

Vocabulary Study DX6 Tutorial

Kyle Ross
<kyle@osl.iu.edu>

2006-03-05

Introduction

Vocabulary Study is a small program intended to help people learn general or domain-specific vocabulary through a simple, repetitive process that emphasises the connexion between a word and its definition. Simple, plain-text data files encode the study items—files that can be created with any text editor on any platform. New features such as save-state and resume-state highlight the renewed focus on vocabulary and on providing useful, needed functionality. –and– Finally, Vocabulary Study is open-source, so users can modify it to fit their specific needs.

History

Vocabulary Study started with the first version in 1995 as a command-line Basic program to help me learn Spanish vocabulary in High School. As a famous man once said: “Wow. What a difference a decade makes.”. In the last 10 years, there have been quite a few versions of Vocabulary Study, most focusing on fancy graphics and experimental (mostly-useless, to be honest) features—CD players, screen savers, and other goodies.

The old versions of Vocabulary Study needlessly required using a special-purpose editor and setting options in a number of esoteric configurations files. To the best of my understanding, old versions would only run at low-resolution in full-screen mode in modern versions of Windows, and they had exceptionally poor performance.

It is time for Vocabulary Study to return to its roots—to the basic features that it needs without all the extras, to do the job it was intended for and nothing else. It is time for Vocabulary Study to leap out of the DOS world and to conquer every operating system in a new, minimalist form. It is time for Vocabulary Study DX6!

Example Use

To start using Vocabulary Study, start the executable; this is done in a system-specific way, but probably typing “./vocab”. Using a tool like LEdit¹ is also helpful, but not required. In this example, we will consider using the “sample.data” file included with the distribution; using other data files is analogous. Once Vocabulary Study has loaded, you should see a welcome message á la:

```
Welcome to \vs
created 2005 Royal Ross
version DX6 0.1
  [1 remaining]

<please enter a command; ".help" to see valid commands>
```

this will be followed by the prompt “#”. Input is line-based in Vocabulary Study—text we enter will be processed only when we type the “return” key; this detail is, therefore, not mentioned again in the tutorial.

We will now load a file; the details of what we’re doing (specifically of “.load”) are explained later. As mentioned earlier, for the purposes of demonstration, we’ll use the “sample.data” file. First, we load the file by typing “.load”:

```
<please enter a command; ".help" to see valid commands>
# .load
```

Vocabulary Study prompts for the file we want to load; we enter “sample.data”:

```
please enter a file to load
# sample.data
```

Vocabulary Study tells us that the file has been loaded:

```
loaded:
  title:Test Data
  author: Royal Ross
  date: Today
  version: 0.1
```

Now, we enter what is known as a “study iteration”—Vocabulary Study will display each question in turn, prompting for the associated answer. Here is the question and the prompt:

```
[2 remaining]

test question 1
#
```

¹Available at: ftp://ftp.inria.fr/INRIA/Projects/cristal/Daniel.de_Rauglaudre/Tools/

Notice that Vocabulary Study displays also how many questions remain. Now let's suppose we type the wrong answer:

```
# foo
wrong!
```

Vocabulary Study displays a message that we have entered the wrong answer and allows us to try again until we enter the correct answer, keeping count (for statistical purposes) of how many times we've entered the wrong answer. Now, let's enter "test answer 1", the right answer:

```
test question 1
# test answer 1
correct!
test hint 1
```

The hint associated with this item is then displayed, "test hind 1" in this case; nothing is displayed if no hint is defined for the current test item. After we enter the correct answer, Vocabulary Study moves on to the next item. After we've finished all the items, we go through those we had wrong. This process continues until no items are answered incorrectly. Vocabulary Study then displays some statistics showing how we've done:

```
Results summary:
Most often wrong:
  1 wrong 1 time
Score: 650
Mark: More knowledge needed.
Advice: Study more.
```

It should be made extremely clear that the "advice" is exactly that—it's simple, mechanical advice based on a few calculations. It may be helpful to see your progress, but use your best judgement regarding whether to study more.

Once the summary has been displayed, we're back where we started!

Commands

Normally, you can enter either the answer to a question or a command; each command consists of '.' followed by a word. ".help", therefore is a command, and entering ".help" will, as stated, list available options:

```

# .help
commands:
.h[elp]      -> display this help
.a[nswer]   -> display answer to current item
.l[oad]     -> load a data file
.s[avestate] -> save the current state to file
.l[oad]s[tate] -> load a state from file
.m[ain]     -> return to the main menu
.q[uit]     -> exit the program
.r[andom]   -> randomly select a subset of items to study

```

Let's explore each command in turn.

.help

We already know what that does! Entering “.h” will invoke the same behaviour.

.answer

If, during a study iteration, you do not know the answer to a particular question, you can use the “.answer” command to display the answer. Doing so, though, counts as a wrong answer. For example, again using the “sample.data” file, if we have the question “test question 2” and would like the answer:

```

[2 remaining]

test question 2
# .answer
the answer is: test answer 2

[2 remaining]

test question 2
#

```

Vocabulary Study re-displays the question and the prompt, waiting for you to type the answer. Entering “.a” will invoke the same behaviour.

.load

Loads a data file from disk. For example, if we want to load the “sample.data” file that is included with the distribution, we type “.load”; Vocabulary Study then prompts for a file name, and we enter “sample.data”:

```
# .load
please enter a file to load
# sample.data
loaded:
  title:Test Data
  author: Royal Ross
  date: Today
  version: 0.1
```

Vocabulary Study loads the file and displays its header before beginning a fresh study iteration. Entering “.l” will invoke the same behaviour.

.savestate

Saves the current session to file; it can be re-loaded by “.loadstate”. For example:

```
[2 remaining]

test question 2
# .savestate
please enter a file to save to
# test.save
state saved
```

It is now safe to exit since the file can be reloaded later. Entering “.s” will invoke the same behaviour.

.loadstate

Loads a session saved in a file; presumably this session was saved by “.savestate”. For example:

```
# .loadstate
please enter a file to load from
# test.save
state loaded

[2 remaining]

test question 2
#
```

The session is precisely as it was when saved; Vocabulary Study re-displays the question and the prompt, waiting for you to type the answer. Entering “.ls” will invoke the same behaviour.

.main

Aborts the study iteration, returning to the main menu. This throws away the current state information, so the iteration cannot be resumed (unless it has been saved via “.savestate”). Entering “.m” will invoke the same behaviour.

.quit / .exit

Exits Vocabulary Study. Entering “.q” will invoke the same behaviour.

.random

Selects a specified-size, randomly-selected subset of the entries to study; this is useful in case, e.g., of large, cumulative files. It will prompt for how many items to select. For example:

```
loaded:
title: French Cumulative Review
author: K.D.P.Ross
date: 20050303

version: 0.3

middle school

[629 remaining]

# .random
please enter the number of items to select

# 100
april

[100 remaining]

#
```

Entering “.r” will invoke the same behaviour.

Data Files

In order to use Vocabulary Study, you will probably want to enter your own data. This is easy, though, because data files are now in a simple, plain-text format with a 4-line header and 3-line-per-item (question-answer-hint) data format. Specifically, they are in the following format (this just defines the precise format, you can skip this if you like and look at the example file, below):

```

datafile  d ::= h b
header    h ::= t ↵ a ↵ d ↵ v ↵
  title   t ::= string
  author  a ::= string
  date    d ::= string
  version v ::= string
  body    b ::= it+
  item    it ::= q ↵ ans ↵ hnt ↵ — imp
question  q ::= string
answer    ans ::= string
  hint    hnt ::= string
import    imp ::= {{import string}}

```

where ↵ is a line break and *string* is an arbitrary sequence of characters that does not include a line break. There is an additional caveat that program behaviour is undefined if the first character of answer is ‘.’.

As an example consider the following valid data file:

```

Test Data↵
Kyle Ross↵
Today↵
0.1↵
test question 1↵
test answer 1↵
test hint 1↵
test question 2↵
test answer 2↵
test hint 2↵

```

where, again, ↵ is the normally-invisible line-break character.

Import

A single-line import declaration in a data file causes the loader to invoke itself on the file referenced after the “import” keyword. This allows creation of modular data files that can be loaded in arbitrary combinations. More specifically, the following data which, say, resides in “test.data”:

```
Test Data↔
Kyle Ross↔
Today↔
0.1↔
test question 1↔
test answer 1↔
test hint 1↔
{{import foo}}↔
test question 2↔
test answer 2↔
test hint 2↔
```

will be loaded as follows: The lines from the above file will be read from “test.data”. The header will be processed. The first data item will be processed. The import declaration will be read, and processing of “test.data” will be suspended whilst the loader fully loads “foo.data” (and any files imported by “foo.data”). The loader will then resume processing “test.data”, starting with the second data item. The end of file will be reached, resulting in a successful load (assuming that loading “foo.data” terminated successfully). Behaviour is undefined if any data file involved an a “load set”—the set of files containing the directly-loaded file and those loaded because of the existence of some “import chain” from the directly-loaded file—is deleted before completion of the entire load; behaviour is also undefined if there is an “import cycle” in the import chain—if there exists some subset of the files $\{f_1..f_n\}$ in the load set such that f_1 imports f_2 , f_2 imports f_3 , \dots , f_{n-1} imports f_n , and f_n imports f_1 ; for obvious reasons (i.e., that this is unchecked), this will probably cause Vocabulary Study to “freeze”² or display the error “failed to load file”, depending on the system and particular data files; this behaviour is by design (although it may be specified in future versions of Vocabulary Study). The lesson is: Do not have cycles in your data files!

As a final note, the import name is processed relative to the current working directory. The easiest way to ensure that your files load properly is to create symbolic links (i.e., “shortcuts”) in the Vocabulary Study directory to your data files and then refer to them by unqualified names. Absolute paths will also work, but this will, of course, cause problems if the files are moved and is, therefore, discouraged.

About the Implementation

For those who are programmers, a few words about the implementation of Vocabulary Study DX6: The implementation was a bit of an experiment in several

²In practise, this is very unlikely: Even if no data items are ever loaded (because the import-cycle import declarations precede the data items in their respective files), there will most likely be a stack-overflow because the recursive calls are non-tail calls. It may, however, take quite some time before an error is displayed.

senses. These are briefly discussed in this section.

SCurses

SCurses is a very simple library that I wrote to interface O’Caml with NCurses, a widely-used text-based user interface library. This is the first application in which I put SCurses to use. My main interest in doing so was to determine which abstractions of the low-level SCurses functionality would be useful in building an application using the library. The library Curses.util has these abstractions, most of which should be highly reusable: Centring, line-wrapping, single-dimension cursor movements and position queries, relative cursor movements, drawing a frame around the terminal.

Interface Design

I have taught a course several times in which students implemented a project (in Java, but that is hardly a relevant choice here) with a user interface (UI). I always made the argument that it *should* be possible to implement the core functionality and logic of a project *parametrised over a UI*. In “Caml speak” this means that the core is implemented as a functor `ProgramLogic` taking an argument satisfying the UI signature. This is precisely how Vocabulary Study is implemented. As a proof-of-concept, both a Curses interface (using the library discussed supra) and a line-based command-line interface. My ineptitude programming GUI’s is the only reason that there is no graphical interface; theoretically, it should be easy to add one.

I view the experiment as successful: The implementation works well, and it is a simple matter of switching the instantiation of `ProgramLogic` to change interfaces. However, I cannot draw any general conclusions regarding how practical this parametrised-interface paradigm is: I have not implemented it with a GUI and the present application is very small—both in terms of code-base size and in terms of the complexity of the interface requirements. I do not, though, see any reasons that a commercial-scale application could not be implemented in similar fashion. My hypothesis is that this method could be used to implement a platform- and interface-independent “kernel” that could then be instantiated with environment-specific components, resulting in clearer separation of concerns, modularity, and all the other positive software-engineering adjectives one might care to insert here. It also occurs to me that this is probably something that large-scale projects have been using for years; my only claim of novelty is in formalising the approach by using the language-supported functor (or type-class in Haskell) abstraction.