

Beste de savoir

Petite leçon de Vim : Apprendre à gérer
ses motions

29 octobre 2022

Table des matières

Introduction	1
1. Anatomie d'une commande.	2
2. En haut, en bas, à gauche, à droite...	3
3. Opérateurs sur objets.	5
4. Dessine-moi un opérateur.	5
Conclusion	6

Introduction

Vous êtes-vous déjà demandé pourquoi un éditeur tel que Vim, à l'apparence si austère dans sa configuration par défaut, jouissait d'une popularité importante sur les Internets ? Ou peut-être qu'en assistant à une de ces présentations où quelqu'un utilise Vim et où l'on voit s'afficher chacune des touches appuyées vous avez pensé qu'il fallait être fou pour être capable de mémoriser autant de [combinaisons obscures](#) ? Ou encore peut-être êtes-vous tombés complètement par hasard sur cet article et cherchez de quoi occuper vos dix prochaines minutes.

Quoiqu'il en soit, vous êtes probablement plutôt novices en ce qui concerne l'utilisation de Vim, et vous avez envie d'apprendre à l'utiliser mieux. C'est parfait car vous êtes au bon endroit. Attrapez une tasse de thé, ouvrez Vim¹ et préparez-vous à parler d'*operators*, *motions* et *text-objects*.



Ce billet a été initialement rédigé pour la newsletter [FedeRez](#) de Octobre 2022 (non disponible publiquement à ma connaissance).

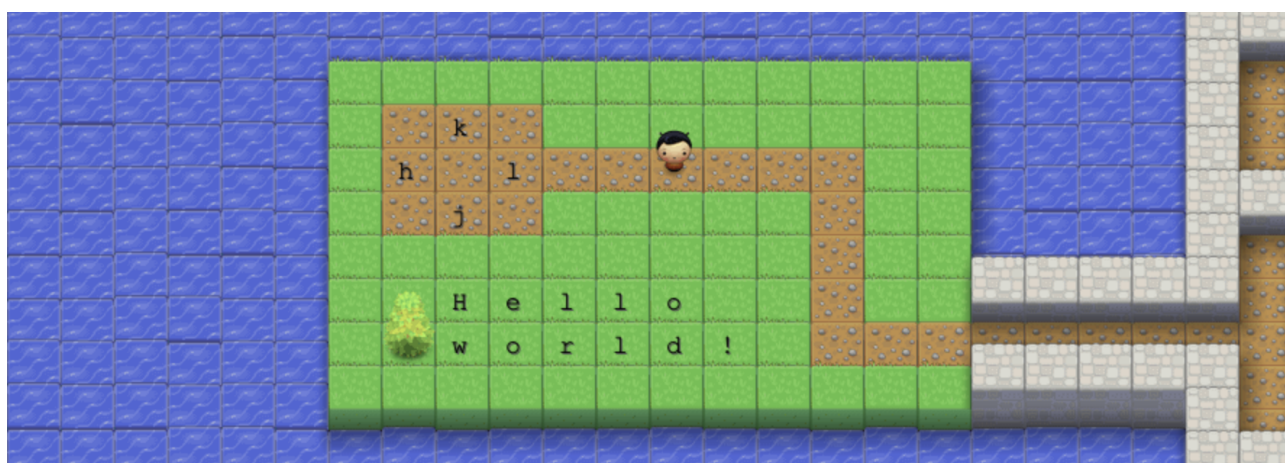


FIGURE 0.1. – Capture d'écran du jeu vim-adventures par [Doron Linder](#) ↗

1. Anatomie d'une commande.

Vous le savez probablement, Vim possède un certain nombre de *modes*. Pour être exact il existe sept modes dits basiques et six modes complémentaires qui sont des variantes des modes basiques (`:help vim-modes`). Dans la pratique, vous avez probablement utilisé les modes *Normal*, *Visual*, *Insert* et *Command-line*. Rapidement (parce que ce n'est pas le sujet de cet article), le mode *Normal* est le mode dans lequel on passe le plus de temps et qui est actif à l'ouverture de Vim ; on peut y lancer toutes sortes de commandes. On entre dans le mode *Visual* et ses sous-modes (*character*, *line* et *block*) en tapant `v`, `V` ou `Ctrl+V`. Le mode *Insert* est actif depuis le mode *Normal* en tapant `i`, et permet l'édition de texte comme dans des éditeurs plus classiques tels que nano ou VS Code. Finalement, le mode *Command-line* est activé depuis le mode *Normal* en tapant «`:`» (oui, oui, c'est par là qu'on passe pour quitter Vim avec `:q`).

Les opérateurs sont lancés en mode *Normal* ou en mode *Visual*, mais pour l'instant, nous allons nous concentrer sur le mode *Normal*. Les opérateurs de base sont capables de modifier, supprimer du texte ou encore le placer dans un registre. Ceux que vous utiliserez probablement le plus sont `c`, pour *change*, `d`, pour *delete* et `y`, pour *yank*.

Ces opérateurs correspondent en réalité à des familles d'opérateurs qui peuvent agir sur des registres (`:help registers`). Par défaut, ils agissent sur le registre anonyme, «`''`». Nous allons revenir sur la syntaxe des commandes d'opérateurs dans un petit instant, mais en premier lieu, on peut s'intéresser à la forme simple qui fait agir ces opérateurs sur la ligne courante. Ainsi, si vous tapez `dd` en mode normal, Vim supprime votre ligne et son contenu est transféré dans le registre anonyme «`''`». Vous pouvez coller le contenu de ce registre anonyme en utilisant la commande `p`. Entrer `cc` aura le même effet que l'opérateur `dd`, excepté que vous vous retrouverez en mode insertion à la fin de l'exécution, vous permettant de changer la ligne courante. Enfin, la commande `yy` va simplement copier la ligne actuelle dans le registre anonyme sans altérer le texte.

Nous avons seulement utilisé le registre anonyme, mais il existe 47 registres dans Vim! Les plus utiles sont probablement les 26 registres nommés ("`a`" à "`z`"). Ainsi, si je sais que je vais répéter plusieurs fois la même ligne, je peux l'écrire une seule fois, puis la copier dans le registre "`a`" en utilisant l'opérateur "`ayy`". Je pourrai ensuite la coller à volonté un utilisant la commande "`ap`". Les variantes de `c` et `d` fonctionnent également sur le même principe.

Il existe 16 opérateurs, que l'on retrouve en cherchant `operator` dans l'aide. Outre le fameux trio `c`, `d` et `y` que nous avons déjà rencontré, vous aurez sûrement l'utilité de «`!`» pour filtrer du texte *via* un programme externe. Ainsi, vous pouvez notamment invoquer `grep` sur les lignes de votre *buffer*¹, ou faire des opérations compliquées sur des données JSON avec `jq`. Une astuce utile est de créer un alias vers `!$SHELL`. Ainsi, en l'utilisant, on peut remplacer une ligne par le résultat de son exécution par votre shell. Par exemple `ls grep .dat` sera remplacé par *n* lignes correspondant aux *n* noms de fichiers du répertoire courant contenant `.dat`.

1. Ou NeoVim, on est en 2022 et Vimscript est à Lua ce qu'un coup de matraque est au parfum d'une marguerite.

2. En haut, en bas, à gauche, à droite...

On peut également mentionner les opérateurs `gq` et `=` qui respectivement formate et indente un buffer, et dont le comportement peut être paramétré finement dans vos fichiers de configuration *via* des réglages donnés dans leurs documentations respectives. Enfin, il y a des opérateurs étonnamment spécifiques tel que `g?`, qui permet de transformer le texte *via* l'algorithme ROT13².

J'ai parlé un peu plus haut de syntaxe. Nous pouvons déjà explorer un peu le premier élément de syntaxe lié aux opérateurs: la répétition. On peut utiliser les versions simplifiées des opérateurs agissant sur une seule ligne (`dd`, `yy`, `cc` et consorts) pour agir sur plusieurs lignes! Ainsi, si je souhaite couper trois lignes, je peux simplement taper `3dd`. Le comportement sur les registres est celui attendu: on aura bien trois lignes dans le registre anonyme. Si je souhaite utiliser un autre registre, comme `"b`, alors je peux simplement envoyer la commande `3"bdd`.

2. En haut, en bas, à gauche, à droite...

... à la dernière lettre du troisième mot après le curseur.

Que vous trouviez la syntaxe de répétition puissante ou gadget, restez encore un peu, parce que les *motions* vont la rendre encore plus pratique (et les *text-objects* de la partie suivante seront également très intéressants).

Si vous avez déjà ouvert Vim, vous savez probablement que l'on peut se déplacer dans le document avec les flèches directionnelles. En pratique, on préfère utiliser les touches `h`, `j`, `k` et `l`, pour respectivement gauche, bas, haut et droite, car elles sont plus facilement accessibles et évitent de déplacer sa main lorsque l'on tape. De plus, sur de nombreux claviers, la touche `j` dispose d'un marquage en relief qui permet de la retrouver facilement si l'on se décale. Cette utilisation de `h`, `j`, `k` et `l` peut sembler contre intuitive, mais en pratique on s'y habitue assez facilement avec un peu de rigueur. Personnellement, j'ai réussi à m'habituer à les utiliser en désactivant volontairement les touches directionnelles dans mon fichier de configuration; en une semaine de pratique, utiliser les nouvelles touches me paraissait tout à fait naturel.

Tout comme les opérateurs, les mouvements peuvent être multipliés. Ainsi, en entrant `10j`, le curseur sera déplacé de dix lignes vers le bas. Il est d'ailleurs courant de créer dans sa configuration un alias entre la majuscule d'une lettre directionnelle et sa répétition dix fois. Ainsi, entrer `J` sera équivalent à entrer `10j`, ce qui permet de faire défiler rapidement un fichier¹.

Il existe une myriade d'opérateurs de mouvement. La documentation de Vim les classe en mouvements latéraux, verticaux, mouvements de mots et mouvements divers.

Vous serez probablement intéressés par les opérateurs `t` (*to*) et `f` (*forward*) qui permettent d'avancer jusqu'à un caractère donné. Par exemple, `tm` permet d'avancer juste avant le prochain caractère «m». En le combinant avec l'opérateur `c`, formant la phrase `ctm`, on peut ainsi changer le contenu d'une ligne jusqu'au prochain caractère «m». Ces deux mouvements possèdent des équivalents en majuscule (`T` et `F`) qui déplacent vers l'arrière.

1. Rappel: dans Vim, un *buffer* est un fichier chargé en mémoire pour l'édition. On le voit au travers d'une *window*.

2. En cherchant un peu, il semblerait que ce soit un [héritage des premiers temps de l'internet](#) `☞`, avant la démocratisation de l'usage des balises *spoiler*. Mais peut-être que quelqu'un de mieux versé que moi dans les arts sombres a plus de choses à raconter sur le sujet!

2. En haut, en bas, à gauche, à droite...

Les mouvements latéraux `0`, `^` et `$` permettent respectivement de se déplacer au premier caractère de la ligne, au premier caractère non vide et au dernier caractère de la ligne. Les mouvements verticaux `gg` et `G` sont probablement les plus utiles. Par défaut, ils permettent de se déplacer respectivement à la première et la dernière ligne du *buffer*. En y ajoutant un multiplicateur *n*, on peut également se déplacer à la ligne *n*.

Les mouvements de mots, comme le nom le laisse penser, permettent de se déplacer entre les mots. Ainsi, `w`, `e`, `b` et `ge` permettent de se déplacer jusqu'au premier mot qui suit (*word*), à la fin du mot actuel (*end*), au mot précédent (*backward*) ou à la fin du mot précédent. Ces mouvements peuvent se coupler à un multiplicateur. Ainsi, `d3w` va supprimer les trois prochains mots suivant le curseur. Vim fait la différence entre les mots dits `words` et ceux dits `WORDS`, les premiers n'incluant pas la ponctuation, tandis que les seconds oui. Les mouvements de mots que j'ai présentés agissent sur des `words`, mais ils possèdent des équivalents pour les `WORDS`: `W`, `E`, `B` et `gE`. Je vous invite à consulter la documentation dédiée (`:help word` et `:help WORD`). Enfin, parmi les opérateurs divers, mon préféré est `%`, qui permet de se déplacer jusqu'au caractère ouvrant/fermant correspondant. Par exemple, si mon curseur se trouve sur une parenthèse ouvrante, `%` le déplacera jusqu'à la parenthèse fermante correspondante. Une fois de plus, consultez la documentation correspondante!

Nous avons déjà commencé à le voir, Vim propose une syntaxe permettant de combiner les opérateurs de la première partie avec des mouvements. Elle se présente ainsi dans sa forme générale: `{multiplicateur}{opérateur}{mouvement}`. Ainsi, je peux supprimer les trois prochains mots en utilisant la commande `3dw`, qui est équivalente à `d3w`. On peut donc créer des commandes plutôt complexes. Par exemple, supposons qu'un *buffer* contienne la ligne suivante (avec le curseur sur le premier caractère):

```
1 ma_fonction(a, b+1, x)
```

Je peux placer les deux premiers arguments de l'appel à la fonction dans le registre `"a` en entrant `«2w"ay2t,»`. Non, mon chat n'a pas marché sur mon clavier². Vous êtes maintenant suffisamment équipé pour décortiquer cette commande! Vous pouvez remarquer qu'il s'agit en réalité de deux commandes. La première, `2w`, déplace le curseur de deux mots vers la droite. Il se trouve alors sur le caractère `a`. On peut alors lancer la seconde commande: `"ay2t,`. Elle est composée de l'opérateur `"ay`, qui copie l'objet de la commande vers le registre `"a`, et du mouvement `2t,` qui est une répétition du déplacement vers la prochaine virgule, sans l'inclure. L'opérateur va donc agir sur la chaîne de caractères `«a, b+1»`. Cette formulation est de plus suffisamment générique pour pouvoir être utilisée dans une macro, car elle ne dépend pas de la forme que prennent les deux premiers arguments de la fonction.

Enfin, je souhaite mentionner ici la possibilité de se déplacer jusqu'à une marque. Sans entrer dans les détails (vous pouvez consulter `:help mark-motions` pour cela), certaines commandes ou actions peuvent déposer des marques dans un *buffer*, et vous pouvez vous déplacer jusqu'à elles. Par exemple, si vous sélectionnez du texte en mode *Visual*, Vim va déposer les marques `<` et `>` respectivement au début et à la fin de la sélection. Vous pouvez par exemple supprimer jusqu'à la fin de la précédente sélection du mode *Visual* en entrant `d>`. Le mode *Command-line* vous permet également de travailler sur des *ranges*, qui peuvent entre autre être définis par des

3. Opérateurs sur objets.

marques. Ainsi, entrer «`:<,>d`» va supprimer le texte de la sélection du mode *Visual*. Je vous laisse lire la documentation pour en apprendre plus sur les *ranges*: `:help range`.

3. Opérateurs sur objets.

Si vous avez commencé à jouer avec Vim en lisant la partie précédente, il est possible que vous ayez constaté une limitation. Supposons que le curseur se trouve au milieu d'un mot. Si j'entre la commande `dw` Vim va supprimer tous les caractères jusqu'au début du prochain mot, sans se soucier des premiers caractères du mot. En effet, on a appliqué l'opérateur `d` sur le mouvement qui nous emmène au prochain mot. Les objets textuels répondent entre autre à cette problématique. Au lieu d'appliquer un opérateur sur un mouvement, ils permettent d'agir sur des objets. Il existe un bon nombre d'objets, tels que `aw` (*a word*), `as` (*a sentence*) ou `ap` (*a paragraph*) qui agissent respectivement sur un mot, une phrase ou un paragraphe. Vous pouvez ainsi supprimer le mot courant en entrant `daw` (*delete a word*). Les objets textuels commençant par `a` sélectionnent l'objet en incluant les espaces. Ils disposent aussi de cousins commençant par `i` (pour *inner*) qui sélectionnent l'intérieur de l'objet, sans les espaces, ou juste l'espace (si le curseur se trouve sur une espace).

Comme avec les mouvements, il est possible d'affecter les objets avec un multiplicateur placé avant. Ainsi, vous pouvez supprimer les trois prochains mots en incluant celui sur lequel se trouve le curseur en entrant `d3aw`.

Il existe d'autres *text-objects* et vous pouvez les retrouver dans la documentation (`:help text-objects`, vous commencez à connaître la chanson). Parmi les plus utiles, `i"` permet de sélectionner la prochaine chaîne de caractères qui se trouve entre des guillemets doubles, et a de nombreux cousins tels que `a(`, `i` ` [` ou `i`.

Certains plugins définissent leurs propres objets textuels, par exemple [julia-vim](#)¹ définit les objets `aj` et `ij` permettant de sélectionner le bloc Julia actuel. Un autre exemple est [vim-tex](#)², qui définit des objets pour les commandes, les environnements ou encore les sections, ainsi qu'un certain nombre de mouvements. Cela permet de travailler directement avec les blocs sémantiques qui ont du sens dans le contexte de l'édition. Ils rendent ainsi Vim plus intelligent en vous permettant d'appliquer ses opérateurs sur vos propres objets.

4. Dessine-moi un opérateur.

Pour finir cet article par une petite séance de travaux pratiques, jouons avec l'opérateur `g@`. Si vous êtes allé lire la documentation de cet opérateur avant d'avoir terminé la lecture des parties précédentes¹, vous avez constaté que `g@` va appeler la fonction définie dans `operatorfunc`. Pour illustrer le fonctionnement de cet opérateur et la gloire de l'autrice du [chapolibot](#)², écrivons un opérateur qui rend sa-victime votre texte~ `^w^ ^w^` un p-peuwu uh... plus magique.

1. Attention toutefois, mon vaillant relecteur me signale qu'en agissant ainsi, on vient remplacer la commande native de Vim pour joindre des lignes.

2. Car je n'ai pas de chat.

1. Vous pensiez vraiment que j'allais faire un article sans mentionner notre seigneur et sauveur?

Conclusion

Vous trouverez [ici](#) une version simplifiée du [chapolibot](#), qui forme un programme qui attend une entrée et la passe à la moulinette `uwuify`. Ensuite nous allons écrire une fonction en vimscript qui ira chercher le texte délimité par les marques et (`:help :marks`), et l'enverra au programme Rust compilé².

La fonction peut prendre la forme suivante, et vous pouvez la voir en action [ici](#).

```
1 ma_fonction(a, b+1, x)
```

Si on oublie la syntaxe un peu pénible de Vimscript, cette fonction est relativement simple. Puisque l'opérateur `g@` vient placer les marques et au début et à la fin du texte sur lequel on va opérer, il suffit de récupérer la position des marques avec `getpos`, puis de lire le texte qui se trouve entre les marques avec `getline`. Il y a un peu de travail à faire pour gérer le cas où la première ou la dernière ligne n'est pas complète, puis on peut envoyer le tout au programme `uwuify` et stocker le résultat dans le registre anonyme. Enfin, on place le curseur au début du texte sélectionné, et on effectue une commande en mode *Normal*, qui sera la dernière commande analysée pour illustrer cet article.

À ce stade, le curseur se trouve au début du texte que l'on souhaite remplacer, et le texte qui doit être inséré est dans le registre anonyme. La première partie de la commande (`d]`) se charge de supprimer le texte d'entrée. La seconde partie ("`0P`") colle le contenu du registre "`0`" juste avant le curseur. Le registre "`0`" est défini lorsque l'on écrit dans le registre anonyme. On utilise le registre "`0`" et non le registre anonyme parce que le résultat de la commande ne se trouve plus dans le registre anonyme: son contenu a été modifié par la première partie de notre commande. La dernière partie de la commande (`gq]`) formate le texte que nous venons de coller avec l'opérateur `gq`. Cela permet par exemple de respecter les contraintes de nombre maximal de caractères par lignes. L'opérateur de mouvement qui lui est associé est `]`, car la marque `]` se trouve toujours après le texte que l'on vient de coller, et notre curseur a été déplacé au début du texte par l'opération de collage.



Conclusion

Ouf ! vous voici au bout de cet article un *poil* dense. Si vous vous demandez comment il est possible de retenir tout ceci, la réponse est simple : la pratique. Je peux vous conseiller de choisir quelques exemples de commandes et de cas d'utilisation qui vous semblent particulièrement intéressants pour votre travail, et de vous forcer à les comprendre en détail et à les appliquer dans votre vie quotidienne. Cela vous semblera au début un peu plus long, mais vous devriez rapidement gagner en efficacité. Il ne faut pas hésiter parfois à recommencer une action pour le refaire de manière plus élégante grâce aux commandes Vim. Le fait de se forcer à comprendre les commandes que vous employez facilitera également l'apprentissage de nouvelles commandes, car notre éditeur essaie de conserver une certaine logique entre ses différents opérateurs, mouvements et objets.

1. Bande de petits garnements

2. J'ai compilé un binaire [ici](#), mais je ne garantis pas qu'il restera en ligne pour toujours. Le binaire doit être accessible par Vim pendant notre expérience. Pour faire simple placez-le simplement dans le répertoire courant.

Conclusion

Il existe des outils plus amusants pour apprendre Vim, ou comparer sa maîtrise de l'outil avec d'autres utilisateurs. Par exemple, [vim-adventures](#) , dont une capture d'écran illustre cet article, est un jeu à la Zelda créé par Doron Linder et qui a pour but de vous apprendre les bases de l'éditeur de manière ludique. Le premier niveau est gratuit, mais attention c'est addictif! Un second jeu, qui va peut-être plaire à ceux qui aiment la compétition, est [vimgold](#) . Ici, le but est d'opérer une modification sur un *buffer* avec le moins d'appuis clavier possible. On peut ensuite partager son score sur le site web pour chacun des 543 challenges existant.

Enfin, on ne le mentionnera jamais assez, mais la documentation de Vim est très complète. Je ne peux que vous conseiller de vous perdre dans la lecture qui s'offre à vous en entrant la commande `:help`.