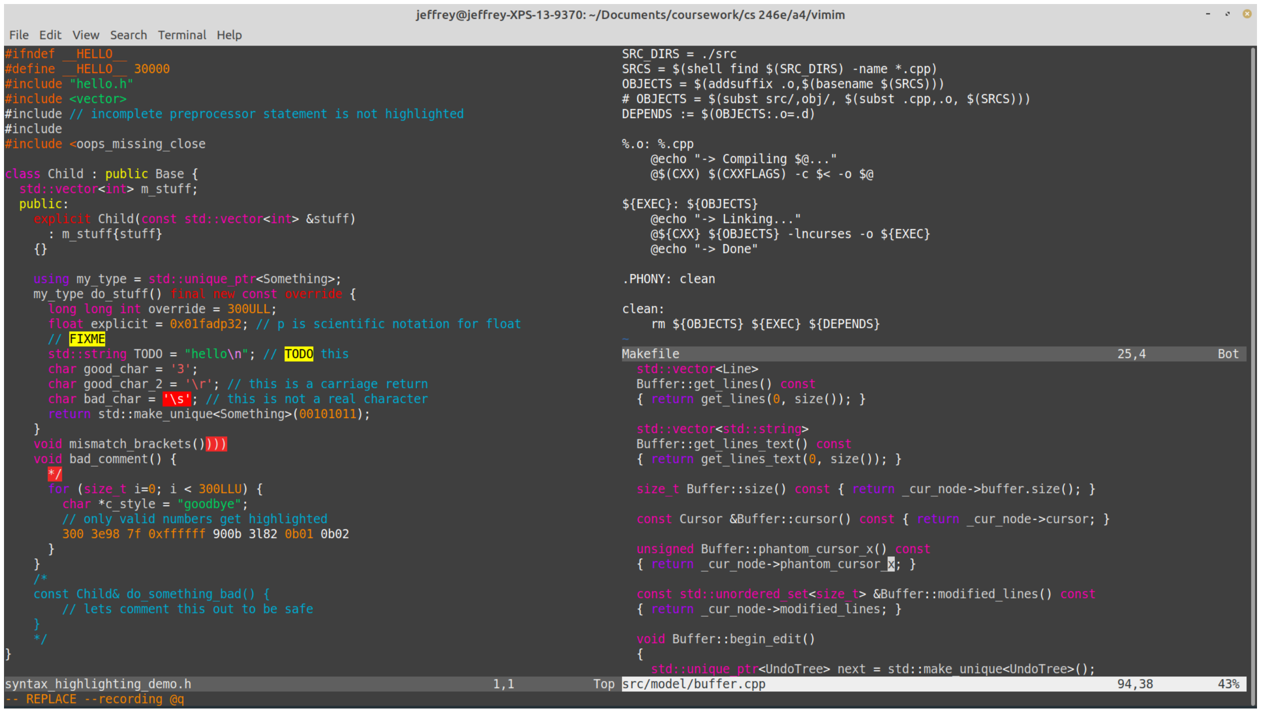
Vimim Design Document

Ayon Bakshi (`a4bakshi`) and Jeffrey Liu (`jy39liu`)

Submitted on December 3, 2019 for CS 246e (Prof. Lushman)

1. Introduction

We present an implementation of the Vm text editor. Our implementation (henceforth known as Vimim, or Vim IMproved) supports the 55 required commands, as well as display splits, a fully functional undo-redo tree, macros and recording, syntax highlighting, regex-based searching, and more. We also exclusively use vectors and smart pointers, and therefore avoid the use of `new` and `delete` anywhere whilst never leaking memory. Finally, our codebase was designed with organization and extensibility in mind, and so it follows good OOP principles, design patterns, and is well documented.



jeffrey@jeffrey-XPS-13-9370: ~/Documents/coursework/cs 246e/a4/vimim
File Edit View Search Terminal Help

```
#ifndef __HELLO__
#define __HELLO__ 30000
#include "hello.h"
#include <vector>
#include // incomplete preprocessor statement is not highlighted
#include
#include <oops_missing_close

class Child : public Base {
  std::vector<int> m_stuff;
  public:
    explicit Child(const std::vector<int> &stuff)
      : m_stuff{stuff}
    {}

    using my_type = std::unique_ptr<Something>;
    my_type do_stuff() final new const override {
      long long int override = 300ULL;
      float explicit = 0x01fadp32; // p is scientific notation for float
      // FIXME
      std::string TODO = "hello\n"; // TODO this
      char good_char = '3';
      char good_char_2 = '\r'; // this is a carriage return
      char bad_char = '\s'; // this is not a real character
      return std::make_unique<Something>(00101011);
    }
    void mismatch_brackets()))
    void bad_comment() {
      */
      for (size_t i=0; i < 300LLU) {
        char *c_style = "goodbye";
        // only valid numbers get highlighted
        300 3e98 7f 0xffffff 900b 3l82 0b01 0b02
      }
    }
    /*
    const Child& do_something_bad() {
        // lets comment this out to be safe
    }
    */
}
syntax_highlighting_demo.h                                1,1          Top
-- REPLACE --recording @q
```

```
SRC_DIRS = ./src
SRCS = $(shell find $(SRC_DIRS) -name *.cpp)
OBJECTS = $(addsuffix .o,$(basename $(SRCS)))
# OBJECTS = $(subst src/,obj/, $(subst .cpp,.o, $(SRCS)))
DEPENDS := $(OBJECTS:.o=.d)

%.o: %.cpp
    @echo "-> Compiling $@..."
    @$(CXX) $(CXXFLAGS) -c $< -o $@

${EXEC}: ${OBJECTS}
    @echo "-> Linking..."
    @${CXX} ${OBJECTS} -lncurses -o ${EXEC}
    @echo "-> Done"

.PHONY: clean

clean:
    rm ${OBJECTS} ${EXEC} ${DEPENDS}
~
Makefile                                                  25,4         Bot
  std::vector<Line>
  Buffer::get_lines() const
  { return get_lines(0, size()); }

  std::vector<std::string>
  Buffer::get_lines_text() const
  { return get_lines_text(0, size()); }

  size_t Buffer::size() const { return _cur_node->buffer.size(); }

  const Cursor &Buffer::cursor() const { return _cur_node->cursor; }

  unsigned Buffer::phantom_cursor_x() const
  { return _cur_node->phantom_cursor_x; }

  const std::unordered_set<size_t> &Buffer::modified_lines() const
  { return _cur_node->modified_lines; }

  void Buffer::begin_edit()
  {
    std::unique_ptr<UndoTree> next = std::make_unique<UndoTree>();
src/model/buffer.cpp                                      94,38        43%
```

Figure I. Syntax highlighting, splits, replace mode, recording, what else could you possibly want?

2. Overview

We followed the classic MVC design pattern to organize our project at a high level. Each of the model, view, and controller are owned by a Vimim application, and they communicate with each other by maintaining references.

2.1 Model

Nearly all of our program state was stored in plain old data objects, with the exception of the undo/redo tree and the split tree. These trees do not handle any business logic; they contain only getters, setters, and simple data-structure logic such insertions and deletions, and are managed by their respective engines.

2.2 View

Our view objects were very simple. Their state consists only of their current positions on the screen and offsets in the file they are rendering. Their responsibilities, on the other hand, are quite diverse. They are responsible for converting text and colour data from our models into ncurses function calls. To help facilitate this, we use ncurses wrappers to hide most of the details (since the less C-style libraries we use the better). Our view objects are also responsible for parsing and cleaning user input for the controller later downstream. They need to keep track of the focused split, as well as any highlighted text, the cursor position, etc., so that our controller does not need to worry about the small details (eg., checking that the cursor is in bounds, resizing splits, and so on).

2.3 Controller

We separate the controller logic into two subsections: engines (which are stateless collections of functions which act on parts of our model) and commands (which map keystrokes to engine calls). This allows us to reuse code between similar commands. For example, the commands `i` and `a` both switch from normal mode to insert mode, so this logic can be separated into its own engine function.

2.3.1 Engines

There were five engines in our code, each supporting a different class of functions: editing, movement, recording, searching, and splits (and each discussed later). These engines were singleton objects which do not store state. Instead, they were passed in a relevant window/text panel to work on. This was better than the alternative of attaching engines to individual panels or windows (which would mean they don’t need to be passed in a window/text panel) since there shouldn’t be a ‘has-a’ relationship between panels and engines.

2.3.2 Commands

Our primary objective with our command infrastructure was to reduce code reuse between the 80+ commands in our final program. Many of these commands share common features with each other, for example, there are several commands which all edit text in the buffer and so need to increment the current undo-tree frame, there are many commands which repeat in the same way (i.e. when we type `[count][cmd]`), some commands may begin recording as soon as they are typed, and so on. These features all “decorate” the `BaseCommand` class. We explore the specifics later in section 4.1.

3. UML

See the attached (in Marmoset).

4. Design

We will only discuss design challenges in the base part of the project. We explore the design implications of the extra credit features in Section 7. Note that extensions of base features, such as undo/redo, regex searching, and similar are also discussed in Section 7 and not here.

4.1 Commands

We use several object-orientated design patterns in order to keep our commands extensible while minimizing code reuse. Most notably, we have several flags available in `BaseCommand` to add common features to core features, inspired by the decorator pattern (where any combination of these flags can be enabled or disabled). For example, there is a lot of shared boilerplate code between all the commands which run before and after `can_run` and `run` are called, including code needed by undoable commands, commands which are repeatable by specifying a `[count]` parameter, and so on. The base command class also handles all of the parsing of commands, recording key input for commands which may need to be recorded (for example on entering insert mode we need to start recording in order to playback with `.`), navigating the split tree to get the current focused split, and so on. All of these features can be enabled/disabled by enabling/extending certain flags. As a remark, instead of using inheritance and the decorator pattern to enable and disable these features, we used flags, since for each command it is known at compile time which flags they would like to enable.

By having our base class do most of the heavy lifting, our child classes (the individual commands) are very lightweight. They only need to implement two functions: `can_run` and `run`. The `can_run` function is called when the controller is trying to decide which command was executed. It accepts a command string and returns a response signal, which is one of `SUCCESS`, `WAITING`, or `FAILED`. Depending on the response, the controller will either call the `run` function, wait for more user input, or continue and try the next command in the list of commands. The `run` function is usually a very lightweight function which makes a series of calls to our engine objects to administer the business logic.

Due to space concerns, we will not go in depth into each individual command in this document. Indeed, the bulk of the commands are movement commands, which were really quite trivial to implement. Some of the more elaborate implementations follow below.

4.2 Copy, Cut, and Pasting

We maintain a static map from characters (registers) to vectors of strings. The copy and cut commands all behave in the same way to write to this map. Parsing a command of the form `d_`, where `_` is any movement command, is somewhat tricky. We accomplish this by creating a temporary undo frame in our undo tree, running `_` through our controller, and examining the result. By checking the difference in cursor positions before and after running `_`, we know exactly if `_` was a movement command, and how it moved the cursor. Otherwise, if the buffer is modified in any way, we can pop the undo frame from our tree and it would be as if `_` was never ran in the first place.

Pasting on the other hand is simple. We simply read from corresponding register in our map and insert the lines into the buffer.

4.3 Status Bar

We reuse the text panel object from our display split hierarchy to implement the status bar. This lets us use all the buffer tools we've established earlier, including typing in the footer (for example, in command mode), colouring and highlighting the text, and so on. By reusing our display split tree (more on this in Section 7.1.), we automatically resize the split if the command text overflows. This status bar is where we propagate error messages to the user, display file information, cursor position, which mode we are in, etc.

4.4 Modes

Modes, in contrast to commands and engines, are very simple in our framework. A mode is simply a list of commands which can be executed while in that mode. When requested by the window, they simply search through their respective lists until they find a command who's `can_run` request doesn't return a `FAILED` signal.

4.5 Movement Engine

The movement engine is unremarkable. It provides an interface which allows us to move the cursor in all directions, to a specific line number, to the next word, and so on. We do need to keep track of a phantom *x* cursor position, as if the cursor *x* is greater than the length of the line, it must be displayed at the end of the line. Important to note: moving to the end of the line moves the cursor to positive infinity (or as close as we can get with unsigned integers) instead of the end of the current line to mimic the behavior of Vim.

4.6 Edit Engine

The implementation of text insertion and editing is also quite straightforward. Displayable characters (which we define as spaces, tabs, numbers, letters, and symbols) are inserted straight into our buffer. We take some care to allow the cursor to exceed the length of the line by one in insert mode without array index issues.

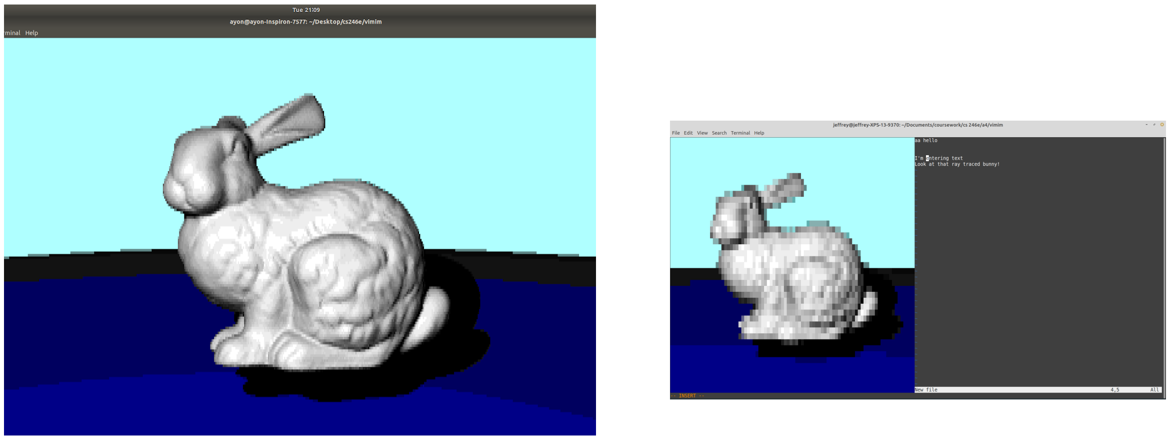
4.7 Replace Mode

The main challenge in replace mode is the finicky backspace key. The backspace key behaves as an "undo" for the most recently replaced character. To keep track of this, we store a stack of these characters and push into the stack as we replace characters in the buffer. This stack is cleared whenever an arrow or delete key is pressed. A bit more care was needed to handle newlines, but in the end we were able to reuse a lot of code from insert mode with the addition of this stack.

5. Resilience to Change

Actions speak louder than words. There was internal debate about which features should be included or not in the final implementation, and so our project design is flexible to support this discourse. There is no coupling between the different commands and engines, and so new feature implementations can be done independently of one another and painlessly. For larger feature requests, we fall back to the abstractions. We follow the dependency inversion as often as possible to facilitate this.

To prove a point, we incorporated a functional ray-tracer (written by the first author) into Vimim. Since Vimim chooses a renderer based on a file extension, we were able to create an extended renderer to handle ray-tracing. Consequently, Vimim is able to create a scene with an `.obj` file and display a ray-traced image of that scene in the console. Without any other modifications, our splits, command mode, and so on are still completely functional.


Tue 21:09
ayon@ayon-Inspiron-7577: ~/Desktop/cs246e/vimim
minal Help
jeffrey@jeffrey-XPS-13-9370: ~/Documents/coursework/cs 246e/a4/vimim
File Edit View Search Terminal Help

```
aa hello


I'm entering text
Look at that ray traced bunny!
New file                                   4,5        All
-- INSERT --
```

Figure II. A fully functional ray tracer in action. In [right] we see that our display splits work well with the ray tracer.

6. Answers to Questions

6.1 Switching Between Files

Although this project does not require you to support having more than one file open at once, and to be able to switch back and forth between them, what would it take to support this feature?

We actually do support it (see Section 7.1 for implementation details). In the end, we did not have to change much code to get display splits working, despite it being one of the last features we implemented. This was because our commands operate on the file text buffer instead of the window, and so it was easy to transition into having multiple buffers open without the need of duplicating windows.

6.2 Read-Only Files

If a document's write permission bit is not set, a program cannot modify it. If you open a read-only file in vim, we could imagine two options: either edits to the file are not allowed, or they are allowed (with a warning), but saves are not allowed unless you save with a new filename. What would it take to support either or both of these behaviours?

First, we would add permission values in our file structure. For example, we could add a "editable" and "savable" field. Since the logic for each command is contained within a command object, we could modify the load/save/insert commands to check these permission values and act accordingly. We can handle the two possibilities in the following ways:

If edits to the file are not allowed, we can disable the setter for the internal file buffer using the "editable" field. The insert commands will still run, but will not be able to modify the file buffer. In addition, we could throw an exception from the command (that is caught by the command caller) and display exception information in the footer.

If file edits are allowed, we can keep the file buffer setter enabled, but have the insert commands throw an exception to its caller after performing their action. The caller can then display the warning message from the exception. When the user wants to save, we can enforce that they use a different file name by checking the “savable” field.

If we wanted to handle both of these cases, we could ask the user which option they prefer upon opening a read-only file. If they wanted to edit, then they would be forced to save with a different name. If they wanted to open it in read-only mode, the file buffer would be disabled.

7. Extra Credit Features

7.1 Display Splits

One of the most useful features in Vim is the ability to open multiple files at once with display splits. These display splits form a tree-like structure, and can be made in arbitrary vertical and horizontal formations. We offer a full-featured implementation of display splits which replicates the behavior in Vim. Some notable commands include `<^w-v>` and `<^w-n>` for creating a new split, `<^w-[hjkl]>` for movement between splits, and `:e <file_name>` for opening a file in the current split. As a consequence of our split structure, we are also able to offer dynamic resizing of the window and buffers. For example, Vimim reacts correctly when the terminal is resized, or when the command mode footer overflows to the next line, and in many more similar scenarios.

The logic for the splits is separated between the model and the controller. The model keeps track of cleaning and processing the data of each split. For example, the model is in charge of loading and saving files, the individual buffers in each split, as well as reacting to resize requests by either the view or the controller. The controller on the other hand is responsible for the lifetimes of the splits: their creation, navigation, and destruction. The controller also needs to keep track of which split is currently in focus by the user, and passing this information on to the view. For example, the focused split has a white-highlighted footer, while the other splits have grey-highlighted footers, with the exception of when the split is a singleton in which case the footer is merged with the footer of the window. For implementation details, the file `window_engine.cpp` contains the controller logic, while any files matching the regex `.*_panel.cpp` form the model.

7.2 Undo and Redo Tree

We’ve also implemented undo-redo tree, which is probably our most algorithmically complex feature. Instead of treating undo/redo as a linear progression of changes (for example, with a stack data structure), we considered the undo history as a tree and provide machinery for the user to navigate through this tree as they choose.We differ slightly from Vim in the sense that Vim allows for time based queries while we only allow for tree-style navigation. For example, in Vim you can query `:earlier 10m` to reset the buffer to the state it was in ten minutes ago, while in our vim you can only specify how many edits back (in the sense of inorder traversal) you would like to go. We do manage to achieve a time complexity of $O(1)$ for our undo/redo and $O(m \log n)$

our line editing, insertion, and removal, where *n* is the length of the file in lines, and *m* is the length of the new line. This is close to the best time complexity known to the authors, *O(log*m*log*n*)*, however the constant time overhead would make such a solution impractical (as it requires a BST of BSTs).

To accomplish this, we used implemented our undo tree as a tree of persistent array data structures, which is a well known immutable data structure which supports logarithmic time updating and constant time copying. These persistent arrays are really wrappers of STL balanced binary search trees, which is how they are implemented in the back-end. Our undo and redo operations were then reduced to simply navigating a tree with an iterator. For implementation details, see `buffer.h/cpp`.

7.3 Recording and Macros

We sought to abstract the recording process as much as possible, as many Vimim features can be reduced to recordings and playbacks. For example, the `.` command can be interpreted as a playback of the last undoable command. Our main tool will be `record_engine.h/cpp`, which is an intelligent key event handler. The record engine intercepts all keystrokes before they are interpreted by our commands and records this sequence to an arbitrary number of user-specified queues. We also support event listening, meaning we are able to pass in lambdas which are executed whenever a certain key (or sequence of keys) is pressed. For example, we can add an event listener for the `<ESC>` key which signals whenever we enter normal mode, or we can add an event listener for `q` to stop the current macro recording. With event listeners, it then becomes a trivial task to implement macros.

7.4 Syntax Highlighting

Through trial and research, we discovered that there are many ways to approach syntax highlighting. We decided to go with a regex based approach, similar to the real vim. To facilitate the generation of the syntax rules, we created our own `.vimim` file type. This file is used to define "types" in a language, where "types" are groups of keywords/regular expressions that match (a) token(s) in the language. In addition to top-level types, we also have a recursive "contains" feature. This feature is used when a "type" contains another type, which allowed us to do contextual highlights (such as a "TODO" in a comment, contextual keywords, etc.). Refer to the `cpp.vimim` file in the syn_rules directory for an example. Additionally, our `.vimim` highlighting engine is efficient enough to highlight very large files.

The cpp.vimim file contains a comprehensive list of primitive types, control structures, constants, STL types and more. Additionally, it has regexes to match numeric/string/character literals, preprocessor directives, identifiers, certain syntax errors and more. Since the `.vimim` files are general, creating one for another language is simple and doesn't require recompilation.

Due to limitations with regular expressions, mismatched parentheses/brackets/braces could not be correctly calculated with instructions from the `.vimim` file. Instead, we implemented this feature as an extra rendering step in our syntax highlighting renderer.

7.5 Regex searching

To extend our search feature, we implemented regular expression based search. When searching, the search string is parsed as a regular expression, and a regex iterator yields the next/previous match in the active file. The most recent search is stored and used when required.

7.6 Miscellaneous

We support simple mouse movement as well. We can parse mouse clicks from the user and move the cursor to the location of the click in all modes. However, fancier mouse clicking across splits was not completed.

There were also plenty of other small features that were not part of the project specification, but because of how we organized our project, were very easy to add anyways. These additional commands include `t`, `T`, `*`, `#`, `E`, arrow keys in insert mode, We also consider a few obscure command multiplier interactions of the form `[count][cmd]`, where `[cmd]` is one of `$`, `^`, `0`, `gg`, `G`, `x`, or `X`. Notably, we implemented `[count]i` and its variants, which was another application of our record engine.

8. Final Questions

8.1. What lessons did this project teach you about developing software in teams?

We learned about the importance of using git branches when working on new features. For a brief period, our master branch was in a non-compiling state due to a refactor. During this period, we weren't able to work on anything besides the refactor. This could've been avoided by doing the refactor in a different branch, and then merging once it was complete.

We also learned about the importance of abstract interfaces. Not only do interfaces help a reader to easily see the general design of a project, but it also allows for easier collaboration. Once our header files were written, we were able to divide up the project and work on implementation. This resulted in infrequent and small merge conflicts, as well as an easy way to track our progress.

8.2. What would you have done differently if you had the chance to start over?

We would have experimented with different libraries instead of sticking with ncurses throughout the project. Although capable, certain limitations really hindered our development process (its written in C, colour limitations, it isn't in a namespace, and so on). Perhaps in another project we would aim to emulate GVim (graphical vim) instead of Vim.

There were also many bonus features that we would have liked to implement but did not due to time constraints. In no particular order, these include tabs, visual mode, regex based search and replace, and basic vim script and vimrc in the style of our syntax highlighter. None of these features would have taken very long (except maybe vim script), and we certainly have the infrastructure to support them, but in the end we were unable to complete them.