An Introduction to Modal Text Editing with Vim

The first thing to note about Vim is that it is not merely an editor or a family of editors sharing a common software pedigree (Ed, Vi, Vim, GVim, NeoVim). Vim is also a philosophy of editing text—a way of doing things that could be (and often is) implemented in various contexts. For this reason, it is not our intention to start another text editor war. Vim-like modes are commonly implemented in many modern text editors like Emacs, Atom, and Sublime. It is also possible to implement distinct ideas from Vim into other programs. There are several plugins that offer Vim shortcuts for the browser, for example. Our plan then is to talk about Vim as both an idea and a piece of software. Because switching to Vim is not a trivial endeavor, we want the reader to understand the stakes and the mind shift made possible by this elegant interface with the word. We will begin then with the philosophy behind Vim editing and end with some instrumental details of Vim as an editor.

Key Concepts

What are the ideas behind the Vim way of doing things? Rather than divining the intentions of the program's developers (Ken Thompson, Bill Joy, Bran Moolenaar, and others) we will try to capture the spirit of the endeavor as we see it.

Let's start with the most obvious one: Vim is a **plain text** editor. That means that you will use other tools to format and to set your text into print. In Vim, text is just text—there are no italics or fancy fonts to speak of. We use Markdown and Pandoc to format our texts and to generate files in Microsoft Word, Open Office, Adobe Acrobat, and HTML formats. If that sounds limiting, it is, on purpose! The Unix philosophy of software development values "doing one thing and doing it well." Rather than being mediocre at many things related to word processing, Vim does one thing exceedingly well. Along with Emacs and a few other candidates for the title, it is the most advanced *text editor* available. Other complimentary and powerful tools can fill the role of typesetting and formatting of text.[1] Our goal then is to assemble a "tool chain" or a "stack" of

[1] On what these tools are and on the why it is good for our community to work in plain text file formats, read "Sustainable Authorship in Plain Text using Pandoc and Markdown" By Dennis Tenen and Grant Wythoff in The Programming Historian.

programs that answer to the requirements of academic writing. Vim will serve as the first and the foundational building block in assembling that stack.

Vim is further synonymous with **modal editing**. Modes will require the most cognitive adjustment for someone coming from mode-less text editing (Open Office, Microsoft Word, or Gmail, for example). The idea behind modal editing is simple. The process of writing consists of several distinct kinds of operations, chief among them typing and manipulating text. In most editors, those two things happen in the same "place" if you will. Vim organizes actions related to typing and actions related to text manipulation into separate modes. You type in what is called the "insert mode," which functions pretty much as you would expect from using tools like Microsoft Word or Gmail. A whole new series of powerful text manipulation tools becomes available to the writer in "normal mode." Think of it as lifting the pen from paper to pick up other tools like scissors and an eraser. In the *normal mode*, you can use your keyboard for editing text instead of typing. We will go over the commands in detail later. For now, you can start Vim (gVim or NeoVim) and press `i` to enter the *insert mode* (see bottom left of your screen). Type a few dummy sentences, then press `Esc` to exit into the *normal mode* again. As you get better at Vim you will spend more and more of your time in the *normal mode*. To quit Vim, type `:q`. Incidentally, the colon takes you to the "command mode." Here you can issue written commands to the editor, like `:w` to save or to "write" the file and `:q` to quit.

Text awareness and composability of commands are some of the implemented ideas that make Vim such a powerful tool for academic writing. Text awareness means that the editor intrinsically "understands" humanly-meaningful semantic units like characters, words, sentences, and paragraphs. Say for example you want to delete a word in your regular text editor. Pay attention to your exact keystrokes. It is likely that you would either just backspace a few times until the word is gone, or use the mouse to select the word to be deleted. This approach is slow and imprecise, because it treats text either as a sequence of individual characters or as a configuration of geometric shapes on the screen. You can still edit in that way in Vim's *insert mode* (`i`). But exit to the *normal mode* (by pressing `Esc`) and you are now able to delete the word under your cursor by typing `daw`. The keys `daw` stand for "delete a word." Here's where things get neat. What do you think `d3w` does? As one would intuitively expect, `d3w` stands for "delete three words." What about `das`? In *normal mode*, the incantation `das` stands for "delete a sentence." In either case, the delete command will delete around your cursor position. Press `u` a few times in *normal mode* to undo. The incantations `daw` and `das` delete the current word and the current sentence, respectively, even when your cursor is located mid-word or mid-sentence. Likewise, `dap` will delete the current paragraph, while something like `ci)` will "change inside parentheses," allowing the author to replace rather than simply delete words. Try experimenting with incantations like `d2s` or `cap` to see what happens.

Vim is ergonomically designed to keep your fingers at the keyboard, at the home row, minimizing finger movement involved in chord-like progressions like `Ctrl-C` and `Ctrl-V` (the usual way to copy and paste selections, for example). Instead of the arrow keys or the mouse, vimmers use `hjkl` for navigation. With time you can start using sentence-based navigation with `(` and `)`, or move around by paragraph with `[` and `]`. To move to the end of the line press `$`. To move to the beginning, try `0`. Remember to evoke these form the *normal mode*. Because you don't need to use the mouse that much (or at all) when you are good at Vim, and because the commands are compact, your wrists remain relatively still. Many writers report reduced hand strain. But these are small luxuries. The big payoff of modes and text awareness is **command composability**. In some important sense, Vim is a language for interacting with language. "Delete a word" has a grammatical structure: a verb and a noun. Once you become fluent in this language, you will be able to compose commands "on the fly," without thinking or looking things up. A measure of flow and fluency becomes possible, elevating mere editing into handcraft.

As with any language, the road to fluency is not short. But compared to a foreign language, Vim makes use of a small, controlled vocabulary. It is quick to learn. You will be able to do most common tasks after a week or so of practicing. And you can stop there. At a level of basic proficiency, Vim will already feel like a professional tool, adequate to the task of composing text with a sense of kinetic joy (some people liken it to being good at a video game). But once you understand that Vim is a language, you may also start getting interested in Vimscript, the computer language behind Vim. Although not the most beloved of computer languages, Vimscript ultimately offers **a modular toolkit for building your own editor.** Where other editors have fairly rigid ideas about how writing and text editing should work, Vim gives you complete control over all expects of word manipulation. For those not interested in programming, hundreds if not thousands of modules are available, tailoring Vim to individual needs and workflows. (We will cover the most common ones relevant to academic writing at the end of the article.)

To summarize, Vim's philosophy is a set of related concepts that start with plain text and lead to modal editing, text awareness, command composability, and, finally, a language for building your own customized text editing software. Any of these ideas could be implemented independently of Vim. Vim just happens to implement all of them in one small package, installed on most of the world's computers (except for Windows) by default since the 1970s, available for free, and in open-source, making further modification possible. We want our colleagues to use Vim because we want our community to build better interfaces with the word. The text editor provides the foundations of all scholarly activity. It is therefore paramount that we develop mastery over and take ownership of the tool (and not the other way around).

Getting Started with Vim

It is not our intention to supplant the many excellent tutorials on how to get started with Vim, but here is a sampling of Vim commands that may stimulate your curiosity. Think of the commands as having a grammar that usually begins with a verb and ends with a noun, with an adverb or an adjective in between. Use the built-in documentation by evoking `:help [command]`. Some of the common verbs available to you include:

Table 1: Vim Verbs

command	meaning	default scope	try
`d`	delete	explicit	`d7w`, `das`
`r`	replace	character	`r0`, `4r0`
`y`	copy or yank	explicit	`yy`
`p`	paste or put	once	`7p`
`a`	append	after cursor	
`A`	append	end of line	
`g`	go	explicit	
`u`	undo	change stack	
`Ctrl-r`	redo	undo stack	
`.`	repeat	last action	
`>`, `<`	shift right, left	current line	`3>>`
`/`	search forward		
`o`, `O`	new line below, above		

Some verbs require explicit scoping. For example, delete or `d` by itself does nothing until you specify what to delete.

Table 2: Text Objects

command	meaning	scope	examples
`aw`	a word	a sequence of letters	cat
`aW`	a WORD	a sequence of characters	dog56-72
`s`	sentence	punctuation	A bird?
`p`	paragraph	blank lines	

Evoke `:help text-object` to learn more about text objects. The best way to understand text and motions objects (coming up next) is just to try using them with *delete* (`d`), *yank* (`y`) and *put* (`p`). Next come the motion commands, which like text objects give you a quick way to compose evocations like “delete the line above” and “move to the next parentheses.”

Table 3: Motion Commands

command	meaning	try
`k`, `j`	up, down	`7k`
`h`, `l`	left, right	`3l`
`t`, `i`	until, inside	`dt`, `di"`
`w`, `b`	word forward, word backward	`y3w`, `9b`
`$`, `0`	until the end, beginning of line	`y$`
`)`, `(`	beginning of the next, previous sentence	`d)`
`gg`, `G`	beginning and end of document	`dG`
`n`, `N`	next, previous (used with `/`)	

Motion commands often take a number as an argument for how many times the motion should be repeated. Try this in your file: evoke `/the` to find all instances of the definite article. Then type `5n` to move to the fifth "the" in the document (assuming you have that many). Finally, it may be useful to review some of the major modes:

Table 4: Vim Modes

mode	key	purpose
Normal	`Esc`	You should escape to normal mode always
Command	`:`	You can issue longer commands here like `:help`
Insert	`i`	Your basic typing mode
Visual	`v`	Select and operate on characters
V-Line	`Shift-V`	Select and operate on lines
V-Block	`Ctrl-V`	Select and operate on blocks

To learn more, run the tutorial program `vimtutor`. If you use Linux or MacOS, you already have this program installed, so it's as easy as opening a terminal and typing `vimtutor`. The program takes about 25 minutes to complete, and will familiarize you with all the basics of editing with Vim. There are also lots of great tutorials online, including the adventure game Vim Adventures. Just like with any language, we suggest aspiring Vimmers master one Vim idiom at a time. Once you are fluent in basics, you can move on to advanced topics like buffers, folds, markers, search and replace, and file exploration.

Vim for Prose

The barrier to entry into Vim for the non-programmer is its code-centric "out-of-the-box" defaults. We therefore intend to discuss the issue of setting up Vim for prose editing in particular in this section. In the next section, we will

conclude by suggesting a few "quality of life" improvements that go beyond basic functionality.

Because Vim is a toolkit for building a better editor, we need to do some work to customize it for writing prose. To do this, we can edit Vim's configuration file, `.vimrc`, found in your home directory. You can edit it in vim by typing `vim ~/.vimrc` into your terminal program, or by double-clicking on the file[2]. Here are a few common settings:

```
" This turns off backwards-compatibility with the `vi` editor, which we won't need.
set nocompatible

" This tells Vim not to format long lines of prose as if they were code.
setlocal formatoptions=l

" Turn on word wrapping, to avoid typing really long lines that extend
" horizontally.
set wrap

" Break lines on words, instead of characters, which looks better for prose.
set linebreak

" Use `j` and `k` to move within wrapped lines, in addition to ordinary lines.
map j gj
map k gk
```

Note the use of double quotes for comments, which are ignored on startup. For more useful settings, take a look at some of the `.vimrc`s that other Vim users have posted to GitHub. A recent GitHub search for 'vimrc' returned over seven thousand examples. The authors' configuration files can be found here and here. Your `.vimrc` file will soon become your prized possession. It is what makes Vim uniquely yours. We recommend that you do not add any lines there that you don't understand. A good `.vimrc` configuration file will be well annotated by the owner.

Spell Checking

Vim has a built in spell-checking mode that you can activate by evoking `:set spell`. Since this is a long command to type, we can create a shorcut for it by adding this to our `.vimrc`:

```
map <F6> :set spell<CR>
```

[2]You may need to enable "show hidden files" in the file explorer or finder of your choice.

With spell mode enabled, here are some common commands you can use in normal mode:

Table 5: Spell Checking Commands

command	meaning
`]s`	move to the next misspelled word
`[s`	move to the previous misspelled word
`z=`	offer spelling suggestions for the current word
`zg`	mark the current word as "good" (`g`) or correctly spelled
`zw`	mark the current word as "wrong" (`w`) or incorrectly spelled

Copy and Paste from Outside of Vim

Vim features a very powerful copy-and-paste system. While most word processors can only copy and paste one thing at a time, Vim can store copied text in about forty separate registers. It does not, unfortunately, share the contents of these registers with the system clipboard by default, so copying and pasting from other applications takes some configuration. Add these lines to your vimrc:

```
" Better copy & paste, needs a vim version newer than 7.3.74.
set clipboard=unnamedplus
```

Common Plugins

There are hundreds, if not thousands of plugins that have been written to extend and improve Vim's already rich functionality. Plugins can be installed manually, by dropping the files in a `.vim` folder, or by using a plugin manager like Vundle. Here are a few that are useful for writing prose:

- goyo: a distraction-free writing mode
- online-thesaurus: looks up the current word in an online thesaurus
- vim-pandoc: converts documents between many different formats, such as between vim-friendly markdown and Microsoft Word .docx

To find more, look through the directory Vim Awesome, which has a list of plugins, sorted by popularity. Although it is tempting to jump into the world of Vim plugins right away, the common wisdom suggests deferring until intermediate to advanced proficiency is reached. The writers found it useful to tackle one customization at a time. Vim is a remarkably flexible and deep tool that can grow with the user's expertise. It is free and open source, which means that with enough users, we further tailor it to the needs of our academic community.