# Relative Keyboard Input System*

**Daniel R. Rashid**
Language Technologies Institute
Carnegie Mellon University
drashid@cs.cmu.edu

**Noah A. Smith**
Language Technologies Institute
Carnegie Mellon University
nasmith@cs.cmu.edu

## ABSTRACT
This paper describes a "relative keyboard," where keystrokes are treated as inputs in a continuous space relative to each other, instead of a discrete, unambiguous sequence. A user with the ability to touch-type may type anywhere on the sensing surface without the need for a visual keyboard. An implementation of such a system is explored and evaluated on simulated data and real user data.

### Author Keywords
relative keyboard, soft keyboard, predictive keyboard

### ACM Classification Keywords
H5.2 [Information interfaces and presentation]: User Interfaces.

## INTRODUCTION
There have been many solutions to keyboard input proposed for a wide range of hardware devices. With the exception of more exotic systems such as the Non-Keyboard [1], these usually take the form of a physical or digitally displayed keyboard that shows the user the location of all the keys. These systems are not ideal for mobile situations because they increase the mass of the hardware that must be carried, or they require screen real estate for visualizing the keys.

An alternative to these systems (for users who can touch-type) would be a software system that allows the user to type on a surface with no visible keyboard, relying solely on their ability to touch-type. Such a system would take up very little screen space on a tablet PC, or could be paired with a sensor to allow the user to type on any flat surface [4]. To accomplish this goal we treat the user's input as "relative," meaning the information gathered from the inputs consists of the relative distances between keystroke locations, rather than their absolute positions. An easy way to visualize this representation is as a directed graph that can be overlaid on a keyboard at various positions; depending on the overlay position, different strings will be generated. This paper explores the feasibility of such a "relative" input system, describing our implementation and evaluation of one design.



## RELATED WORK
Several prior designs are related to ours. The Non-Keyboard seeks to eliminate the keyboard and replaces it with gloves which can sense only pressure and not location [1]. Missing location information creates ambiguity in the input stream; the system must resolve which key each keystroke was intended to press, given the finger that did the pressing. Our system is presented with a complementary problem: the location of a keystroke is known, but we have no information regarding which finger was used to generate the input or the pressure used.

Another related system is the Canesta Projection Keyboard [4], which projects a keyboard onto a surface and uses a sensor module to record user input. This design is portable, and was shown in user studies to result in the lowest fatigue ratings when compared to a mechanical keyboard, a thumb keyboard, and the graffiti input system. There has also been a lot of work in improving accuracy using techniques such as pattern matching [3] and language models [1, 2]. Here we address a general form of the same problem, avoiding the projection aspect (therefore introducing more ambguity in the input).

## SYSTEM OVERVIEW
We describe an implemented system that carries out word prediction, and seeks to tackle the central challenges of resolving ambiguity in the positioning of the hypothetical keyboard and mis-strokes by the user. We formulate the problem as follows. The input is a sequence $\{(x_i, y_i)\}_{i=1}^{n}$ of $n$ absolute keystroke positions on a two-dimensional surface. The output is a word $w$, or more generally a ranked list of words $\{w_j\}_{j=1}^{k}$ from which the user may choose. We are assuming for now there is another form of interaction which allows the user to notify the program that a word has ended, or select a word from a list. Future versions of this system will attempt to tackle higher-level issues such as segmentation of an ongoing coordinate stream into a sequence of words.

Given $\{(x_i, y_i)\}_i$, the system uses an internal representation of the keyboard to compute the most likely number of keys separating two positions in the horizontal and vertical directions. We call this the sequence of offsets from the starting point, denoted $\{(h_i, v_i)\}_i$, where $h_1 = v_1 = 0$. Once all the

points are converted to offsets relative to the first keystroke, a set of strings can be generated corresponding to words the user may have intended to type. For example, if the offset sequence generated was $\{(0,0),(1,0)\}$ (i.e., the second point is 1 key to the right and in the same row) a few of the possible strings generated are *qw*, *we*, *er*, *rt*, *ty*, *yu*. These are derived by treating the keyboard as a grid and placing the starting key at all possible keys on that keyboard. For single letters there will be as many possible strings as keys on the keyboard, but for more complex offset patterns the number will be reduced under the assumption that the widest span in a sequence must fit on the keyboard. Many of the possible combinations are not real words, so filtering the results though a dictionary will significantly decrease the number of likely words. The logical steps of the prediction system are:

2D Point Generation: Given an input specific to the hardware of the sensing surface, translate that input into a sequence of two-dimensional points in a coordinate space, $\{(x_i, y_i)\}_{i=1}^n$.

Offset Generation: Given a sequence of 2D points in the input coordinate space, find the most likely offset sequence, $\{(h_i, v_i)\}_{i=1}^n$.

Dictionary Matching: Given an offset sequence, find the dictionary words that approximately match. Here we perform a simple kind of error correction since words may be chosen which don't exactly match the offset sequence.

List Ranking: Order the list of words by plausibility.

## OFFSET PATTERN UNIQUENESS

Since offset sequences are relative to the first keystroke and not based on absolute positions, we are forced to accept that collisions of words mapping to the same pattern will occur. However, generating the patterns of 109,000 word types in English (taken from a dictionary) on the standard QWERTY layout, over 99.5% of patterns generated by English words were found to be unique (see Table 1).

| Collisions | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Words | 109133 | 310 | 75 | 20 | 10 | 0 | 14 | 0 |

**Table 1. Number of collisions in a list of 109,000 English words taken from an electronic dictionary.**

This high percentage show that the ambiguity introduced as a result of treating the inputs as relative has the potential to be overcome. However, we will see that overcoming the user error coupled with this ambiguity is more difficult.

## INPUT REQUIREMENTS AND 2D POINT GENERATION

Our system could be paired with any of a wide variety of hardware input devices, including cameras, sensor systems, or touch screens. Touch screens are the simplest (the system provides absolute screen coordinates of "touches" to the software), but for other hardware systems, a coordinate sequence $\{(x_i, y_i)\}_i$ must be provided. In the current system, it is assumed that there is no *rotation* of the input coordinates; the user must always position her hands at the same angle relative to the input surface. However, in the future lifting this constraint could be useful in some settings.

## KEYBOARD MODEL

To understand the offset generation stage we must first discuss the keyboard model. The keyboard model is the internal representation used to simulate the keyboard. The user never interacts directly with this model since we do not know how the input space maps to the keyboard's internal space, given the location of the keyboard in the input space is unknown. It is used primarily in computing the offsets, where we will see it is not required to know the alignment of the spaces in order to compute the offsets, which are relative values.

First, we define the virtual keyboard. Let $\mathcal{K} = \{q, w, e, r, ...\}$ be the set of keys on the keyboard. The first component models the noise in the typing process using a probabilistic model over $\mathbb{R}^2$, for the internal keyboard space. This model views keystroke positions as random events generated by a different stochastic process for each key. Associated with each key $k \in \mathcal{K}$ is an elliptical Gaussian distribution over positions:

$$p(x, y \mid k) = \mathcal{N}\left(\begin{bmatrix} x \\ y \end{bmatrix}; \begin{bmatrix} \mu_{k,x} \\ \mu_{k,y} \end{bmatrix}, \begin{bmatrix} \sigma_{k,x}^2 & 0 \\ 0 & \sigma_{k,y}^2 \end{bmatrix}\right) \quad (1)$$

$$= \frac{1}{2\pi\sigma_{k,x}\sigma_{k,y}} \exp - \left[\frac{(x-\mu_{k,x})^2}{2\sigma_{k,x}^2} + \frac{(y-\mu_{k,y})^2}{2\sigma_{k,y}^2}\right]$$

In this research, we define $(\mu_{k,x}, \mu_{k,y})$ to be the center of the rectangular region associated with the key $k$, and we define $\sigma_{k,x}$ and $\sigma_{k,y}$ to be a constant value based on the desired key size for all $k$.

The second component encodes a representation of a keyboard on a grid. It is a mapping of the form

$$F : \mathcal{K} \times \mathbb{Z} \times \mathbb{Z} \to \mathcal{K} \cup \{*\} \quad (2)$$

where $\mathcal{K}$ is the set of keyboard keys and $\mathbb{Z}$ the set of integers. $F$ maps keys $k$ and two-dimensional offsets $(h, v)$ to other keys, or to $*$ in cases where there is no key at the resulting position (i.e., the move is "out of bounds"). For example, $F(n, -1, -1) = g$ and $F(c, 1, 11) = *$.

## OFFSET GENERATION

The goal of offset generation is to take the input coordinates $\{(x_i, y_i)\}_{i=1}^n$ and map them to offsets using the keyboard model. This breaks down into a decision about the values of $(h_i, v_i)$ given $(x_1, y_1)$ and $(x_i, y_i)$.[1][2] The offsets are chosen according to:

$$(h_i, v_i) \leftarrow \underset{(h,v)\in\mathbb{Z}^2}{\operatorname{argmax}} p(h, v \mid x_1, y_1, x_i, y_i) \quad (3)$$

[1]We note that in the current system each offset for $i \in \{2, ..., n\}$ is generated independently of the others, but it would be possible to define a system that jointly infers the offset values without making this independence assumption. This would incur a runtime cost.

[2]We simplify the notation here by ignoring scaling transformations on $\{(x_i, y_i)\}_{i=1}^n$ to account for differences between the physical plane where the user types and the keyboard space, assuming that they are on the same scale.

To calculate this quantity for a given $(h, v)$, we need to consider all possible values of the first key and $i$th key.

$$\begin{aligned}
&\operatorname*{argmax}_{(h,v)\in\mathbb{Z}^2} \operatorname*{argmax}_{\substack{k_1,k_i\in\mathcal{K}\\ F(k_1,h,v)=k_i}} p(h, v, k_1, k_i \mid x_1, y_1, x_i, y_i) \\
&\quad = \operatorname*{argmax}_{(h,v)\in\mathbb{Z}^2} \operatorname*{argmax}_{\substack{k_1,k_i\in\mathcal{K}\\ F(k_1,h,v)=k_i}} p(k_1, k_i \mid x_1, y_1, x_i, y_i) \\
&\quad = \operatorname*{argmax}_{(h,v)\in\mathbb{Z}^2} \operatorname*{argmax}_{\substack{k_1,k_i\in\mathcal{K}\\ F(k_1,h,v)=k_i}} \begin{pmatrix} p(x_1, y_1, x_i, y_i \mid k_1, k_i) \\ \cdot p(k_1, k_i) \end{pmatrix} \\
&\quad = \operatorname*{argmax}_{(h,v)\in\mathbb{Z}^2} \operatorname*{argmax}_{\substack{k_1,k_i\in\mathcal{K}\\ F(k_1,h,v)=k_i}} \begin{pmatrix} p(x_1, y_1 \mid k_1) \\ \cdot p(x_i, y_i \mid k_i) \\ \cdot p(k_1, k_i) \end{pmatrix}
\end{aligned}$$

The first equality holds because $h$ and $v$ are fully known if $k_1$ and $k_i$ are known. The second equality uses Bayes' rule (modulo a marginal in the denominator, which is constant with respect to $h$ and $v$). The third equality assumes the coordinates for each key are generated independently, given the keys. We next assume that $(x_1, y_1)$ are the center of $k_1$'s region so that $p(x_1, y_1 \mid k_1)$ is a constant, and that the key pairs are uniformly distributed, so that $p(k_1, k_i)$ is a constant.[3] This gives:

$$(h_i, v_i) \leftarrow \operatorname*{argmax}_{(h,v)\in\mathbb{Z}^2} \operatorname*{argmax}_{\substack{k_1,k_i\in\mathcal{K}\\ F(k_1,h,v)=k_i}} p(x_i, y_i \mid k_i) \qquad (4)$$

We wish to use the Gaussian model (Equation 1) associated with $k_i$ to compute $p(x_i, y_i \mid k_i)$, but we cannot use $x_i$ and $y_i$ directly since the keyboard and input spaces are not aligned. However, given our assumption that $(x_1, y_1)$ is at the center of $k_1$'s region, we know this point in the keyboard space is equal to $(\mu_{k_1,x}, \mu_{k_1,y})$. We then use this value and the distance between the points $(x_1, y_1)$ and $(x_i, y_i)$ to compute the new point in the keyboard space:

$$\begin{aligned}
x'_i &= \mu_{k_1,x} + (x_i - x_1) \\
y'_i &= \mu_{k_1,y} + (y_i - y_1)
\end{aligned}$$

We can then compute $p(x'_i, y'_i \mid k_i)$ directly from Equation 1.

## DICTIONARY MATCHING

Here we group the logical steps of string generation, dictionary filtering, and error correction together which permits us to accomplish all the tasks at once fairly easily. To do this, we cache all dictionary entries with their equivalent offset sequences. After that we can define a distance between two arbitrary offset sequences. Here we consider two distances.

The first, $\Delta_{\text{sub}}$ is strict but fast to compute. We assume that the length of the intended word (in characters) exactly matches the number of registered keystrokes. We align the hypothesized offset sequence, $H = \{(h_i, v_i)\}_{i=1}^n$, with the candidate word's offset sequence, $W = \{(\bar{h}_i, \bar{v}_i)\}_{i=1}^n$, and calculate the total Manhattan distance:

$$\Delta_{\text{sub}}(H, W) = \sum_{i=1}^{n} \left|h_i - \bar{h}_i\right| + \left|v_i - \bar{v}_i\right| \qquad (5)$$

For candidates whose length does not match the hypothesized sequence, we let $\Delta_{\text{sub}} = +\infty$.

A second distance, $\Delta_{\text{sub/ins/del}}$, permits insertions and deletions of keystrokes with respective costs; it is the minimum Manhattan distance between the two offset sequences. Computing it requires $O(|H| \cdot |W|)$ runtime, which in our current implementation is too slow for real time. However, because it is likely that errors will include insertions and deletions, we believe it is appropriate to consider this distance in our experiments.

Given a hypothesized offset sequence, we compute $\Delta_\bullet$ for each dictionary entry. Since we are not explicitly generating strings from $H$, and only comparing it to dictionary entries, dictionary filtering is done by default.

## RANKING CRITERIA

The final step is to choose an ordered list of the most plausible words to present to the user. To do this, we use a linear combination of three scores:

Frequency: The relative frequency of a word (also called its "unigram probability") estimated from a large corpus of English, models the tendency of users to type more common words.

Location Data: Here we assume that the user is typing in a consistent location, and maintain an estimate of where we believe the location of the keyboard currently is for the user. This can be computed based on the word/input pairs we have observed in the past. For a candidate word we can then compute how closely it matches our estimate of the keyboard's current position.

Error-Distance: The distance between the hypothesized offset sequence and the word's offset sequence ($\Delta_{\text{sub}}$ or $\Delta_{\text{sub/ins/del}}$) gives penalties to words whose offset sequences differ too sharply from the given word.

## PERFORMANCE EVALUATION

We carried out two experiments. In the first, a simulated user was used to generate noisy 2D point sequences $\{(\tilde{x}_i, \tilde{y}_i)\}_i$. For each of 100 random words chosen uniformly from the dictionary we added random Gaussian noise to the keystroke centers, with a standard deviation up to a key height (or width). Different standard deviation values were tested. The relative keyboard's output on the simulated data is shown in Figure 1. We note that even with fairly high levels of noise ($\sigma = 0.5$), the system correctly predicts the intended word over 70% of the time. Considering the top three candidates (which could be presented in a small menu to the user for quick selection) improves accuracy by at least ten points at all noise levels $\sigma > 0.2$.

Our second experiment uses data gathered from real users. We created a set of 160 words so that there were ten words

[3] These assumptions make our system simple; in future versions we will explore how a model estimated from data can improve performance.

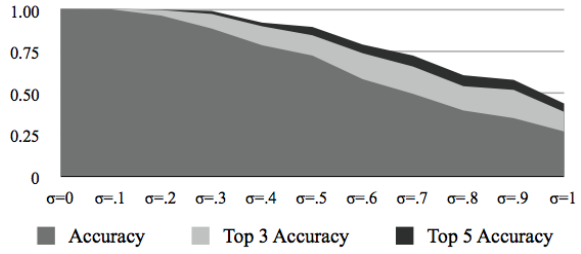


**Figure 1. Synthetic data with added noise: Accuracy of top 1, 3, and 5 hypotheses, with varying levels of noise. An average over five trials is shown.**

randomly selected from each of 16 length/frequency classes.[4] Users were presented with a single word at a time on a screen and told to type the word; each user was tested in both a baseline physical keyboard condition and a touch screen condition. After typing the word, the user hit the spacebar or a spacebar "button" on the touch screen to move to the next word. No feedback was given, and the backspace key was not available to correct mistakes. Half of users performed the physical keyboard test first, and the other half used the touch screen first.

Note that this experiment does not depend on our system at all; the accuracy of competing word prediction models can be computed offline and compared to the accuracy of the physical keyboard. So while this experiment gives us little information about the relative keyboard's usability, it does let us measure accuracy of variants of our prediction model without repeated data-gathering.

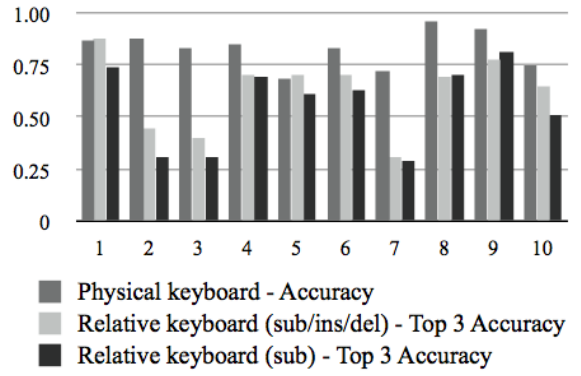


**Figure 2. Accuracy of typing, by word, for ten users: physical keyboard accuracy, accuracy of top three hypotheses for the relative keyboard prototype using $\Delta_{\mathrm{sub}}$ and using $\Delta_{\mathrm{sub/ins/del}}$. Naturally, top $t$ accuracy rises and falls directly with $t$.**

The average accuracy across ten users in the physical keyboard condition was 0.832 (range 0.688 to 0.963). Using $\Delta_{\mathrm{sub/ins/del}}$ for dictionary matching, our system achieves 0.485 on average (range 0.213 to 0.782); this average falls to 0.456 when we use the faster but stricter $\Delta_{\mathrm{sub}}$ (range 0.213 to 0.688). There is a wide variation in the accuracy achieved by different users, so we show the trend by user (Figure 2). The top 3 accuracy of six out of ten users, using $\Delta_{\mathrm{sub/ins/del}}$, is within 10% of their physical keyboard accuracy, with two of the ten showing a slight improvement over their physical keyboard results.

We note that this is a preliminary study meant to assess the accuracy of our prototype system and help target future development. These results suggest that for many users, a more robust distance measure will improve accuracy at the dictionary matching step. We also suspect that many users could improve accuracy with practice (none of the users tested were given the opportunity to use the relative keyboard before the experiment). In the future, more thorough usability studies will be required, but here we demonstrate a proof of concept for software-driven prediction of keystrokes in context and show that, at least for some users, the relative keyboard is a viable text-input system.

[4]A length/frequency class is defined by a word-length (here, 4, 6, 8, or 10 letters) and a frequency decile in a very large corpus (we used the first, third, fifth, and seventh deciles).

## FUTURE RESEARCH

This paper presented a prototype relative keyboard that predicts single words; in the future we intend to build a system which will handle capitalization, auto-segmentation of words, and intuitive interfaces for online typo correction and selection of words. More sophisticated language models might take advantage of a word's context in prediction. We believe generative probabilistic models will serve as a framework for learning accurate predictors and also for user adaptation (e.g., adaptive scaling or distortion of the keyboard to better fit a user's internal model, reducing errors). We also anticipate a more in-depth study of usability and design.

## CONCLUSION

This paper presents a simple implementation of a new idea for handling keyboard input. By treating the inputs as "relative" we have created a transparent keyboard layer for use in many different types of systems. Possible applications range from mobile phones that permit any nearby surface to be used as a keyboard to touch screen computer interfaces that do not require extra screen real estate. Pairing novel hardware solutions with sophisticated prediction algorithms has the potential to lead to high-accuracy, portable text input systems driven by intelligent software.

## REFERENCES

1. M. Goldstein, R. Book, G. Alsiö, and S. Tessa. Non-keyboard QWERTY touch typing: a portable input interface for the mobile user. In *Human Factors in Computing Systems*, 1999.

2. J. Goodman, G. Venolia, K. Steury, and C. Parker. Language modeling for soft keyboards. In *AAAI*, 2002.

3. P.-O. Kristensson and S. Zhai. Relaxing stylus typing precision by geometric pattern matching. In *IUI*, 2005.

4. H. Roeber, J. Bacus, and C. Tomasi. Typing in thin air: the Canesta projection keyboard—a new method of interaction with electronic devices. In *Human Factors in Computing Systems*, 2003.