Writing

With

Vim

by Anthony Panozzo

```
-- VISUAL LINE --          1
```

Writing With Vim

Anthony Panozzo

This book is for sale at
http://leanpub.com/vim-for-writers

This version was published on 2016-10-31

Contents

1. Get in touch

Thanks for previewing this book! I learned a lot while writing it, and am glad that I can share what I know with you.

Notice any issues or have any questions? Shoot me an email at panozzaj@gmail.com. If you have kind words, those are also always appreciated!

If you'd like to follow my future work, which will almost certainly include writing about (and with!) Vim, subscribe to my free newsletter[1] that gets updated every now and then. Alternately, you can subscribe to my blog via RSS[2] or follow me at @panozzaj[3] on Twitter.

To purchase the full version, which has over 20,000 words and nine chapters, visit the official page of Writing With Vim[4]

[1]http://22ideastreet.us2.list-manage1.com/subscribe?u=4f7071bc6396efbc773233548&id=74a42961ae

[2]http://www.panozzaj.com/atom.xml

[3]https://twitter.com/panozzaj

[4]https://leanpub.com/vim-for-writers

2. Introduction

This book aims to quickly get you up and running with composing and editing long-form and formatted documents with Vim.

I envision this book will be the most useful for programmers or developers who are already familiar with Vim and who seek to use it for tasks traditionally associated with word processor applications.

The power of Vim may also be a great reason for technical writers or book authors to learn a new tool. I think there are advantages to using any text editor for writing, and the Vim ecosystem has many plugins that can make your writing experience better.

Whether you are new to Vim or a veteran, try following along with the examples in this book to get a sense of what settings you like and will get the most use out of.

Prerequisites / assumptions

Here are things that you should know or do prior to being able to get the most out of the book.

What versions of Vim?

I recommend at least using the latest stable Vim version available. The newer the version, the more likely that it will contain bug fixes and useful new features. For example, prior to Vim version 7.0, there were no advanced spelling features. I personally stay current with the latest versions of the code by tracking the repository, but sticking to major revisions will take you far.

This book assumes that you have a working, modern, copy of Vim installed. If not, you can install the Vim on many platforms, including Mac OS X, Linux, and Windows. The methods for installing Vim[1] are system-dependent but normally straightforward.

There should not be much difference between the various flavors of Vim. For example, running Vim from the command line should work, as would running a more polished version like gVim or MacVim.

You could even use Neovim[2], a new fork of Vim that is trying to clean up some of Vim's crufty underlying code and add better asynchronous support. There might be a plugin or two that doesn't work, but the authors of Neovim try to maintain backward compatibility with Vim. Also, most plugin maintainers generally try to support Neovim.

[1]http://www.vim.org/download.php
[2]https://neovim.io/

Basic Vim concepts and usage

I may expand upon this section in future editions of the book, but for now I will assume that you have a basic working understanding of Vim. Concepts like how to create documents, navigating through documents, and explanation of basic modes will not be covered here. To get more familiar with how to work with Vim, check out the official Vim resources[3]. Those new to Vim or those looking for a brush-up can also look at this free PDF[4] which has little overlap with this book.

Something to write

It helps to have something that you want to write in order to set up and practice the concepts in this book. Maybe you have some blog posts that you have been mulling over, or an idea for a manuscript. Having something that keeps your interest will help make the learning curve easier. I generally think this is the best way to learn new programming languages or frameworks as well. As you read through this book, try out different settings and plugins and see which ones appeal to you and your preferred writing style.

[3]http://www.vim.org/docs.php

[4]http://therandymon.com/papers/vimforwriters.pdf

Philosophy of using Vim for writing

There are several benefits for using Vim for writing, and some tradeoffs you need to be aware of.

Portability

One of the benefits of using a text editor like Vim for writing is portability. You write documents using simple text files or lightweight markup. These files can be easily shared across operating systems and can be read by many applications. Plain-text files are one of the basic building components of Unix-like operating systems, and because of this, there are many tools for manipulating plain-text files.

Vim is open source, which means that anyone can modify the program and extend it. You don't need to worry about not being able to read your file when a new version of Vim comes out, since there are thousands of programs that need to operate on plain text files. There are no proprietary formats to worry about, and we will be able to read plain-text files as easily today as we will a hundred years from now.

Speed

When you compare the time it takes to load up a traditional word processing program to the time it takes to open a text file in Vim, Vim wins hands down. My computer typically gets bogged down for at least thirty seconds when loading up a Microsoft Word or LibreOffice document, while I can start editing within seconds with Vim. The cursor does not skip or lag when I type in Vim, resulting in a feeling of flow as the words pour smoothly onto the screen.

If you are familiar with Vim or are willing to take the time to get to know the keyboard shortcuts it provides, composing, editing, navigating, and searching can be much faster than traditional word processor applications.

One positive side effect of writing speed is that you can stay in the flow of composing or editing. Each interruption breaks up the cadence of your thinking, so writing seems more coherent when there are fewer impediments.

If you are a software developer who uses Vim regularly, you already have a lot of the muscle memory needed to quickly move around your documents. If you generally lapse into using the `j` and `k` keys to scroll around in entirely inappropriate applications, or send emails with `:wq` at the end of them, then your muscle memory can be used to write great articles and books.

Muscle memory is important because it is one less thing to think about, and again, it contributes positively to the flow of your writing. When you can dice up and reformat several paragraphs with just a few keystrokes, it is much better than messing around with a mouse. Instead of hunting around for the “make bulleted list” feature in a complex ribbon, you can just make some dashes at the beginning of the line and be done with it.

Content over formatting

The creators of the World Wide Web had the right idea when they separated content from presentation with HTML and CSS. HTML is a basic markup language that indicates what certain parts of web pages *mean*. However, CSS complements HTML by taking that semantic markup and giving style. You can have the exact same content and style it different ways.

In the same way, it helps to consider the layout and presentation of your book or article separately from the content of it. Sure, if you are making a graphic novel, these two are intertwined. But for the most part, tweaking margins or line spacing or any number of settings is less important than writing something that is useful for others to read.

In today's world, if you publish a document on the web, you can imagine that it might be viewed in a browser on a desktop, on a tablet, or on a mobile device. Similarly, if you publish a book, it could be published in a number of different formats. So writing with Vim leads you to focus on the content first, and the presentation second.

Flexibility

Vim has a variety of settings and can be configured and scripted. Many Vim plugins can be found on vim.org[5] and

[5]http://www.vim.org

on code sharing sites like Github[6]. This book will cover common configurations and useful plugins for writing.

You have the power to make Vim do exactly what you want. The best configuration will follow the Unix philosophy[7] of doing one thing well. There are dozens of plugins that you can install to give you bits and pieces of the setup that you want, without bogging you down with much unneeded functionality.

Caveats

Writing with Vim is not necessarily easy, but hopefully we can mitigate any potential disadvantages with this book.

The flip side of extensibility is that you can spend inordinate amounts of time that would otherwise be spent writing fiddling with Vim configuration or publishing pipeline scripts. I try to minimize this by focusing on having a writing goal per day. Maybe the goal is a few hundred words a day. When I get that, then I give myself permission to fiddle with my build scripts for a little while to streamline the process.

I also recommend starting with the basics and growing what your editor can do as you find it limiting. It's tempting to install every plugin that you see, but sometimes they conflict or do the same thing but in a different way. Try to

[6]https://github.com

[7]https://en.wikipedia.org/wiki/Unix_philosophy

install one thing at a time and test it out for a bit to make sure that it is valuable enough to make the cut into your editor.

Vim itself takes a bit of time to learn, so if you need to brush up, it might diminish your effectiveness for a little while. However, using Vim for writing is probably slightly less taxing than using it to edit code since so much time is spent in insert mode when writing.

Like learning to touch type or other tools and techniques with a steep learning curve, there is a period of lowered productivity followed by even higher productivity. Stick through the learning period and you will emerge with a powerful writing tool.

Typeface Conventions

Throughout the book I use varying text types to more clearly delineate some things.

`Monospaced text` typically represents a filename or piece of code.

If the monospaced text is preceded by a dollar sign ($), it is used to show things you would type at the command-line console window:

```
$ ./run_some_script
```

If the monospaced text is preceded by a colon (:), it is used to show things you would type in Vim in normal mode to get into command mode:

```
:set someVimOption=true
```

Any time you can type something in the command mode, you can also insert it into your `.vimrc` by not using the colon. (I'll explain the `.vimrc` file in the next chapter.)

I write a mnemonic in quotes in square brackets after the first instance of abbreviated normal mode commands. For example: Type `w` ["word"] to move forward a word.

If something is dangerous or depends on something else, I'll have a little image to indicate this fact:

⚠ Whoa! Be careful!

Vim Conventions

Where possible, I will use the expanded version of any Vim commands. This should make examples clearer at the expense of slightly more text. I recommend doing this throughout your `.vimrc` file for better documentation. If you want, you can use the abbreviations of them that Vim recognizes to reduce the amount of typing you need to do.

Learning More

Whenever you want to learn more about a command, you can invoke Vim's internal help by typing `:help <command>` when in normal mode.

Dogfooding

This entire book was written just using Vim. No other word processors or text editors were used.

3. Setting Up Vim to Enable Smooth Writing

Your `.vimrc`

Vim can be configured through a variety of commands. You can invoke these commands by entering into command mode and typing them out. However, once you have done this a few times, you will probably want to save the configuration changes somewhere.

At program startup, Vim looks in several files on your filesystem for configuration settings. The most common file to edit (and the one I will reference) is generically called your `.vimrc`. It typically lives in your home or documents directory. For Linux and Mac OS X systems, it will be located at ~`/.vimrc`. On Windows machines, it will be called _`vimrc` and located wherever your HOME environment variable points at. Of course, you can edit your `.vimrc` file with Vim.

(To easily find the location of the `.vimrc`, you can run `:echo $MYVIMRC`. The absolute path will be displayed at the bottom of the screen.)

Vim will not automatically save any configuration settings you set. It will discard them at the end of each session.

In the `.vimrc` file you can type any commands you would normally type in command mode in the Vim window and they will be loaded whenever you start up a new Vim instance. Commands in the `.vimrc` should not have the colon in front of them. For example, if this book says:

```
:set foo=bar
```

In your `.vimrc` you would only have:

```
set foo=bar
```

To test changes, you can source the `.vimrc` file with `:source ~/.vimrc` to pick up the newly changed function, or restart Vim (since Vim loads the `.vimrc` on startup.)

If you make a mistake, you can always edit the `.vimrc` file and revert the changes you made.

Set `nocompatible`

By default Vim starts in Vi compatibility mode, which ensures that Vim behaves exactly like the classic Vi. However, since we are going to be using Vim features that are not present in Vi to make your writing experience better, you need turn off this setting. Ensure that your `.vimrc` contains the following line near the top:

```
" Use new Vim features
set nocompatible
```

Your Vim writing function

If you simply put commands in your `.vimrc` file, they will execute for all files you open with Vim. Well that's great, but what happens if you have settings that you *only* want to use when you plan to write prose? Currently, they would also be present when you are modifying normal documents or code files.

In this case, I recommend setting up a writing function that you can call whenever you want to write.

You can do this by defining a function inside ~`/.vimrc` like so:

```
function! Writing()
  " your calls go here
endfunction
```

The middle line has a comment starting with the double quote character that you can remove at any time and replace with configuration calls. You can also use this to document your `.vimrc` so you remember what certain settings do. Again, if you change the function in your `.vimrc`, you'll need to source it again or restart Vim.

To actually call the function and execute the contents, use `:call Writing()`.

Reducing distractions and basic configuration

One of the first things I would do to get a finely-tuned writing program would be to turn off some non-essential Vim functions to get a cleaner working space. There are also some plugins that reduce visual clutter that I'll cover later in the book.

Turning off line numbers

I find line numbers to be more of a hindrance than a help when writing prose. You can turn them off once by typing `:set nonumber` while in normal mode, or turn them off in your writing function by adding to it like:

```
function! Writing()
  set nonumber  " Turn off line numbers
endfunction
```

Removing sidebars and other elements

If you want to get rid of screen elements to aid you in focusing on the task at hand, you can mess around with `guioptions`. Here are some sample commands to add to your writing function:

```
function! Writing()
  " Remove right sidebars
  set guioptions-=r
  " Remove left sidebars
  set guioptions-=L
endfunction
```

If you have some symbols (toolbars) or menus at the top of the screen, you can try subtracting the `T` or `m` flags also. You can invoke `:help guioptions` for more information. Note that the non-spacing between the `-=` and the other part of the line is important.

If you remove scrollbars, you can still know where you are in your document. Just press `Ctrl+G` for something like:

```
"chapter2.markdown" line 48 of 271 --17%-- col \
52
```

This will tell you what file you are in, which line you are on, what percentage through the document you are, and what column of the line you are on.

Turn off cursor blinking

When I'm pondering how to finish a thought or the next sentence to write, I don't like to have a blinking cursor to distract me. If you feel the same way and are using a graphical version of Vim, try:

```
" Set the cursor to not blink
:set guicursor=a:blinkon0
```

Removing the status bar

Toward the bottom of the screen, there is usually a status bar that shows your filename. If you are just writing in one file, fullscreen, you may want to hide that since there is only one file you are editing. Fortunately, Vim provides us an easy way to do this by saying:

```
" Hides status bar when only one file open
:set laststatus=1
```

Changing your writing font

Note: this section only applies to graphical Vim (non-terminal).

To set the font, you can manually change it by going to the menu bars and hunting around for the font setting submenu. Hunting each time is not convenient, so there are ways to set the font with commands in your ~/`.vimrc`.

To see the current font setting, run:

```
:set guifont
```

The result of the `guifont` check will be something like:

```
guifont=Inconsolata:h20
```

in MacVim or like:

```
guifont=Inconsolata\ 20
```

in other Vim versions. This says that the current font is “Inconsolata”, at a text size of twenty points. There are generally seventy-two points to an inch.

To select from a list of fonts, you can run:

```
set guifont=*
```

This will open a dialog to select the font or list out the fonts available.

With all of this information, it should come as no surprise that you can set the font setting with something like:

```
set guifont=Inconsolata:h20
```

I use something like this in my writing function to bump up the font size. This helps prevent simple typos and to be able to more easily see what I am writing.

Local writing function

The writing function as currently specified has one problem: if you have multiple buffers open, it will change the configuration for all of them. This might present a problem if you are doing different types of work in different windows inside of Vim. You could easily have one buffer that has your project's documentation that is in "writing mode", and another right next to it that shows code in your normal configuration. If you run the writing function once though, it will wipe out line numbers in the gutter of your programming files, enable spelling for your variable names, and so forth.

Sometimes this is acceptable. Maybe you are just editing prose files or you open prose and programming files in different Vim instances. But overall it seems better to call the function and have it operate on only one buffer.

To do this, you can use `:setlocal` in place of `:set` in your writing function. Not all settings that are done using `set` can be done with `setlocal`, but it is a step forward. Other settings like setting insert abbreviations can only be done at the global level, to the best of my knowledge.

Calling writing function automatically

If you don't want to have to invoke your writing function manually, you can have Vim run it whenever you open or create a file that has a certain extension or filetype:

```
augroup RunWritingFunctionForWritingFiles
  autocmd!
  autocmd BufNewFile,BufRead {*.txt,*.markdown} \
call Writing()
augroup END
```

This would work best when the writing function only operates on the current buffer, as the above section shows how to do.

Fullscreen mode

I generally like to have my Vim running in fullscreen mode to try to get into "the zone". Then, I can just focus on writing.

If you are using a graphical Vim and this sounds appealing, try invoking `:set fullscreen`. On MacVim, you can hold `Command` and press `CTRL-f` to put the window in fullscreen mode. You can get out of fullscreen mode by typing `:set nofullscreen` or pressing the same keyboard sequence.

If you are running Vim in a terminal window, you should put your terminal in fullscreen mode and Vim will match it by being fullscreen. The way to do this generally depends on your platform and terminal application type, so I won't cover that here.

4. Get in touch

Thanks for previewing this book! I learned a lot while writing it, and am glad that I can share what I know with you.

Notice any issues or have any questions? Shoot me an email at panozzaj@gmail.com. If you have kind words, those are also always appreciated!

If you'd like to follow my future work, which will almost certainly include writing about (and with!) Vim, subscribe to my free newsletter[1] that gets updated every now and then. Alternately, you can subscribe to my blog via RSS[2] or follow me at @panozzaj[3] on Twitter.

To purchase the full version, which has over 20,000 words and nine chapters, visit the official page of Writing With Vim[4]

[1]http://22ideastreet.us2.list-manage1.com/subscribe?u=4f7071bc6396efbc773233548&id=74a42961ae

[2]http://www.panozzaj.com/atom.xml

[3]https://twitter.com/panozzaj

[4]https://leanpub.com/vim-for-writers