Teaching the Ancients to Type: Better Unicode Text Entry for Ancient Greek and Hebrew

Steven Tammen

June 24, 2018

Contents

1 Section 1: What is this project? Why this project? Why this paper?

1.1 What is this project?

While this project has many different goals and subgoals (and continues to add more as additional matters of convenience and usability come up), the essential aim is to create easy-to-use keyboard layouts for *non-native* languages. What exactly does this mean?

Typists for a particular language can usually be classified rather easily into native speakers and non-native speakers. Native speakers type their language "a lot" – with respect to both frequency and quantity – while non-native speakers do not. For example, someone who is bilingual in English and Spanish might type approximately 50% of their text in either language; they have two native languages. However, someone who types 90% of their text in English and 10% in German (perhaps to communicate with a colleague or business associate) has only one native language – English. The lines can get blurry, of course, but the general idea is that one can usually cleanly categorize languages based on how often and how long they are typed: some are typed often and for a long time (native languages), and others are not (non-native languages).[1]

In theory, keyboard layouts for native languages should be designed according to certain keyboard design metrics that make typing more efficient. Nowadays, optimization is accomplished through computer programs that change letters in a configuration until additional changes do not improve the layout any more. Such an approach is known as a *genetic algorithm*. Examples of this approach may be seen for Chinese in Liao and Choe (2013) and for Arabic in Malas, Taifour, and Abandah (2008).[2]

Layouts designed in this manner perform better with respect to typing metrics such as low finger travel distance (which helps reduce unnecessary movement away from the home row) and high hand alternation (which helps prevent many characters from getting typed contiguously by one hand while the other sits idle). However, since different languages have different phonetic patterns and orthographies, so-called "optimal" layouts for different native languages will place even phonetically and orthographically identical letters in very different places.

The question of whether or not it is better for bilingual and trilingual individuals to try and learn one keyboard layout that is a compromise between their multiple languages or separate keyboard layouts for each language is a fascinating one, but it is ultimately separate from the matters this project concerns itself with. This project is instead focused on situations of language imbalance – given that there is a dominant keyboard layout (presumably for one's native language(s)), what is the best way to type non-native languages?

[1]This is admittedly not exactly how native and non-native languages are typically defined, but hopefully it is a forgivable simplification. People who type a language they did not grow up speaking as a significant percentage of their total volume may not be "native speakers" by some people's definitions, but the terminology is employed here for the purpose of avoiding such verbose titles as "effectively native languages" and "non-effectively native languages."

[2]People interested in this process are encouraged to visit `http://www.adnw.de`. This site contains much background on the history of keyboard layout optimization, and a well-documented C++ optimizer program. The main focus of this site is German layouts, but there is a fair bit of discussion for English layouts as well.

1.2 Why this project?

Multilingual text input for non-native languages is a solved problem. By this I mean that at the time of writing, it is possible with various existing software options to enter text both in a primary language and in multiple alternate scripts (e.g., Greek, Hebrew, Cyrillic, Arabic, Devanagari) with relative ease.

So why bother working on another project addressing these things? It's a fair question.

1.2.1 To combat the lack of open source, *customizable* software

This section will use the present state of Greek text input as an example to illustrate how customizable software is currently lacking. Similar situations and observations hold for other languages that I have knowledge of, but will not be discussed for brevity's sake.[3]

Current options for Greek text input

{Todo: discussion of various already existing software options. List from Bibliography, explicit coverage of system options.}

Positive characteristics

Among different options one can observe several important design characteristics. Some (which? be specific) solutions are *homophonic*, meaning that alpha is put on the A key, beta on the B key, and so forth. Some (which? be specific) attempt to avoid complex chord sequences when entering diacritical marks by using punctuation-based mnemonics. Some (which? be specific) allow flexible entry order, meaning that breathing-then-accent or accent-then-breathing (for example) both correctly display. Many (which? be specific) allow for text entry across applications rather than having to copy and paste out of some "special" window. All of these things are definite positives for a typist, especially one switching into Greek text entry from time to time while primarily typing in his/her native language(s).

Customization and open source

However, users who wish to customize things are out of luck with present options. Some people may wish to change how diacritics are handled, for example,

[3]Discussion of options from research in Hebrew. Maybe put in Appendix somewhere?

or to change which Latin-script letter chi goes on to conform to their preferences (both C and X are popular, but it is irritating to have to deal with both mappings). Because current options are closed source without significant customization interfaces, this is simply not possible.

Customization and open source software go hand-in-glove, and especially for a program such as this – which is dealing with a very domain-specific problem that contains much that is subjective and/or related to user preference – there is significant benefit to making software community-driven.

Of course, open source software has some other benefits as well. Open source software is free and relatively more stable than close sourced software. There is never a guarantee of long-term stability with programs that do not publish their source code, since if the projects stop getting maintained (a company goes out of business, e.g., or the primary developer dies suddenly), nobody else can pick them up and keep the code running smoothly on new hardware and/or operating system environments. This is actually a somewhat greater concern for projects of this sort: since programs dealing with keyboard layouts must depend on system calls to interface with keyboards, they are necessarily less insulated from the operating system environment than many other kinds of programs. In other words, if an operating system changes one of its low-level libraries for handling streams of keys, it will likely break a program dealing with keyboard layouts, while a browser or music player might still work just fine.

If I were forced to pick "just one" reason why this project existed, it would be this: to create a customizable and open source framework for text entry of non-native languages.

1.2.2 To combat the lack of software that bundles multiple language layouts together

This software is being developed in close association with Classicists, and the initial project scope is, in many ways, targeted at solving the problems of Greek scholars in this field. However, I am trying to create a framework that may be comfortably extended to other languages and alphabets as needed.

Some academic fields (e.g., Historical Linguistics, Classics, Ancient Near East, and Ancient History), have significant language demands. It is not uncommon for people studying in these fields to pick up multiple ancient languages (including, but certainly not limited to, Latin, Greek, Hebrew, Arabic, Syriac, and Sanskrit), with many of these having complex alphabets. A lack of consistency in approaches can be frustrating, particularly if one has to go through the bother of installing and updating text entry solutions for all these languages on all the computers used for writing.

Additionally, much secondary scholarship in these fields is in German, French, and Italian, all of which share the basic English character set, but demand a few special characters and/or accents. It is conceivable for a scholar working on research about Mediterranean trade in Late Antiquity, for example, to need to type in English for their core analysis, Latin, Greek, and Syriac for primary sources, and German, French, and Italian for secondary sources. Assuming Latin is typed without macrons and accents, that leaves 5 additional languages on top of English that must be dealt with.

While it is more a future goal than a priority of ”round one” of this project, bringing multiple language layouts together in the same program is one of the central motivations behind creating another project dealing with these things. Starting from scratch rather than adding on to an existing program ensures that there will be seamless interoperability in the future, and that standards and design guidelines may be established.

1.2.3 To combat the lack of software that adds functionality without removing any

Using keyboard shortcuts can be a frustrating experience when you have to type in another language. If there is no intelligent handling of modifier keys, people typing in a non-native language might miss such shortcuts as Ctrl-C (copy), Ctrl-X (cut), Ctrl-V (paste), Ctrl-Z (undo), and Ctrl-S (save). The situation is especially bad for those who use Vim, Emacs, or other text editors that make use of the keyboard (rather than a GUI) for functionality, and for people who use keyboard-driven window managers, browsers, application launchers, window switchers, and so on.

It can also be frustrating to ”lose access” to some English keys (typically punctuation such as brackets) when typing in another language. If a language layer ”steals” English punctuation keys thinking that they will never be needed when typing that language, but does not provide any way to access said keys short of disabling the software temporarily, it can create an unpleasant user experience.

Things like these are not the most obvious design factors when one thinks of typing in non-native languages, but it has been my experience that these are actually almost as important as the layout design itself. The devil truly is in the details.

1.2.4 To combat the lack of software that works for nonstandard keyboard layouts

Another reason for the creation of this project in particular is the fact that currently available homophonic layouts (at least those that function at the system level) do not work for ”nonstandard” keyboard layouts – they all assume a QWERTY base mapping.

People typing on Dvorak, Colemak, QWERTZ, BÉPO, and so forth may wish to have the benefits of homophonic letter layouts in their non-native languages while retaining their native base mapping. Portability is a high priority of this project, and all of the functionality in any language can be implemented on whatever base layout is desired, with full customization as an option.

1.3 Why this paper?

1.3.1 Justifying design choices

This paper is intended to fill the void between low level implementation details (Should arrays or strings be used to send keys? Global variables or classes?) and the end result of fully functioning keyboard layouts.

I personally find it extremely frustrating when design decisions have no specific thought process behind them. For this reason I am attempting to document things in such a way that I would be satisfied as a user of this software, if I were not the one designing it in the first place. The placement of letter keys, the choice of particular punctuation keys for diacritics, the mechanism for switching languages, the process of entering ”normal” punctuation when on a non-native layer; these are the sorts of design decisions that this paper sets out to explain.

The idea is to have something to point to when someone asks, ”but why?” Rather than saying ”just because” or trying to come up with rationalizations *ex post facto*, attempting to rigorously justify everything from the get-go should lead to a project wherein there are not an abundance of arbitrary program characteristics. At least in theory.

1.3.2 Creating a starting point for people that may have different opinions than myself

With all this being said, this paper is certainly not attempting to close discussion on these topics or be the last word on design factors. At the time of writing, I have worked with Greek for approximately two years, and any sort of serious coding for about as long. I am sure one could easily find people more qualified than myself for virtually any aspect of this project, and also for all of them put together.

Instead, the idea is start a conversation about these things in a more formal manner. I am certain that Classicists, for example, are opinionated about how they wish to type Greek, and things that drive them crazy about current options that let them type Greek. If this paper can present one rationale that can be critiqued and examined, and the code behind this project is designed in such a way that it is sufficiently flexible, it should be possible in the future for this project to come to encompass multiple points of view, and circle in on an increasingly sophisticated understanding of the design variables in play.

{Todo: maybe mention survey and results here?}

2 Section 2: Nuts and bolts

Before getting into this project in particular, it is proper to briefly examine the nuts and bolts that make multilingual text input a possibility on modern operating systems. Much more could be written about any of the things here, but the present section will seek only to provide a sufficient amount of background to give readers an appreciation for the complexity at play behind the scenes.

2.1 Keyboard layouts

To be able to type in a language that is not the default for your physical keyboard and system layout (e.g., a QWERTY ANSI keyboard used for American English), a different keyboard layout is necessary. In essence, a keyboard layout translates presses of physical keys into characters or key events (like Enter or Tab).[4] I find it helpful to split up keyboard layouts for languages into smaller semantic groupings to make them easier to think about, especially for people that must implement them in software.

2.1.1 Letters

For languages with alphabets ({Todo: footnote: as opposed to syllabaries or Abjads}), keyboard layouts must provide a means for typing all of the letters. English has 26 letters, but other languages often have more or less.

[4] To be more precise, keyboards send hexadecimal scancodes that are interpreted by the operating system kernel. Depending on permissions, different programs can inject themselves into the input system, and intercept keypresses before they get sent to other programs. This is what allows a remapping program to change the behavior of sent keys: the scancodes sent by the physical keyboard are the same, but they are intercepted and replaced with virtual key codes that encode different behavior.

Letters may be further subdivided into vowels and consonants. Vowels are typically the more interesting variety inasmuch as most markup (such as accents) revolves around vowels, and therefore they typically require more work to integrate into the layout. For example, Greek vowels may take accents, breathings, iota subscripts, and so forth, while Greek consonants (with the exception of rho) take none of these things. This means that designers do not need to keep track of consonants as closely as vowels, generally speaking.

Many languages have uppercase and lowercase letterforms, but not all languages do. Hebrew, for example, does not have any casing distinctions. In general, implementing uppercase forms involves keeping track of shift state, but not too much extra work other than that.

2.1.2 Context-specific/alternate letter forms

Some languages have letters that change their form based upon their position in words. For example, word-final sigma in Greek changes forms, and many letters in Hebrew and Arabic also exhibit this behavior.

Semantically, the letter is still the same, and should not therefore be thought of as a new or different entity. However, implementing positional letterforms does require some extra work, particularly in terms of identifying word boundaries. One approach to handling final forms is replacing the base form with the final form when and only when a key signifying a word boundary (such as Space or .,?!) is pressed immediately following a letter with final form behavior.

In addition to final forms, some languages have alternate forms of letters. In Hebrew, for example, some of the so-called Begadkephat letters (tav, dalet, gimel) have alternate forms for when they are aspirated, while others (bet, khaf) fully change their phonetic value through an alternate form. The line here can be a bit blurred between these alternate forms (which use a mark called a *dagesh*) and letters with diacritics. The dagesh can be used with other Hebrew consonants to double phonetic value, for example, which could be considered a separate use. But the same mark is used.

For simplicity in programming, I recommend structuring development around *program features* (for example, the ability add a dagesh to things... alternate form or no) rather than *language features* (for example, working on developing the capacity to support all possible sounds in a language, including aspirated forms and those that optionally change their phonetic value). This allows the designer of a keyboard layout to focus on one thing at a time, rather than trying to organize development around language features that may not cleanly map onto structured commits. As long as pains are taken not to forget any essential language features, this approach is easier on the programmers while accomplishing the same goals.

2.1.3 Mandatory markup: accents, vowel points, etc.

Most languages have some system of diacritical marks that are considered mandatory, diacritical marks that are essentially ”part of the language.” For example, Spanish and Italian have accents, Hebrew has vowel points, and Greek has accents, breathing marks, and the iota subscript.

These mandatory diacritical marks must be present for language text to be considered correct, and are typically fairly common. For this reason, they require more thought in placement, since an inconvenient location or entry method can render text entry for the entire language unpleasant.

2.1.4 Additional markup: vowel quantity, cantillation marks, etc.

Some languages have another set of markup symbols used in specific circumstances or by specific groups of people. Good examples of symbols in this category are diacritics that indicate vowel quantity: the macron and breve are not ”required” in Latin-script languages, but commonly show up in dictionaries and grammar books to help with pronunciation.

There are also other domain-specific symbols, depending on the language. Hebrew scholars working with the Masoretic text in any capacity will inevitably have to deal with the cantillation marks (the טעמי המקרא, *ta'amei ha-mikra*), used in ritual chanting of the *Tanakh*. Greek and Latin scholars may wish to use metrical symbols to mark dactyls, spondees, and caesurae when scanning ancient epics in dactylic hexameter. Etc.

Implementation of these additional markup symbols is in some sense optional, inasmuch as they are used only by certain groups of people. However, it is best to think of them as features that should be included eventually for robustness, even if they do not make it into the first implementation.

2.1.5 Punctuation; language-specific symbols

While the dominance of English as a computer language has served to standardize international punctuation to a certain extent, some languages still have specific punctuation that is used in lieu of, say, the question mark. Greek, for example, uses a semicolon to indicate questions, and a dot in the middle of the line to indicate a break in thought (i.e., to indicate a semicolon).

The situation is somewhat complex in that ”casual typing” of many languages has led to a situation in which punctuation systems are mixed. It is not uncommon to see Greek imperatives followed by exclamation points in introductory texts, for example, even though this has no precedent in ancient sources.

Numerals are another interesting case. Arabic numerals (0-9) are very much the international standard nowadays, but many languages used to use different numerical systems with different character sets (sometimes some subset of the alphabet, as with Hebrew), which may have special numerical symbols.

Finally, in modern contexts, most foreign currencies have special symbols. It is convenient to be able to access these without complicated and abstruse key sequences.

2.2 Unicode

2.2.1 History

Handling languages with non-Latin alphabets has long been a topic of conversation among people working with computer input systems. Due to historical reasons, computers have developed very much around English and the ASCII character set, with other alphabets being second class citizens.

As computers developed and people moved away from typewriters (which had significant physical limitations that made representing many complex scripts difficult), efforts were undertaken to standardize language input and robustly handle foreign alphabets, even their mixing with English. For example, Knisbacher et al. (1989) discuss Hebrew input on early PCs, Selden (1981) summarizes a early effort to standardize how Arabic was handled on computers, and Mastronarde (2008) summarizes historical Greek options in the first five pages of his excellent presentation on Greek and Unicode.

As memory and storage sizes have increased, it has become acceptable to use multiple bytes for the storage of text characters, and thus much easier to handle all of the characters necessary for multiple complex alphabets. Unicode attempts to solve the challenges of dealing with multiple languages by defining values that map to characters across different numeric ranges. In this way, Unicode allows for multiple languages to be typed without conflict, since the characters are all being represented by different numbers in memory.

2.2.2 Scope and purpose; peculiarities

Unicode is theoretically laid out in terms of ”blocks” for different language sections. Unfortunately, due to various considerations (politics, lack of foresight, an initial project scope that did not encompass historical/uncommon characters), it is not uncommon for characters of the same language to be spread out across several numerical ranges. The initial Greek block, for example is sufficient for monotonic Greek accentuation, but leaves a lot to be desired in terms of polytonic Greek. The Greek extended block helps in the area of polytonic Greek, but still leaves many uncommon or regional characters without official support.

Unicode seeks, in some sense, to be the ”kitchen-sink” solution. When you type Unicode text in a document with encoding such as UTF-8, you have the capability of using all of the 1-million-plus characters together (a decidedly good thing). However, the nature of its all-encompassing haphazard growth has made it somewhat more difficult to understand from a language-centric perspective (as in you are using two of the hundreds of possible languages, and have no need for the rest), and has caused the full encoding to include some puzzling, kludgy behavior.

A good resource discussing such Unicode peculiarities from the Greek side of things is Nick Nicholas’ page on Greek and Unicode: `http://www.opoudjis.net/unicode/unicode.html`. Many Unicode choices that seem strange at first glance may still seem strange at second glance too, but typically there are reasons for why things are the way they are (even if they are unsatisfying and historical).

2.2.3 Precomposed and decomposed Unicode

As time has passed, the Unicode consortium has gotten more and more reserved about adding additional precomposed characters. After all, so the reasoning goes, combining diacritical marks are already supported in the Unicode specification. Why should Unicode have to support ”redundant” precomposed characters if you can just enter the same character as a sequence with combining character(s)?

The logic is fine so far as it goes, but the problem is that the Unicode text encoding is only half of the picture: without fonts that properly support decomposed sequences, decomposed Unicode is not really an option. There have historically been many problems with fonts improperly displaying combining characters. For example:

- The combining characters might be horizontally off center compared to the letter
- The vertical spacing between the letter and the diacritic might be too little or too much
- Multiple combining characters might overlap with each other, or not stack properly
- Etc.

Because different base characters have different physical characteristics (some are taller, or wider, or have ascenders and descenders to deal with, e.g.) there is no cookie-cutter solution for physically placing combining characters. Rather, it must be done for each letter individually.

As will be discussed below, there are actually modern fonts that handle decomposed Unicode well. However, there are still plenty of fonts that do not, especially when you start combining multiple diacritics, or using any uncommon diacritics.

2.2.4 Combining multiple diacritics

An additional wrinkle in decomposed Unicode with multiple combining characters is the entry sequence. What happens if you type all the permutations of three different diacritics – do they all display the same?

The answer will typically be no. In the second chapter of the Unicode 11 manual (`http://www.unicode.org/versions/Unicode11.0.0/ch02.pdf`), section 11 deals with combining characters, and discusses the default combining behavior for multiple combining characters:

> By default, the diacritics or other combining characters are positioned from the base character's glyph outward. Combining characters placed above a base character will be stacked vertically, starting with the first encountered in the logical store and continuing for as many marks above as are required by the character codes following the base character. For combining characters placed below a base character, the situation is reversed, with the combining characters starting from the base character and stacking downward.

2.3 Fonts

2.3.1 Supporting decomposed Unicode

2.3.2 Private Use Areas

- New Athena Unicode

2.3.3 Using the same font for native languages and non-native languages

3 Section 3: The Unicode Language Layers project

3.1 Sane defaults combined with ease of use

- Letters, diacritics, etc. At least have "some reason" for placements of everything

- Defaults should match up to the ”normal user” and what they would find best

3.2 Customizability as a first order priority

- Thorough API
- In-line comments
- Examples in the form of Greek and Hebrew layers

3.3 Minimal interference with normal computer use

- Quick and easy on and off
- Consistent keyboard shortcuts (languages do not interfere with normal shortcuts)
- Leader-prefixed punctuation for normal behavior (for when punctuation gets hijacked by a layer for diacritics and so forth)

3.4 Consistency across multiple languages

3.4.1 For end users

- Base markup for Latin, German, French, Italian, Spanish. Leader-prefixed diacritics.
- Switching between different alphabets; using different alphabets

3.4.2 For designers

- Consistent handling of precomposed and decomposed Unicode
- Abstracted, language-blind functions to extend to new languages with minimal effort
- If you understand how to code a layer for one language, you should be able to code layers for other different languages.

4 Section 4: Greek as an example

4.1 Letters

4.1.1 The relationship between memorability and speed

Touch typing is a skill acquired over time through practice. Given that most individuals typing ancient languages in scholarly pursuits (e.g., Classicists, Ancient Near East scholars) will not need to enter large amounts of text in ancient languages, and will not need to do it with great frequency, it is worth considering the time-cost associated with learning keyboard layouts for ancient languages.

Keyboard layout design is a complicated process with many optimization variables. Today, layouts may be judged using algorithms like {Todo}, which track many metrics that are likely associated with performance. I say likely, because there has not been formal scholarship on the subject done in such a way that we may be sure about such things. Part of the problem involves the difficulty in doing research: you cannot blind research about keyboard layouts (people must know the layout they are typing on), you cannot have a realistic control group (everyone who has used computers already has varying levels of experience typing on keyboards – even people who hunt and peck have cognitive maps of their layout), and many things that one might want to measure – most notably comfort and repetitive stress – are difficult to get good, objective measurements for.

With all this said, there are some things that are not controversial. Having more commonly typed characters on the home row leads to less hand movement and theoretically faster speeds. Avoiding having the same finger type multiple keys in a row (cf. QWERTY's ”minimum”) enables the typist to ”line up” fingers when typing, so that multiple keys may be in the process of being pressed at once.[5] Having work split between the hands is more balanced than having it all concentrated on one hand (cf. QWERTY's ”stewardesses”).

As a general rule of thumb, so-called ”fully optimized” layouts will have relatively poor memorability. If you let a genetic algorithm design an optimized layout for you, it will not keep all the letters in a block or numbers in a row, but mix everything together according to frequency considerations. We humans are very pattern-oriented creatures, and having no apparent structure to characters will make a keyboard layout more difficult to remember, to some degree. Furthermore, it is obvious that keyboards that are easier to remember will be easier to get up to speed with.

[5]While I don't know of a formal source for numbers, many expensive keyboards market themselves as being better for fast typists due to allowing for so-called ”n-key rollover” (NKRO), which lets many keys be pressed simultaneously, as opposed to the 6-key rollover of most USB keyboards.

The issue in all this is that due to a lack of research, I cannot say definitely how much easier semantically-grouped keyboard layouts are to learn, or how much faster people may train them to, say, 35 WPM. The data for this simply does not exist. However, this paper is operating on the safe assumption that these considerations are non-negligible for most people in most circumstances. The hypothesis coming from this is this: since people typing ancient languages will not be typing them with great magnitude and frequency, it is more rational to focus on memorability over raw optimization considerations, since layouts that are easier to remember will be faster to learn, and the benefits of ”brute forcing” an optimized layout (as one might do for one’s native language) will never be realized in typical use cases.

4.1.2 Native-language layouts in muscle memory

The above discussion focused on the interplay of memorability, layout optimality (as measured by finger travel distance, same finger, etc.), and ease of acquisition in the abstract. However, assuming users of this project can already type on a keyboard layout in their own language (in whatever regard: touch typing, hunting and pecking, etc.), we do not need to start from ground-zero.

The general idea is that for the circumstances under which most scholars type ancient languages it is *always* better to associate a keyboard layout for an ancient language with a keyboard layout for a native language already in muscle memory. Associating a new layout with the old layout lets typists reuse neural pathways that are already in place rather than forming new ones from scratch.

What do I mean by this? Let’s take the Greek letter alpha. Most people, Classicists or no, know that alpha corresponds in phonetic value to the English letter A. Alpha also happens to look like the letter A in both its lowercase and uppercase forms. So, rather than putting alpha on some random key, why not simply place it on the same key as the letter A in English?

4.1.3 Issues in constructing associations

If we accept the premise that it is best to form correspondences between ancient languages and keyboard layouts already in use (for English or otherwise), then it follows that we need some formalized system for doing so.

Layouts derived from phonetic matching are typically called ”homophonic layouts.” While homophonic layouts are excellent when correspondences exist, there are some letters in languages that have no clear English equivalent. Theta in Greek, for example, corresponds to the phoneme in English that is represented by the digraph ”th.” These must be dealt with separately.

There are also some cases when a language has two letters for the same phoneme. In Hebrew, for example, the consonant Vet (Bet without a dagesh) is equivalent to the consonant Vav – they both make ”the V sound.” So which one should occupy the V key?

The associations (henceforth keymaps, short for ”key mappings”) below attempt to solve such issues in a systematic way. Following the hypothesis presented above (namely, that memorability is a more important concern in these circumstances than raw optimality), priority is given to phonetic correspondences, then visual correspondences, then transcription correspondences, then, finally, to raw optimality. {Todo: why?}

4.1.4 A Greek-English keymap

Foreword {Todo: footnote this/put in appendix}

I have attempted to make the above discussion general enough that people with native languages significantly different than English (Russian, say) may easily transfer these ideas into layouts that fit their languages. However, from this point forward, discussion will center around English and languages that have a close association with it (the same general alphabet and phonology).

Phonetic correspondences

I have opted to supply the fricative versions of Theta and Phi, according to later developments in the language. People interested in classical 5th century Attic pronunciations can substitute the aspirated plosives if they wish. (I have made this substitution because I have observed that most people learning ancient Greek have a much easier time distinguishing the phonemes this way, and thus avoid mixing up Theta/Tau and Phi/Pi in their writing). {Todo: don’t be arbitrary. Explain, don’t assume}

If a letter has any English equivalent (even if it has additional sounds in some contexts not found in English), I have opted to match them. I have also opted to match ”near misses” – sounds that aren’t quite identical, but are close enough that they are obviously connected (such as the Greek Rho and English R, and many of the vowels). {Todo: handle cases of similar sounds like o/w e/h, etc. Also weighting phonetic correspondence vs. frequency/visual correspondence as with digamma and omega}

Greek letter	IPA	English match
Α α	[a], [aː]	A
Β β	[b]	B
Γ γ	[g], [ŋ] (before velars)	G
Δ δ	[d]	D
Ε ε	[e]	E
Ζ ζ	[zd]	Z
Η η	[ɛː]	
Θ θ	[θ]	
Ι ι	[i], [iː]	I
Κ κ	[k]	K
Λ λ	[l]	L
Μ μ	[m]	M
Ν ν	[n]	N
Ξ ξ	[ks]	X
Ο ο	[o]	O
Π π	[p]	P
Ρ ρ	[r]	R
Σ σ	[s]	S
Τ τ	[t]	T
Υ υ	[y], [yː]	U
Φ φ	[f]	F
Χ χ	[kʰ]	
Ψ ψ	[ps]	
Ω ω	[ɔː]	

This ”first pass” at matching gets us pretty far - only 5 letters remain unmatched.

Visual correspondences

Look-alike letters, even if they have no phonetic correspondence, can be an easy way to remember letters. Anything that helps create mental associations can help speed up the learning process. Both uppercase and lowercase forms are considered.

Greek letter	English match
Η η	H
Θ θ	
Χ χ	
Ψ ψ	Y
Ω ω	w

Uppercase Eta looks identical to the uppercase form of the English letter H, and lowercase Omega looks very similar to the lowercase form of the English letter W. Uppercase Psi looks similar enough to the uppercase form of the English letter Y that it is worth using as a mnemonic, in my opinion.

Note that while Chi looks very similar to the English letter X, we are already using X to represent Xi.

Transcription correspondences

One of the problems with transcription is that it is not terribly standardized. For example, scholars preferring a transcription scheme closer to Greek will typically transliterate Kappa as ”k” and chi as ”kh” as opposed to the more Romanized ”c” and ”ch.” However, ”typical” transcriptions may provide some help in providing mnemonics for our remaining letters.

I have opted to only look at strictly alphabetical transcriptions, rather than any that use diacritics. {Todo: why?}

Greek letter	”Typical” transcription	English match
Θ θ	th	
Χ χ	ch	C

Chi is transliterated as ”ch” in most transcription schemes, even if Kappa is transliterated as ”k.” So it seems logical to use the letter C to represent chi.

Leftovers

Theta is a tricky letter to place, since none of our correspondence efforts appear to help with it. English letters that are left include Q, V, and J.

None of these letters is particularly satisfying as a choice, but J is probably the best for people that type on QWERTY or its variants (like AZERTY, e.g.), since it is on the home row and does not have any same finger with vowels. For this reason, I have made it the default mapping for theta. People that do not type on QWERTY (Dvorak, Colemak, Workman, etc.) may want to alter this location, depending. I type on a custom layout and kept it on J because it was still the best location.

As to Q and V, I have these default to Koppa and Digamma, respectively. Both of these come from earlier forms of Greek that are closer to the Phoenician, but may be useful to type on occasion. For people that read on for the Hebrew keymap, Koppa~Quf and Digamma~Vav, so Q and V are actually logical choices given the Semitic consonants underlying these letters.

Digamma dropping explains the -ευς declension and the development of certain stems and words. For example, βασιληϝ- to Βασιλεύς, νηϝ- to ναῦς, βοϝ- to βοῦς, and so on.

Koppa can be also be useful in explaining language development, as can the third and last early Greek letter: San (allophonic with Sigma). {Todo: explain how to generate San}

4.2 Context-specific/alternate letter forms

4.2.1 Final sigma

4.2.2 Lunate sigma

4.3 Mandatory markup

4.3.1 Breathings

‘

- smooth, rough
- vowels and rho

4.3.2 Accents

- acute, grave, circumflex

4.3.3 Iota subscripts

4.3.4 Diaeresis

4.3.5 The koronis

4.4 Additional markup

4.4.1 Vowel quantity: macrons and breves

4.4.2 The underdot

4.5 Punctuation; language-specific symbols

4.5.1 Question marks and semicolons

4.5.2 A discussion of ”hybrid” punctuation, and accessing normal punctuation when desired

{Todo: [6]}

5 Section 5: Hebrew as an example

5.1 Letters

5.1.1 Handling cases of identical letter sounds

5.1.2 A Hebrew-English keymap

5.2 Context-specific/alternate letter forms

5.2.1 Word final letters: the sofit forms

5.2.2 The Begadkephat letters

5.2.3 Shin and Sin

5.3 Mandatory markup

5.3.1 A note about opinionated design decisions

- ”Case study” – the *matres lectionis* letters. Automatically including vav and yod when they are vowel indicators.

[6] Metrical marks, special numerals, drachma symbol

5.3.2 Basic vowel points

5.3.3 Shva and reduced vowels

5.3.4 The dagesh

5.4 Additional markup

5.4.1 The meteg

5.4.2 Cantillation marks

5.5 Punctuation; language-specific symbols

5.5.1 A discussion of languages that use ”mostly normal” punctuation (from the English point of view)

5.5.2 The geresh

5.5.3 The gershayim (lit. ”double geresh” – this word is plural)

5.5.4 Colon and *sof pasuq*

5.5.5 Vertical bar and *paseq*

5.5.6 Hyphen and *maqaf*

5.5.7 Shekel symbol

6 Section 6: Efficient typing practice for non-native languages

6.1 Introduction to efficient typing

6.1.1 Practicing based on word frequency

6.1.2 Practicing based on N-gram frequency; affixes

- (Derivational) Morphemes rather than words as a training focus

6.1.3 Abbreviating very frequent words and phrases

6.1.4 Practicing the sorts of texts you are going to type

6.2 Creating necessary resources

6.2.1 Word frequency tables

- Perseus, TLG, handling overlapping forms

6.2.2 N-gram frequency tables

- Similar process. Handling semantic boundaries in regexes? How to automate morphological analysis without obvious delimiters like spaces for words?

6.2.3 Area-specific practice texts

- Downloading from free/uncopyrighted sources. Perseus, Project Gutenberg.[7]

6.3 Typing practice

6.3.1 Amphetype

6.3.2 Lesson generation from frequency tables and practice texts

6.4 Crossover benefits

6.4.1 Vocabulary lists by frequency for specific domains

6.4.2 Morphological analysis and generative vocabulary

- Prefixes, suffixes, and roots. Developing an eye for picking up meanings automatically, simply by knowing what different parts of the word mean in general.

[7] Automate with script? Probably also outside scope of project.

7 Section 7: Pedagogical applications

7.1 Orthography for digital natives

7.1.1 Standardization of letterforms

- Reducing the learning load in the first few weeks of Hebrew: block scripts and cursive scripts.
- Possible in handwritten as well (just only writing in block)

7.1.2 Typing speed and writing speed

7.1.3 But the permanence of handwriting

- Tests

7.2 Examples of typing-related pedagogical aids for Greek

7.2.1 Learning the accentuation system

- Practicing the typing of accents while learning about the rule of contonation, morae, and recessive accents.

7.2.2 Common irregular verbs

- Practicing the typing of certain very common irregular verbs (like *eimi*, e.g.) while simultaneously learning their paradigms.

7.2.3 Practicing reading/speaking Greek; ”reading by typing”

- Practicing typing in general by pulling in Greek texts from Perseus as typing training material. Students could be encouraged to also read the texts out loud as they type them. (Not necessarily understanding the Greek, but getting to see how it sounds and flows).

8 Section 8: Concluding remarks

8.1 Specific implementation benefits

8.1.1 Who should make the switch to this system? Is this project really worthwhile?

8.1.2 The low opportunity cost for the next generation

8.2 Moving forward with more languages

8.2.1 Current project: focus on Greek with Hebrew as a foil

8.2.2 Possibility to expand much further

8.3 Suggestions for further research

8.3.1 Corpus generation

8.3.2 Morphological analysis

8.3.3 Graphical frontends for customization

8.3.4 System APIs for keystream manipulations *across platforms*

8.3.5 AI autograders for language exercises

9 Section 9: Appendix

9.1 Integrating general electronic/online resources into classes

9.1.1 Language input as a pain point

- A lack of good keyboard input is a significant damper to the use of electronic/online resources.

9.1.2 The value of electronic/online resources

Elecronic lexica and morphology parsers

Dangers of over-reliance, but great benefits all the same. Arbitrary searches (those that require the ability to type native text) can be necessary when using paper sources rather than cross-linked sources like those on Perseus.

Searches

- Fuzzy search (i.e., lemma search), finding passages and references, searching on word usage or specific form.
- Searching typed notes, if people type class notes

Electronic flashcards

More polarizing whether or not they are useful, but making them easier to construct is definitely a good thing. Spaced repetition studying, Anki.

Autograded sentences

- Practicing typing in general by providing form-fields to enter sentence translations. Depending on the difficulty of implementation, it might be possible to create an autograder for practice sentences in Athenaze, for example. If care was taken to follow vocabulary acquisition (so as to limit the lexicon input for the program and make it deterministic), it would be easy for professors to design supplemental/optional practice exercises that the students could complete with instant feedback and no extra work for the professor.

9.2 Word Processing

9.2.1 Font testing: Gentium Plus + SBL Hebrew

Here is some inline Hebrew from the beginning of Genesis 1 בְּרֵאשִׁ֖ית בָּרָ֣א אֱלֹהִ֑ים אֵ֥ת הַשָּׁמַ֖יִם וְאֵ֥ת הָאָֽרֶץ׃ 2 וְהָאָ֗רֶץ הָיְתָ֥ה תֹ֙הוּ֙ וָבֹ֔הוּ וְחֹ֖שֶׁךְ עַל־פְּנֵ֣י תְה֑וֹם וְר֣וּחַ אֱלֹהִ֔ים מְרַחֶ֖פֶת עַל־פְּנֵ֥י הַמָּֽיִם׃ 3 וַיֹּ֥אמֶר אֱלֹהִ֖ים יְהִ֣י א֑וֹר וַֽיְהִי־אֽוֹר׃ 4 וַיַּ֧רְא אֱלֹהִ֛ים אֶת־הָא֖וֹר כִּי־ט֑וֹב וַיַּבְדֵּ֣ל אֱלֹהִ֔ים בֵּ֥ין הָא֖וֹר וּבֵ֥ין הַחֹֽשֶׁךְ׃ with English around it. And now a block:

> בְּרֵאשִׁ֖ית בָּרָ֣א אֱלֹהִ֑ים אֵ֥ת הַשָּׁמַ֖יִם וְאֵ֥ת הָאָֽרֶץ׃ 2 וְהָאָ֗רֶץ הָיְתָ֥ה תֹ֙הוּ֙ וָבֹ֔הוּ וְחֹ֖שֶׁךְ
> עַל־פְּנֵ֣י תְה֑וֹם וְר֣וּחַ אֱלֹהִ֔ים מְרַחֶ֖פֶת עַל־פְּנֵ֥י הַמָּֽיִם׃ 3 וַיֹּ֥אמֶר אֱלֹהִ֖ים יְהִ֣י א֑וֹר וַֽיְהִי־אֽוֹר׃
> 4 וַיַּ֧רְא אֱלֹהִ֛ים אֶת־הָא֖וֹר כִּי־ט֑וֹב וַיַּבְדֵּ֣ל אֱלֹהִ֔ים בֵּ֥ין הָא֖וֹר וּבֵ֥ין הַחֹֽשֶׁךְ׃

And here is some inline Greek from the *Iliad* μῆνιν ἄειδε θεὰ Πηληϊάδεω Ἀχιλῆος with English around it. And now a longer chunk:

> μῆνιν ἄειδε θεὰ Πηληϊάδεω Ἀχιλῆος οὐλομένην, ἣ μυρί᾽ Ἀχαιοῖς ἄλγε᾽ ἔθηκε, πολλὰς δ᾽ ἰφθίμους ψυχὰς Ἄϊδι προΐαψεν ἡρώων, αὐτοὺς δὲ ἑλώρια τεῦχε κύνεσσιν οἰωνοῖσί τε πᾶσι, Διὸς δ᾽ ἐτελείετο βουλή, ἐξ οὗ δὴ τὰ πρῶτα διαστήτην ἐρίσαντε Ἀτρεΐδης τε ἄναξ ἀνδρῶν καὶ δῖος Ἀχιλλεύς. τίς τ᾽ ἄρ σφωε θεῶν ἔριδι ξυνέηκε μάχεσθαι;

9.2.2 Reasons why something other than Word might be desirable

- Automatic font use rather than manual switching

9.2.3 Example: Emacs' Org mode to PDF using XeLaTeX

- Support for RTL languages and automatic display
- Polyglossia
- Automatic font switches

9.2.4 Yudit?

{Todo: [8]}

9.3 Abbreviations

- More of a personal thing. Can algorithmically generate in theory. (Outside scope of this project).
- Probably good to look at the 10 or 15 most common words and see if anything jumps out at you
- Creating regex hotstrings in this particular AHK implementation.

10 Works Cited

Chen Liao & Pilsung Choe (2013) Chinese Keyboard Layout Design Based on Polyphone Disambiguation and a Genetic Algorithm, International Journal of Human–Computer Interaction, 29:6, 391-403, DOI: 10.1080/10447318.2013.777827

Malas, Tareq M., Sinan Taifour and Gheith A. Abandah. "Toward Optimal Arabic Keyboard Layout Using Genetic Algorithm." (2008).

Knisbacher, Jeffry M., and הכתב העברי, "DESIGN CONSIDERATIONS IN THE USE OF HEBREW AND OTHER NON-ROMAN SCRIPTS ON IBM-COMPATIBLE COMPUTERS." Proceedings of the World Congress of Jewish Studies (1989): 61-68. `http://www.jstor.org/stable/23535305`.

[8]Need to research more.

Deemer, Selden. ”REPORT ON THE ARABIC LANGUAGE IN COMPUTERS SYMPOSIUM.” MELA Notes, no. 23 (1981): 11-13. `http://www.jstor.org/stable/29785130`.

Mastronarde, Donald. ”Before and After Unicode: Working with Polytonic Greek.” Montreal APA Unicode Presentation, 2008.