GARY SHERMAN

THE GEOSPATIAL DESKTOP

OPEN SOURCE GIS AND MAPPING

locate
PRESS

The Geospatial Desktop: Open Source GIS and Mapping
by Gary Sherman

Published by Locate Press

Copyright © 2012 Gary Sherman
All rights reserved
978-0-9868052-1-9

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

Direct permission requests to `info@locatepress.com`, or mail to Locate Press, PO Box 4844, Williams Lake, BC, Canada V2G 2V8.

Editor Tyler Mitchell

Cover Design Julie Springer

Interior Design Based on Tufte-LaTeX

Publisher Website `http://www.locatepress.com`

Book Website `http://geospatialdesktop.com`

Contents

Table of Contents 3

1 Forward **5**

15 GIS Scripting **269**

15.1 GRASS 270

15.2 QGIS 270

To get the book, see **`http://locatepress.com`**

1

Forward

Locate Press is proud to bring this second edition of Gary's book into print. The original title, *Desktop GIS*, published by Pragmatic Press successfully sold out, yet there is increasing pressure to have it back in print.

It is special in another way too. It is the first title being published by Locate Press. As our flagship book for this new venture we are thrilled to have such a well-known and competent author as Gary Sherman. His enthusiasm and inspiration have been a pivotal influence in helping us start our publishing endeavour. Thank you Gary!

We've found that it is particularly sought-after by academics looking for a text book for introducing open source GIS into their courses. Seasoned open source GIS users also have been looking for copies to give to their friends.

The demand for having long term stable access to titles like this won't be going away anytime soon. Open source geospatial technology continues to take root around the world and across sectors. Financial crises tend to encourage open source adoption due to the

Copyright © 2012 Gary Sherman, All rights reserved

low price tag. Likewise more organisations need to adopt open standards for interoperability and open source geo applications have proven themselves regularly on that front. So now, more than ever, users are seeking out high quality training material from subject matter experts such as Gary.

The movement toward both cloud computing and mobile application platforms also puts increased stress on those products that cannot operate in a free and open source operating system. Fortunately, most open source geo applications are available across all major desktop and server operating systems. So when you need to move from one operating system to the other, your experience will still be similar to what you will learn in this book.

You are likely to find that this book will fill gaps in ways you didn't expect, from practical everyday tips for selecting software to in-depth examples for completing obscure tasks. I believe this is a special book, without comparison, as few other authors have yet taken on the challenge of covering this broad landscape with significant depth.

Enjoy the book,

Tyler Mitchell
Publisher

 Copyright © 2012 Gary Sherman, All rights reserved

15

GIS Scripting

(Partial Excerpt from The Geospatial Desktop)

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, including Perl, Python, PHP, and Ruby

 Copyright © 2012 Gary Sherman, All rights reserved

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.

15.1 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 (mingw.org) and Cygwin (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.

15.2 QGIS

Since version 0.9.*x*, QGIS has included 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.

 Copyright © 2012 Gary Sherman, All rights reserved

- Write plugins in Python instead of C++.
- Use PyQt[1] to build complete mapping applications using Python and the QGIS libraries.

[1] http://www.riverbankcomputing.com/software/pyqt/intro

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 Python bindings, this brings a whole new world of possibility—from simple plugins to complete applications. There is a large assortment of Python plugins for QGIS. To find out what's available, view the current list from the `Plugins`→`Fetch Python Plugins` menu.

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 looks similar to a terminal or command window. 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:

```
qgis.utils.iface.mainWindow().setWindowTitle('Hello from Desktop GIS!')
```

In Figure 15.1, on the following page, we can see the result of our

Copyright © 2012 Gary Sherman, All rights reserved

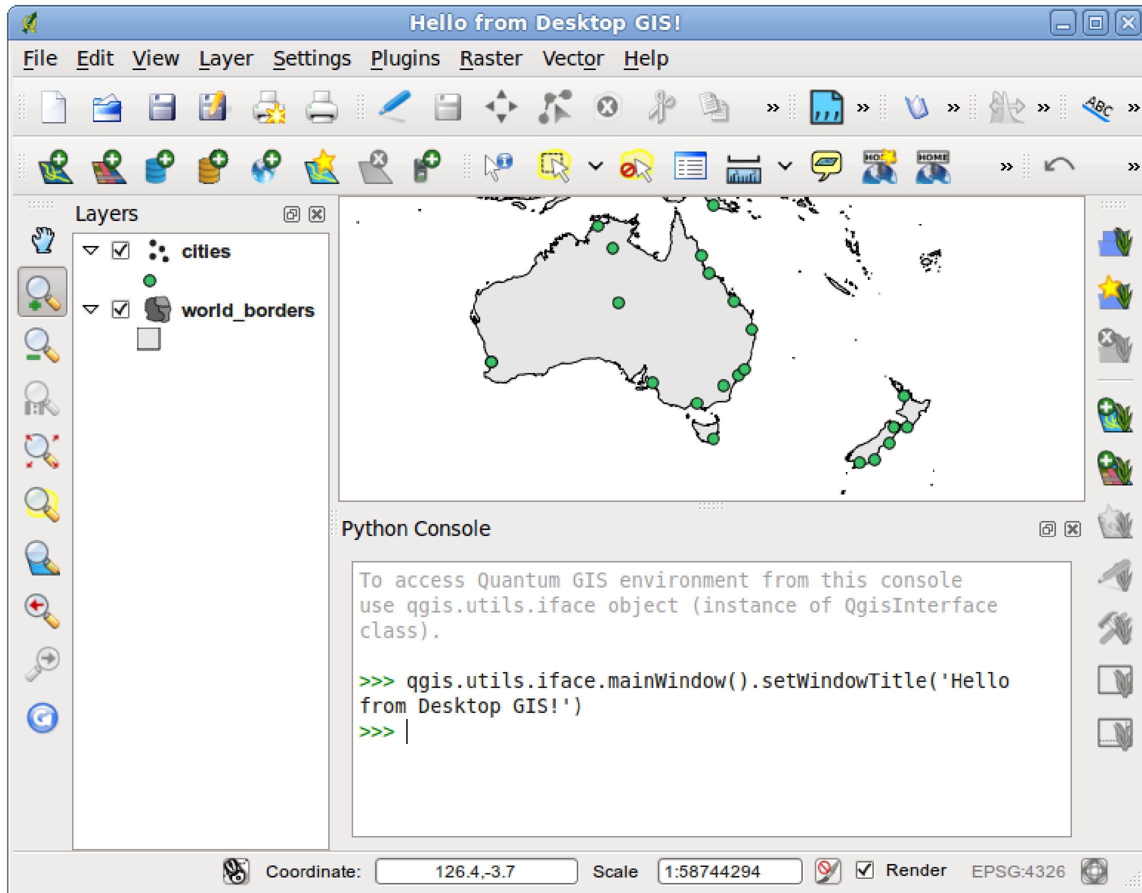

Figure 15.1: Changing the window title with Python

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 access the interface to the QGIS internals. The console is a good exploratory tool for learning about the QGIS API.

To manipulate the map canvas, we can try:

```
qgis.utils.iface.zoomFull()
```

 Copyright © 2012 Gary Sherman, All rights reserved

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.[2] 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.[3] If we look at the documentation for `QgisInterface`, we find functions such as the following:

[2] http://www.qgis.org/api

[3] http://www.qgis.org/api/classQgisInterface.html

- `zoomFull()`: Zoom to full extent of map layers.
- `zoomToPrevious()`: Zoom to previous view extent.
- `zoomToActiveLayer()`: 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(QString rasterLayerPath, QString baseName=QString())`: 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. You should also keep a copy of the PyQGIS Developer Cookbook[4] handy as well. It contains a lot of information for doing common operations like adding and working with layers, dealing with projections, styling layers.

[4] http://www.qgis.org/pyqgis-cookbook/

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.

 Copyright © 2012 Gary Sherman, All rights reserved

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 1.0 or greater, Python, PyQt, and the Qt developer tools.

MooseFinch: A mythical creature

Harrison just received the latest *Birding Extraordinaire* magazine, and in it he finds an article that describes locations for the exotic MooseFinch. 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 plugins the requirements are pretty simple and, as we'll see in a moment, something we don't have to worry about.

Regardless of your platform, you'll find your Python plugins are installed in the `.qgis` subdirectory of your home directory:

- *Mac OS X and Linux*: `.qgis/python/plugins`
- *Windows*: `.qgis\python\plugins`

For QGIS to find our Python plugin, we have to place it in 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 our plugin directory (for example, `$HOME/.qgis/python/plugins on Linux and Mac`) and initialize any plugins it finds. Once that's done, our Python plugin will show up in the QGIS plugin manager

 Copyright © 2012 Gary Sherman, All rights reserved

where we can activate it just like the other plugins that come with QGIS. You can also specify an additional path for QGIS to search for plugins by using the `QGIS_PLUGINPATH` environment variable. OK, enough of that, let's get started writing our plugin.

Setting Up the Structure

Back in the old days (around version 0.9) we had to create all the boilerplate for a Python plugin by hand. This was tedious and basically the same for each plugin. Fortunately that's no longer the case—we can generate a plugin template using the Plugin Builder.

Boilerplate: standardized pieces of text for use as clauses in contracts or as part of a computer program

The Plugin Builder is itself a Python plugin that takes some input from you and creates all the files needed for a new plugin. It's then up to you to customize things and add the code that does the real work. To use the Plugin Builder, first install it from the `Plugins→Fetch Python Plugins` menu as seen in Figure 15.2.

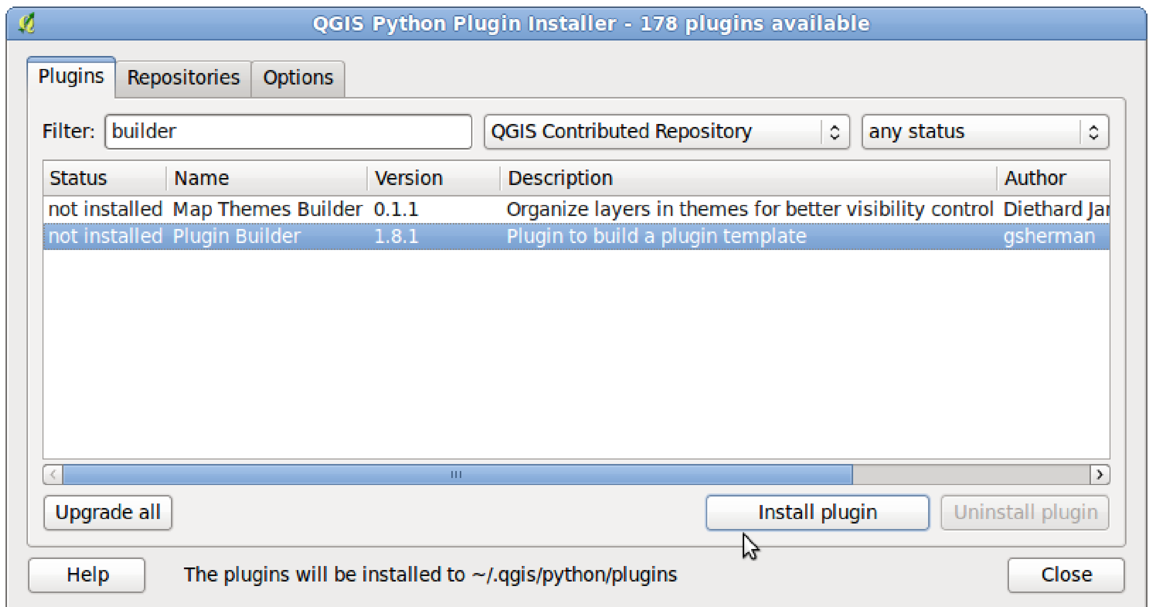

Figure 15.2: Installing the Plugin Builder

When you click on the *Install plugin* button the Plugin Builder will be installed and you'll find a tool for it on the Plugins toolbar and

 Copyright © 2012 Gary Sherman, All rights reserved

menu entries under `Plugins`→`Plugin Builder`

Let's generate the structure for our `Zoom to Point` plugin by clicking on the `Plugin Builder` tool or menu item. We are presented with a dialog that contains all the fields needed to create the plugin. On the left side of the plugin dialog you'll see some hints about what is expected for each field. Figure 15.3 shows all the fields needed to generate the Zoom to Point plugin.

Figure 15.3: Plugin Builder Ready to Generate the Zoom to Point Plugin

When we click on OK, the Plugin Builder generates a bunch of files:

icon.png
: A default icon used on the toolbar. You will likely want to customize this to better represent your plugin.

__init__.py
: This script initializes the plugin, making it known to QGIS. The name, description, version, icon, and minimum QGIS version are each defined as Python methods.

 Copyright © 2012 Gary Sherman, All rights reserved

Makefile

This is a GNU makefile that can be used to compile the resource file `resources.qrc` and the user interface file (`.ui`). This requires gmake and works on both Linux and Mac OS X and should also work under Cygwin on Windows.

metadata.txt

The metadata file contains information similar to the methods in `__init.py__`. Beginning with QGIS 2.0, the metadata file will be used instead of `__init.py__` to validate and register a Python plugin.

resources.qrc

This is a Qt resource file that contains the name of the plugin's icon.

ui_zoomtopoint.ui

This is the Qt Designer form that provides a blank dialog with OK and Cancel buttons. It's up to you to customize this to build your plugin's user interface.

zoomtopoint.py

This is the main Python class for your plugin that handles loading and unloading of icons and menus, and implements the `run` method that is called by QGIS when your plugin is activated. You'll need to customize the `run` method to make the plugin do its magic.

zoomtopointdialog.py

This Python class contains the code needed to initialize the plugin's dialog.

You'll notice the naming of a number of the files is based on a lower case version of the name you provide for your plugin, in this case `zoomtopoint`.

After the plugin generates the needed files, a results dialog is shown that contains some helpful information, as shown in Figure 15.4, on the next page.

Copyright © 2012 Gary Sherman, All rights reserved

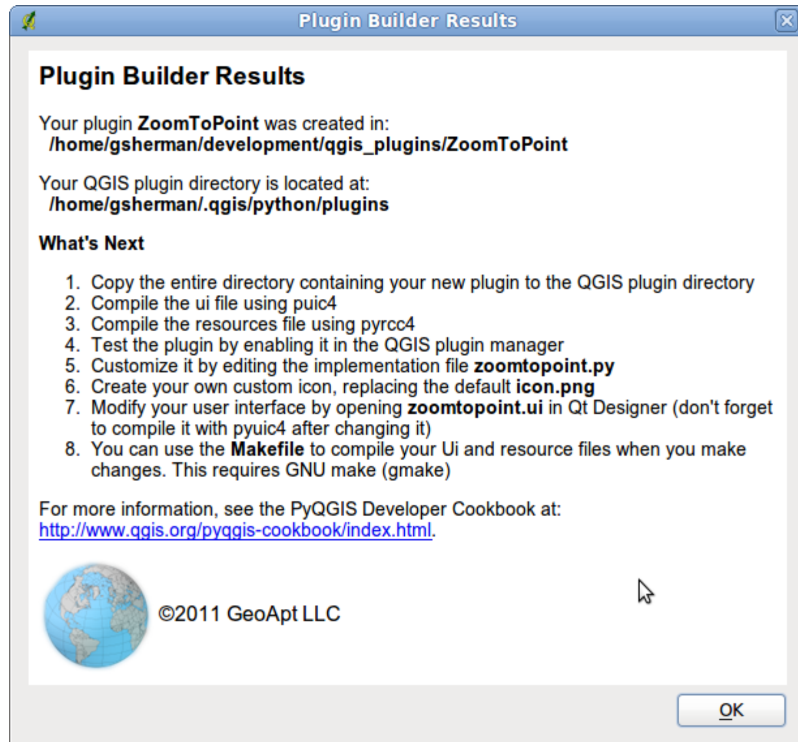

Figure 15.4: Results of Generating the ZoomToPoint Plugin

The results dialog contains helpful information including:

- Where the generated plugin was saved
- The location of your QGIS plugin directory
- Instructions on how to install the plugin
- Instructions on how to compile the resource and user interface files
- How to customize the plugin to make it do something useful

At a minimum the only files we have to modify to get the plugin functional are the user interface file (`.ui`) and the implementation file (`zoomtopoint.py`). If you need additional resources (icons or images) you will need to modify `resources.qrc`.

 Copyright © 2012 Gary Sherman, All rights reserved

Defining Resources

If you use the Plugin Builder you don't have to modify the resources file, but you do have to compile it. Let's take a look at what's in the resource file:

```
<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.[5] The icon 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.

[5] Plugin Builder created the prefix for you based on the plugin name.

You don't have to change the icon during development—the default created by Plugin Builder works fine.

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 compiled, we need to build the GUI to collect the information for ZoomToPoint.

Customizing the GUI

Plugin Builder created the GUI for us, but it needs to have controls added to it in order to get our plugin working. To do this, 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.

 Copyright © 2012 Gary Sherman, All rights reserved

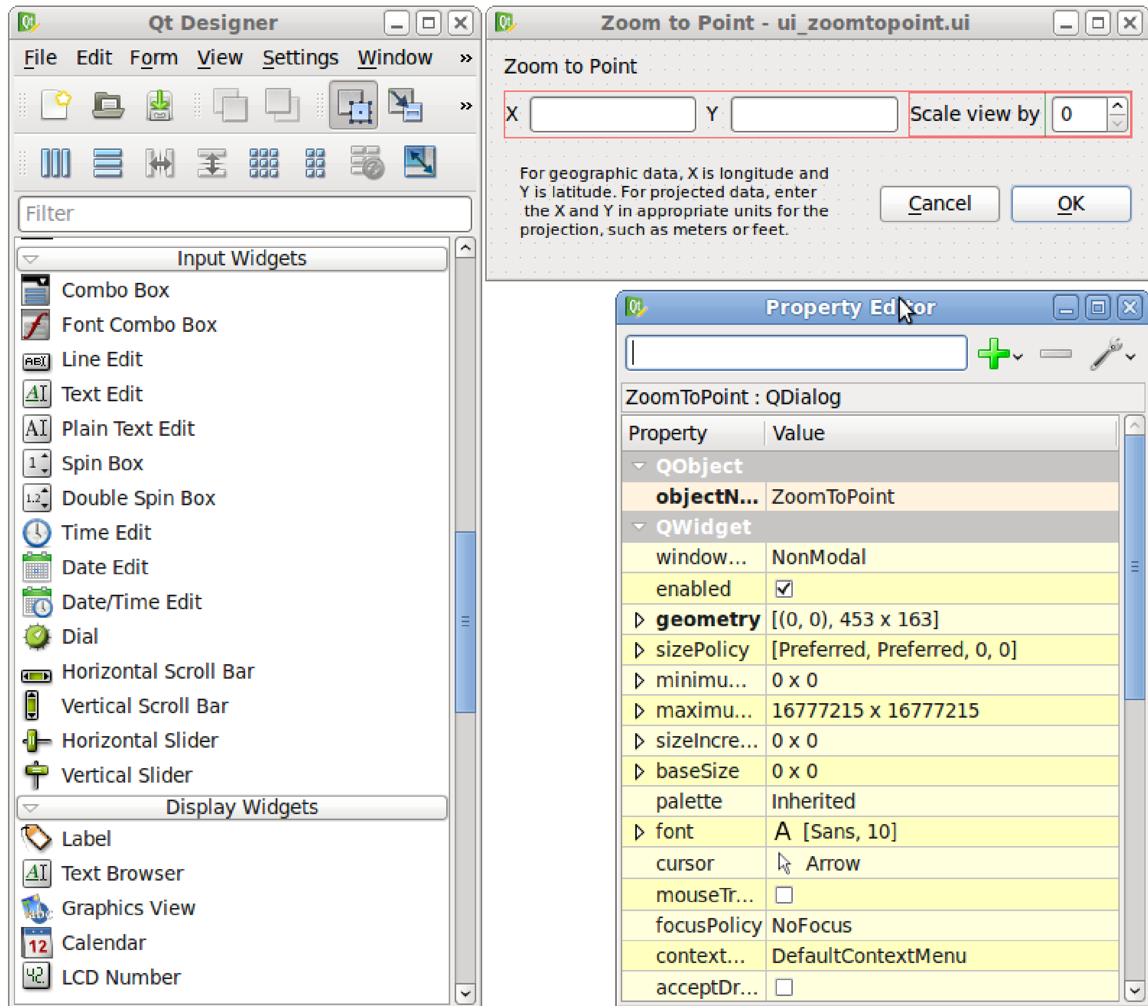

Figure 15.5: Plugin dialog box in Qt Designer

 Copyright © 2012 Gary Sherman, All rights reserved

Our dialog box is pretty simple. In Figure 15.5, on the facing 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 the excellent documentation on `Designer` on the Qt website[6] or in your Qt documentation directory.

[6] `http://doc.qt.nokia.com`

We start by opening our generated dialog box using the `File` menu and selecting `ui_zoomtopoint.ui`. Then we add text labels and text edit controls, as shown in Figure 15.5, on the preceding page. 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 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 completed dialog box to Python, we use the PyQt `pyuic4` command to compile it:

```
pyuic4 -o ui_zoomtopoint.py ui_zoomtopoint.ui
```

This gives us `ui_zoomtopoint.py` containing the code necessary to create the dialog box when the plugin is launched. It's important to maintain the naming convention as the generated Python code relies on specific names in order to find the components it needs. For example, the Python script that initializes the dialog imports `ui_zoomtopoint.py`:

```
  from ui_zoomtopoint import Ui_ZoomToPoint
```

If the compiled dialog is not named properly the plugin will fail to initialize.

Our GUI is now ready for use. All we need to write now is the Python code to interact with the QGIS map canvas.

 Copyright © 2012 Gary Sherman, All rights reserved

Getting to Zoom

Up to this point we've been laying the groundwork for our plugin. Before we write the code to zoom the map, let's take a brief look at some of the requirements for our plugin that are found in `zoomtopoint.py`. The Plugin Builder generates this code for us but it is good to get an idea of what it does.

Every Python script that uses the QGIS libraries and PyQt needs to import the QtCore and QtGui libraries, as well as 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. Here are the first few lines from `zoomtopoint.py`, excluding the header comments that appear at the beginning of the file:

zoomtopoint.py

```
1  # Import the PyQt and QGIS libraries
2  from PyQt4.QtCore import *
3  from PyQt4.QtGui import *
4  from qgis.core import *
5  # Initialize Qt resources from file resources.py
6  import resources
7  # Import the code for the dialog
8  from zoomtopointdialog import ZoomToPointDialog
9
10 class ZoomToPoint:
11  ...
```

You can see that in lines 2 through 4 we import the needed Qt libraries as well as the QGIS core library. Following that we import the resources that contain our icon definition and in line 8 we import the dialog loader class. Line 10 begins our class definition for the plugin. The implementation of our plugin all takes place within the `ZoomToPoint` class. The methods we are about to discuss are all members of `ZoomToPoint` and are shown in the listing below:

zoomtopoint.py

```
10 class ZoomToPoint:
11
12     def __init__(self, iface):
13         # Save reference to the QGIS interface
14         self.iface = iface
```

 Copyright © 2012 Gary Sherman, All rights reserved

```
15
16      def initGui(self):
17          # Create action that will start plugin configuration
18          self.action = QAction(QIcon(":/plugins/zoomtopoint/icon.png"), \
19              "Zoom to point...", self.iface.mainWindow())
20          # connect the action to the run method
21          QObject.connect(self.action, SIGNAL("triggered()"), self.run)
22
23          # Add toolbar button and menu item
24          self.iface.addToolBarIcon(self.action)
25          self.iface.addPluginToMenu("&Zoom to point...", self.action)
26
27      def unload(self):
28          # Remove the plugin menu item and icon
29          self.iface.removePluginMenu("&Zoom to point...",self.action)
30          self.iface.removeToolBarIcon(self.action)
31
32      # run method that performs all the real work
33      def run(self):
34          # create and show the dialog
35          dlg = ZoomToPointDialog()
36          # show the dialog
37          dlg.show()
38          result = dlg.exec_()
39          # See if OK was pressed
40          if result == 1:
41              # do something useful (delete the line containing pass and
42              # substitute with your code
43              pass
```

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 18, we create our action by supplying

Copyright © 2012 Gary Sherman, All rights reserved

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.

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 `addPluginToMenu` 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 the previous listing the `run` method is shown as it was generated by Plugin Builder. In the listing below we have modified the `run` method to zoom to a point.

zoomtopoint.py

```
32   # run method that performs all the real work
33   def run(self):
34     # create and show the ZoomToPoint dialog
35     dlg = ZoomToPointDialog()
```

 Copyright © 2012 Gary Sherman, All rights reserved

```
36      dlg.show()
37      result = dlg.exec_()
38      # See if OK was pressed
39      if result == 1:
40        # Get the coordinates and scale factor from the dialog
41        x = dlg.ui.xCoord.text()
42        y = dlg.ui.yCoord.text()
43        scale = dlg.ui.spinBoxScale.value()
44        # Get the map canvas
45        mc=self.iface.mapCanvas()
46        # Create a rectangle to cover the new extent
47        extent = mc.fullExtent()
48        xmin = float(x) - extent.width()  / 200 * scale
49        xmax = float(x) + extent.width()  / 200 * scale
50        ymin = float(y) - extent.height() / 200 * scale
51        ymax = float(y) + extent.height() / 200 * scale
52        rect = QgsRectangle( xmin, ymin, xmax, ymax )
53        # Set the extent to our new rectangle
54        mc.setExtent(rect)
55        # Refresh the map
56        mc.refresh()
```

The first step is to create the dialog box on line 35, show it, and then call the `exec_` method. 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 39. 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 41 through 43). 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 fetch the extent rectangle from the map canvas using the `fullExtent` method (lines 47). extent. In lines 48 through 52 we calculate the new bounds using the scale value and a constant and use them to create a new rectangle. Once we have the rectangle, we are ready to zoom the map by calling the map canvas `setExtent` method (line 54). To actually get the map to zoom, we call the map canvas `refresh` method in line 56 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 generate

 Copyright © 2012 Gary Sherman, All rights reserved

a template using Plugin Builder. Next we optionally set up our resource file with a custom icon and design the dialog box using Qt Designer. 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, of which we wrote very little by hand.

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 15.6, on the facing page. Behind it you'll see the map zoomed to the coordinates we specified. Notice the magnifying glass icon on the right of the Plugins toolbar (just above the layer list). That's the icon I specified for the final version of the 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]

[7] Actually, you can find information on writing QGIS plugins in C++ on the QGIS wiki at `http://wiki.qgis.org`.

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 don't have to write all that code to 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 a simple tutorial for more in-

 Copyright © 2012 Gary Sherman, All rights reserved

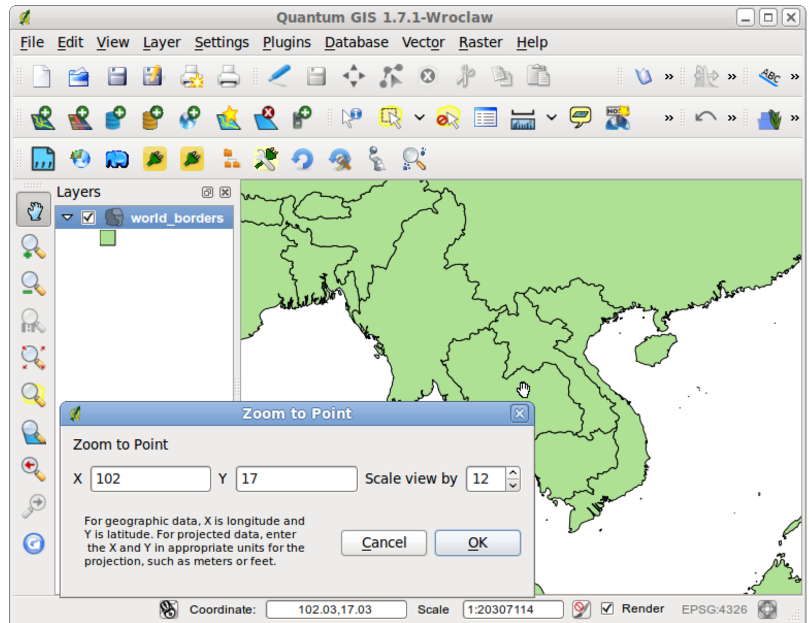

Figure 15.6: ZoomToPoint plugin in use

formation.[8]

[8] http://geospatialdesktop.com/Creating_a_Standalone_GIS_Application_1

To get the complete chapter and book, see **`http://locatepress.com`**

Excerpted from The Geospatial Desktop

Copyright © 2012 Gary Sherman, All rights reserved