig, 39, 265, 267
attribute data, 44–45, 46f
Data Sources dialog box, 40f
displaying world borders, 41f
navigation, 40–42
PostGIS and, 118f, 118–119
symbolizing data, 43f, 44f, 42–44
Unique values (QGIS), 55f, 54–55, 56–58f
U.S. Geological Survey (USGS)
datasets, 153
Digital Raster Graphic (DRG) topographic maps, 34
Map Projections: A Working Manual, 148
projections poster, 148
User classes, 23–25, 27f
Utilities, projections, 145

V

Vector data, 17, 37–66
attribute data, 44–45, 46f, 56–66
converting, 190
defined, 18
digitizing, 82f, 83f, 84f, 85f, 81–89
editing, 89–90
formats of, 91
GDAL/OGR and, 186–189
QGIS and, 46–47, 48f
QGIS appearance options, 48–50
QGIS continuous colors, 51f, 50–51
QGIS graduated symbols, 53f, 52–53
QGIS unique values, 55f, 54–55, 56–58f
shapefiles and, 38
symbolizing data, 43f, 44f, 42–44
viewing it, 37–42
Vector overlays, 226f, 227f, 228f, 225–228

Viewers, 38–39
Visualize, 14–15
Volcano location database search, 122

W

W*S service, 93
Warping images, 199
Web deliverable data, 93
Websites
  Alaskan geologic maps, 55n
  American Society for Photogrammetry & Remote Sensing, 148n
  Cygwin, 236n
  Daley Bay Quandrangle, 82n
  DEM Anchorage, Alaska, 154n
  Desktop GIS Book, 35n, 37n
  Earthquake data (Alaska), 125
  EPSG documentation, 143n
  European Petroleum Survey Group (EPSG), 195n
  Federal Geographic Data Committee's clearinghouse network, 35n
  FWTools, 202n, 251n
  GDAL documentation, 190n
  gdaladdo, 75n
  geocaching, 130n
  Geodata, 21n, 35n
  Geographic Markup Language (GML), 191n
  Geographic Names Information System (GNIS), 114n
  GIS LiveCDs, 29n
  GMane, 32n
  GMT, using with GRASS, 186n
  gpsbabel, 130n
  GRASS FAQ, 137n
  GRASS wiki, 234n
  GRASS Windows support, 236n
  GTOPO30 DEMs, 164n
  Impossible Map (movie), 148n
  KML coordinates, 194n
  Libre Map Project, 15n
  Mapnik, 264n
  MapServer, 202n
  MapWinGIS, 264n
  MSYS, 236n
  Nabble, 32n
  NASA Visible Earth, 71n
  National Elevation Dataset (NED), 153n
  Open Source Geospatial Consortium (OSGeo), 93n
  Open Source Geospatial Foundation (OSGeo), 33n
  OpenGIS Simple Features Specification for SQL, 104n
  OpenOceanMap, 266n
  PostGIS, 101n
  PostGIS documentation, 110n
  Public Geo Data effort, 34n
  PyQt, 236n
  PyWPS, 21n, 264n
  QGIS API documentation, 237n
  QGIS blog, 248n
  QGIS libraries, 265n
  QgisInterface, 238n
  refractions research, 118n
  Ruby, 255
  SDTS format, 92n
  Spatial Reference, 146n, 195n
  spatially enabled databases, 99n
  TopoGrafix, 132n
  Trolltech, 242n
  uDig, 265n
  USGS datasets, 153n
  USGS Map Projections, 148n
  USGS Maps, Archive of, 67n
  Volcano location database search, 122n
Well-Known Text (WKT), 107, 141, 257
Writing applications, 263–268

X

X and Y coordinates, 62

Z

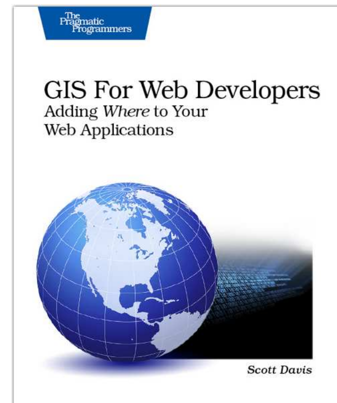
Zoom, 243–247

More GIS and tools

More on GIS and the most popular parsing tool for Java.

GIS for Web Developers

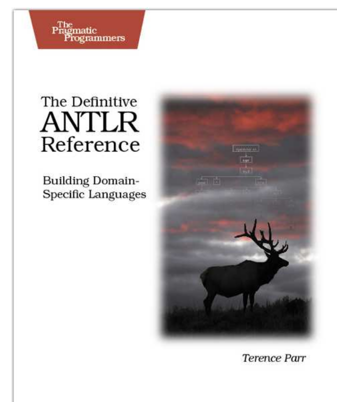
GIS for Web Developers you'll learn more about delivering a user a GIS experience from a Web server. You'll learn about Spatial Databases, Creating and using OGC Web Services and clients, and more.



GIS for Web Developers Adding Where to your Web Applications
Scott Davis
(275 pages) ISBN: 978-0-9745140-9-3. $34.95
http://pragprog.com/titles/sdgis

The Definitive ANTLR Reference

This book is the essential reference guide to ANTLR v3, the most powerful, easy-to-use parser generator built to date. Learn all about its amazing new LL(*) parsing technology, tree construction facilities, StringTemplate code generation template engine, and sophisticated ANTLRWorks GUI development environment. Learn to use ANTLR directly from its author!



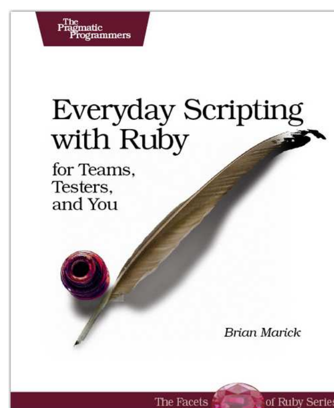
The Definitive ANTLR Reference: Building Domain-Specific Languages
Terence Parr
(384 pages) ISBN: 0-9787392-5-6. $36.95
http://pragprog.com/titles/tpantlr

Ruby & Erlang

Ruby is the object-oriented language of choice for forward-looking professionals. Erlang is your choice for high-availability, distributed systems that work in the real world.

Everyday Scripting with Ruby

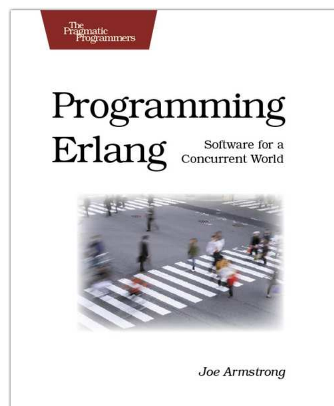
Don't waste that computer on your desk. Offload your daily drudgery to where it belongs, and free yourself to do what you should be doing: thinking. All you need is a scripting language (free!), this book (cheap!), and the dedication to work through the examples and exercises. Learn the basics of the Ruby scripting language and see how to create scripts in a steady, controlled way using test-driven design.



Everyday Scripting with Ruby: For Teams, Testers, and You
Brian Marick
(320 pages) ISBN: 0-9776166-1-4. $29.95
http://pragprog.com/titles/bmsft

Programming Erlang

Learn how to write truly concurrent programs—programs that run on dozens or even hundreds of local and remote processors. See how to write high-reliability applications—even in the face of network and hardware failure—using the Erlang programming language.



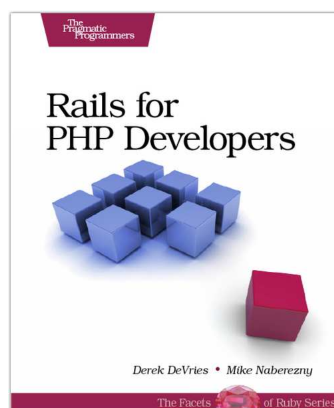
Programming Erlang: Software for a Concurrent World
Joe Armstrong
(536 pages) ISBN: 1-934356-00-X. $36.95
http://pragprog.com/titles/jaerlang

Get Started with Rails

Whether your background is in Java or PHP, we'll get you started in the Ruby on Rails web programming framework the easy way. Coming soon for .NET.

Rails for PHP Developers

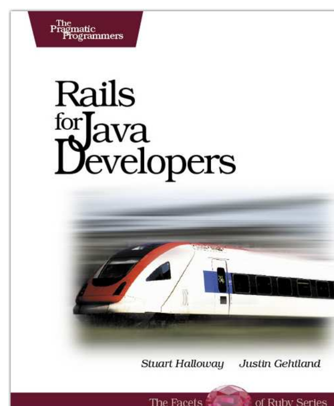
Rails for PHP Developers kick-starts your Rails experience by guiding you through learning both Ruby and Rails from a PHP developer's perspective. Written by developers with deep experience using PHP, Ruby, and Rails, this book leverages your existing knowledge of PHP to learn Rails application development quickly and effectively.



Rails for PHP Developers
Derek DeVries and Mike Naberezny
(375 pages) ISBN: 978-1-9343560-4-3. $34.95
http://pragprog.com/titles/ndphpr

Rails for Java Developers

Enterprise Java developers already have most of the skills needed to create Rails applications. They just need a guide which shows how their Java knowledge maps to the Rails world. That's what this book does. It covers: • the Ruby language • building MVC applications • unit and functional testing • security • project automation • configuration • web services This book is the fast track for Java programmers who are learning or evaluating Ruby on Rails.



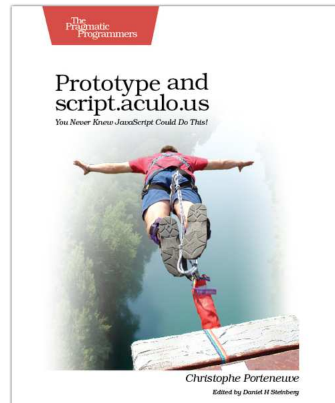
Rails for Java Developers
Stuart Halloway and Justin Gehtland
(300 pages) ISBN: 0-9776166-9-X. $34.95
http://pragprog.com/titles/fr_r4j

Web 2.0

Welcome to the Web, version 2.0. Ajax libraries and accessibility are key.

Prototype and script.aculo.us

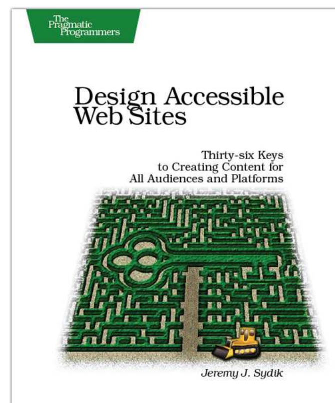
Tired of getting swamped in the nitty-gritty of cross-browser, Web 2.0–grade JavaScript? Get back in the game with Prototype and script.aculo.us, two extremely popular JavaScript libraries that make it a walk in the park. Be it Ajax, drag and drop, autocompletion, advanced visual effects, or many other great features, all you need is to write one or two lines of script that look so good they could almost pass for Ruby code!



Prototype and script.aculo.us: You Never Knew JavaScript Could Do This!
Christophe Porteneuve
(330 pages) ISBN: 1-934356-01-8. $34.95
http://pragprog.com/titles/cppsu

Design Accessible Web Sites

The 2000 U.S. Census revealed that 12% of the population is severely disabled. Sometime in the next two decades, one in five Americans will be older than 65. Section 508 of the Americans with Disabilities Act requires your website to provide *equivalent access* to all potential users. But beyond the law, it is both good manners and good business to make your site accessible to everyone. This book shows you how to design sites that excel for all audiences.



Design Accessible Web Sites: 36 Keys to Creating Content for All Audiences and Platforms
Jeremy Sydik
(304 pages) ISBN: 978-1-9343560-2-9. $34.95
http://pragprog.com/titles/jsaccess

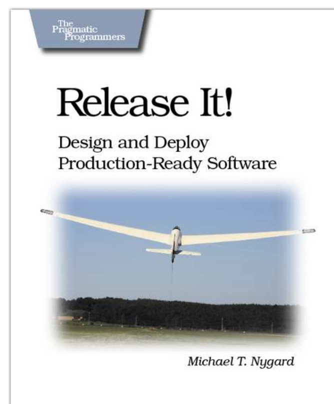
Real World Tools

Learn real-world design and architecture for your project, and a very pragmatic editor for Mac OS X.

Release It!

Whether it's in Java, .NET, or Ruby on Rails, getting your application ready to ship is only half the battle. Did you design your system to survive a sudden rush of visitors from Digg or Slashdot? Or an influx of real-world customers from 100 different countries? Are you ready for a world filled with flaky networks, tangled databases, and impatient users?

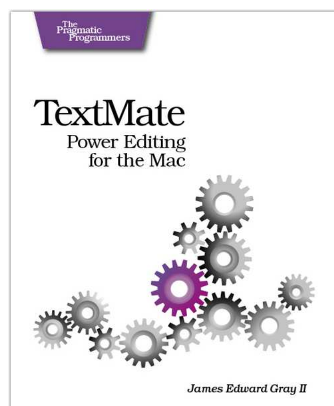
If you're a developer and don't want to be on call at 3 a.m. for the rest of your life, this book will help.



Release It! Design and Deploy Production-Ready Software
Michael T. Nygard
(368 pages) ISBN: 0-9787392-1-3. $34.95
http://pragprog.com/titles/mnee

TextMate

If you're coding Ruby or Rails on a Mac, then you owe it to yourself to get the TextMate editor. And, once you're using TextMate, you owe it to yourself to pick up this book. It's packed with information that will help you automate all your editing tasks, saving you time to concentrate on the important stuff. Use snippets to insert boilerplate code and refactorings to move stuff around. Learn how to write your own extensions to customize it to the way you work.



TextMate: Power Editing for the Mac
James Edward Gray II
(200 pages) ISBN: 0-9787392-3-X. $29.95
http://pragprog.com/titles/textmate

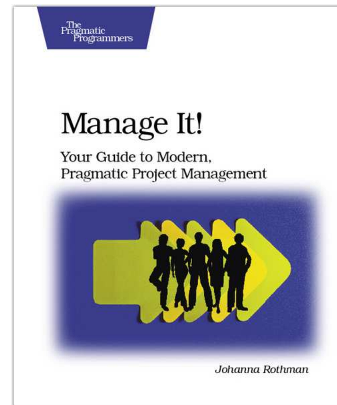
Leading Your Team

See how to be a pragmatic project manager and use agile, iterative project retrospectives on your project.

Manage It!

Manage It! is an award-winning, risk-based guide to making good decisions about how to plan and guide your projects. Author Johanna Rothman shows you how to beg, borrow, and steal from the best methodologies to fit your particular project. You'll find what works best for *you*.

• Learn all about different project lifecycles • See how to organize a project • Compare sample project dashboards • See how to staff a project • Know when you're done—and what that means.

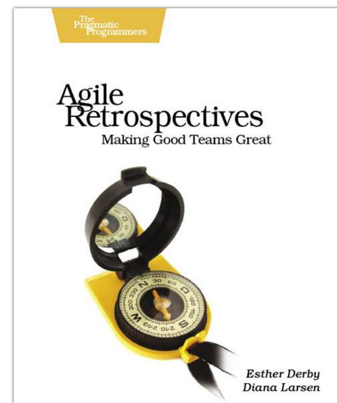


Your Guide to Modern, Pragmatic Project Management
Johanna Rothman
(360 pages) ISBN: 0-9787392-4-8. $34.95
http://pragprog.com/titles/jrpm

Agile Retrospectives

Mine the experience of your software development team continually throughout the life of the project. Rather than waiting until the end of the project—as with a traditional retrospective, when it's too late to help—agile retrospectives help you adjust to change *today*.

The tools and recipes in this book will help you uncover and solve hidden (and not-so-hidden) problems with your technology, your methodology, and those difficult "people issues" on your team.



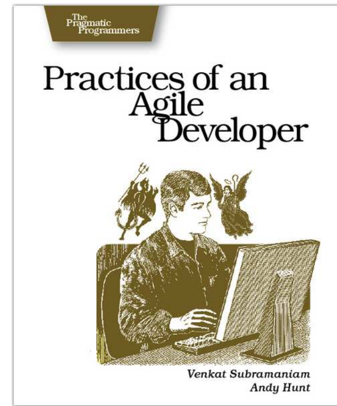
Agile Retrospectives: Making Good Teams Great
Esther Derby and Diana Larsen
(170 pages) ISBN: 0-9776166-4-9. $29.95
http://pragprog.com/titles/dlret

Getting It Done

Start with the habits of an agile developer and use the team practices of successful agile teams, and your project will fly over the finish line.

Practices of an Agile Developer

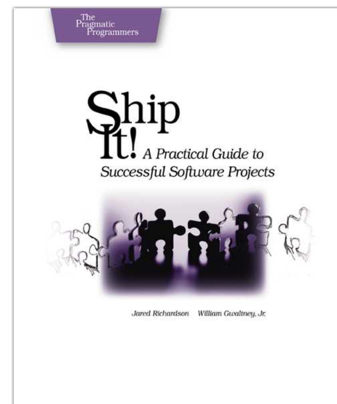
Agility is all about using feedback to respond to change. Learn how to • apply the principles of agility throughout the software development process • establish and maintain an agile working environment • deliver what users really want • use personal agile techniques for better coding and debugging • use effective collaborative techniques for better teamwork • move to an agile approach



Practices of an Agile Developer: Working in the Real World
Venkat Subramaniam and Andy Hunt
(189 pages) ISBN: 0-9745140-8-X. $29.95
http://pragprog.com/titles/pad

Ship It!

Page after page of solid advice, all tried and tested in the real world. This book offers a collection of tips that show you what tools a successful team has to use, and how to use them well. You'll get quick, easy-to-follow advice on modern techniques and when they should be applied. **You need this book if:** • you're frustrated at lack of progress on your project. • you want to make yourself and your team more valuable. • you've looked at methodologies such as Extreme Programming (XP) and felt they were too, well, extreme. • you've looked at the Rational Unified Process (RUP) or CMM/I methods and cringed at the learning curve and costs. • **you need to get software out the door without excuses.**



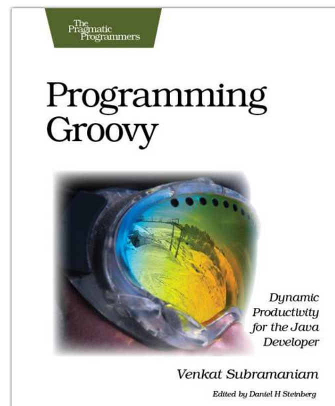
Ship It! A Practical Guide to Successful Software Projects
Jared Richardson and Will Gwaltney
(200 pages) ISBN: 0-9745140-4-7. $29.95
http://pragprog.com/titles/prj

Get Groovy

Expand your horizons with Groovy, and tame the wild Java VM.

Programming Groovy

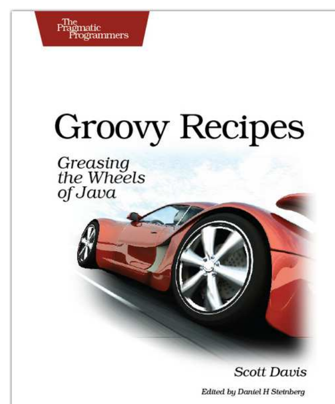
Programming Groovy will help you learn the necessary fundamentals of programming in Groovy. You'll see how to use Groovy to do advanced programming techniques, including meta programming, builders, unit testing with mock objects, processing XML, working with databases and creating your own domain-specific languages (DSLs).



Programming Groovy: Dynamic Productivity for the Java Developer
Venkat Subramaniam
(320 pages) ISBN: 978-1-9343560-9-8. $34.95
http://pragprog.com/titles/vslg

Groovy Recipes

See how to speed up nearly every aspect of the development process using *Groovy Recipes*. Groovy makes mundane file management tasks like copying and renaming files trivial. Reading and writing XML has never been easier with XmlParsers and XmlBuilders. Breathe new life into arrays, maps, and lists with a number of convenience methods. Learn all about Grails, and go beyond HTML into the world of Web Services: REST, JSON, Atom, Podcasting, and much, much more.



Groovy Recipes: Greasing the Wheels of Java
Scott Davis
(264 pages) ISBN: 978-0-9787392-9-4. $34.95
http://pragprog.com/titles/sdgrvr

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Desktop GIS's Home Page
http://pragprog.com/titles/gsdgis
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/gsdgis.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragprog.com/catalog
Customer Service:	`orders@pragprog.com`
Non-English Versions:	`translations@pragprog.com`
Pragmatic Teaching:	`academic@pragprog.com`
Author Proposals:	`proposals@pragprog.com`