# GestKeyboard: Enabling Gesture-Based Interaction on Ordinary Physical Keyboard

**Haimo Zhang***
National University of Singapore
13 Computing Drive, Singapore
z-haimo@comp.nus.edu.sg

**Yang Li**
Google Research
Mountain View, CA 94043, USA
yangli@acm.org


**Figure 1: Gesturing on a physical keyboard. The green overlays show the keys that have been pressed and the red ones show the currently pressed keys. The red line running across keys illustrates the captured gesture stroke.**

## ABSTRACT

Stroke gestures are intuitive and efficient but often require gesture-capable input hardware such as a touchscreen. In this paper, we present GestKeyboard, a novel technique for gesturing over an ordinary, unmodified physical keyboard—that remains the major input modality for existing desktop and laptop computers. We discuss an exploratory study for understanding the design space of gesturing on a physical keyboard and our algorithms for detecting gestures in a modeless way, without interfering with the keyboard's major functionality such as text entry and shortcuts activation. We explored various features for detecting gestures from a keyboard event stream. Our experiment based on the data collected from 10 participants indicated it is feasible to reliably detect gestures from normal keyboard use, 95% detection accuracy within a maximum latency of 200ms.

### Author Keywords

Physical keyboard; gesture detection; modeless interaction.

### ACM Classification Keywords

H.5.2 [User Interfaces]: Input devices and strategies. I.5.2 [Design Methodology]: Classifier design and evaluation.



*CHI 2014,* April 26 - May 01 2014, Toronto, ON, Canada

ACM 978-1-4503-2473-1/14/04…$15.00.
http://dx.doi.org/10.1145/2556288.2557362

## INTRODUCTION

Gesture-based interaction is intuitive and efficient for many interaction tasks, and has become a promising interaction modality on various new-generation devices such as smartphones, tablets and tabletops. However, gesture-based interaction has not seen significant adoption in traditional computing environments and devices such as desktop computers or laptops. One important reason is that either traditional devices lack appropriate gesturing input mediums or existing input devices have been occupied for other purposes.

Existing work that enables gesture-based interaction on a traditional computing device usually introduces additional hardware [6][8][11]. Although it is possible to use existing path-making devices such as a mouse or a trackpad for gestural input, an explicit mode switch is often required to disambiguate the gestural input from other uses of these devices such as target acquisition.

In this paper, we present GestKeyboard, a technique that allows a user to perform gestures on an ordinary, physical keyboard. It requires no additional hardware, allows modeless switching between gesturing and the designed use of a keyboard—text entry and shortcut activation, and reduces homing effort—users can complete many tasks without having their hands to leave the keyboard switching to other input devices.

Previously, physical keyboards have been augmented with additional sensors for various purposes such as pressure-sensing [3] or touch display keyboards [2], and gesture support on keyboards has not been the focus of prior work. In contrast, we intend to enhance a regular keyboard for

* This work was done during an internship with Google Research.

gesture input with no physical modification or additional hardware. Our basic idea is straightforward. We view each key on the keyboard as a binary sensor that tells if a key is pressed, and the entire keyboard as an array of such sensors. By tracking when each key is pressed and released, we can recover a low-resolution trajectory of finger movement on the keyboard that can be fed to a gesture recognizer for interpretation.

However, enabling such a gesturing capability is not without challenges. The keyboard provides a limited spatial resolution and when sliding the finger on the keyboard, key events are often noisy. For example, not every key on the finger travel path is registered and a key press might generate multiple events. In addition, we need to quickly differentiate gesturing from regular keyboard uses especially text entry. In this paper, we contribute the following:

- A user study for understanding gesturing and regular keyboard behaviors and soliciting subjective feedback on gesturing on the keyboard;
- A method for detecting gesturing from typing on the keyboard that achieves high accuracy (95%) within reasonable latency (maximally 200ms);
- A process for transforming a keyboard event sequence into a spatial trajectory for gesture recognition and an adaptation of a template-based recognizer;
- Two examples of GestKeyboard that demonstrate its use at both the platform system and the application level.

In the rest of this paper, we first discuss the characteristic of physical keyboards and clarify our research problems. Two challenges in realizing GestKeyboard: gesture occurrence detection and gesture recognition. We then describe our user study, by which we collected user data for investigating these problems as well as acquiring user feedback on the usefulness of keyboard gestures. Next, we discuss our solutions and report their performance. We present two applications of GestKeyboard and discuss its limitation and future work. Finally, we discuss related work and conclude the paper.

## TECHNICAL PREAMBLE

We started our investigation by understanding our target gesture sensing device—existing physical keyboards, including its hardware characteristics, its event representation and geometry that transforms an array of binary events into a spatial trajectory.

### Keyboard Hardware

Most keyboards detect key presses and releases via a matrix of circuits [4][7], consisting of columns and rows of wires that are not in contact with each other. Keycaps are located at the intersections between column and row wires. When pressed, the keycap would bring the respective column and row wires into contact. The location of the key press is therefore detected by successively setting a high voltage on each column wire, and detecting the high voltage signal on each row wire. This scheme creates two issues, "ghosting" and "masking", when multiple pairs of column and row wires are connected at the same time, which creates ambiguous key presses. "Ghosting" refers to a key press being incorrectly reported even when it is not pressed, due to multiple other keys being pressed simultaneously (usually more than three keys) that short circuit the column and row wires underneath the "ghosting" keycap; and "masking" refers to the pressing and releasing of the "ghosting" key not seen by the keyboard controller, which considers the "ghosting" key to be always pressed.

The "ghosting" and "masking" phenomena can be avoided by using diodes, which prevents the wires from being shorted via other pressed keys. However, this mechanism is still absent in most keyboards. For GestKeyboard to be usable for as large an audience as possible, we decide to focus on single-finger gestures to avoid the ghosting and masking effect, since a single finger is less likely to press more than three keys simultaneously.

### Keyboard Events

In most operating systems, a keyboard event encapsulates information about one single change of the keyboard state, which includes the timestamp that change occurs, the keycap that change involves, and the type of that change. The time is often in milliseconds or microseconds, the keycap is often uniquely identified with a scancode, and the type can be press, release, or repeat of that keycap. Although much of the details are system and hardware dependent, this generic model of keyboard event suffices for our discussion of GestKeyboard.

### Keyboard Geometry

To capture spatial movement on a keyboard for gesture detection, we model two aspects of a specific keyboard: the bounding box of each keycap relative to the top-left corner of the keyboard and the binary adjacency between keycaps. By modeling the keyboard geometry, we are able to calculate various features that are useful for gesture detection and recognition. For example, we can calculate the shortest distance between two consecutive key press events. Due to the existence of large keycaps such as the space bar or the "+" key on the numeric pad, the shortest distance between two keycaps cannot be simply calculated as the distance between their centers. Instead, we first "thin" each keycap as a line segment and then calculate the shortest distance between segments (see Figure 2).

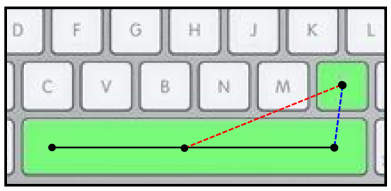


**Figure 2: Centroid distances (red line) versus segment-segment shortest distance (blue line) for non-square keys such as the space bar ("thinned" as the black line).**

## RESEARCH PROBLEMS

There are two important aspects of the problem. First, we need to understand how users would perceive gesturing on a physical keyboard regarding its usefulness and usability including the ergonomic issues. Second, we need to address the technical challenges regarding gesture detection and classification to yield a usable system.

### User Perception on Keyboard Gestures

As we introduce gesturing as a new behavior to the keyboard, it is important to understand how users perceive its usefulness and what potential applications or interaction scenarios are for keyboard gestures. We hope to understand if it is feasible to use a physical keyboard as a gesture articulation device ergonomically. A keyboard does not offer a smooth surface and is not as easy or comfortable as a touchscreen or touchpad for sliding fingers above it. If a user presses her finger heavily, it is difficult to slide it across keys but if a user does so lightly, keys might not be registered and a trajectory will not be captured completely.

### Gesture Detection & Classification

Due to limited spatial resolution of keys on a keyboard and possible sensor failures such as ghosting and masking of keys, we focus our research on single-finger unistroke gestures and assume that typing and gesturing occur in an interleaved way but not in parallel.

Both regular keyboard uses such as typing and keyboard shortcuts and gesturing produce keyboard events, with unobvious differences, and they can be highly interleaved. To allow a user to freely transition between these two types of behaviors without explicit mode switching, we need to determine if the user is gesturing from a stream of keyboard events—the gesture detection issue. Our solution needs to meet two criteria.

To not interfere with the major function of the keyboard--text entry and shortcut activation, our detection algorithm needs to be *highly accurate*. Missing a gesture—false negatives—would result in unintended operations such as gibberish characters entered as text and unintended hotkey combinations invoked that changes the system behavior. Capturing a gesture mistakenly—false positives—would lead to unresponsive interfaces, e.g., typed characters do not appear in a text editor, or unintended gesture execution.

In addition to accuracy, a related issue is the *decision latency*, i.e., the time needed for reliably detecting a gesture occurrence. For example, the user expects to see typed text to appear in a text editor with as little latency as possible. As a result, the detection algorithm should try to reject gesturing as early as possible, to allow for normal dispatching of keyboard events as for typing, of course, without missing intended gestures.

Once a gesture occurrence is detected, we need to recognize the gesture for invoking corresponding actions, given a sequence of keyboard events that are identified by the gesture detection phase. We currently focus on a template-based recognition approach that matches an unknown gesture against a predefined gesture set, because the approach is easy to implement and highly customizable.

We intend to find out if the low resolution of spatial trajectories captured by the key events would hamper the recognition accuracy. Because we aim at supporting both directional (e.g., sliding downwards for scrolling down) and symbolic gestures (e.g., "?" for help), which are useful for assisting common interaction tasks, we need to enhance existing template-based recognizers to support both types of gestures.

## USER STUDY

We started our exploration by conducting a user study, to understand user behaviors and perception on keyboard gestures and to collect training data for gesture detection and classification. To facilitate our investigation, we selected a set of 16 target gestures by adapting a previous gesture set [13] (see Figure 3). In particular, we try not to include gestures with complex trajectories that can be inappropriate for a low-resolution gesture-sensing device—the physical keyboard.

Note that both directional and symbolic gestures are included in the gesture set. In particular, the four straight gestures are sequence-sensitive, e.g., if the articulation direction of the two horizontal gestures is reversible, they would become indistinguishable. The remaining (symbolic) gestures in the set are sequence-invariant. For example, a circle gesture should allow for both clock-wise and counter clock-wise articulation sequences.

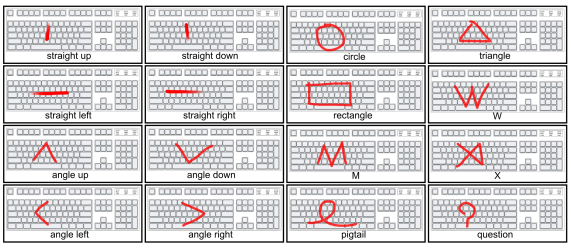


**Figure 3: The proposed gesture set.**

### Keyboard Conditions

We expect the physical form of different keyboards to be an important factor for the applicability of GestKeyboard. On keyboards with a large vertical travel distance (the height of a key) and rugged keycap profile, such as the HP SK-2880 keyboard (Figure 4-a), sliding the finger on the keyboard would require the finger to laterally squeeze against the sharp edge of the keycap, in order to activate the keys along the path of the gesture. Performing a gesture on this kind of keyboard might induce uncomfortable physical experience and hinder speed and accuracy. In contrast, gesturing might be easier and more comfortable on keyboards with a small travel distance and smooth keycap profile, such as the Apple MB110LL/B Keyboard (Figure 4-b).

To observe the effect of different keyboards on gesturing experiences, we used two types of keyboard in our study,

the HP SK-2880, representing the mainstream PC keyboards (thereafter referred as *full-size* keyboards), and the Apple MB110LL/B, representing the keyboards commonly found in laptops and Mac computers (thereafter referred as *compact* keyboards).

**a) HP SK-2880 keyboard ("full-size" keyboard)**

**b) Apple MB110LL/B keyboard ("compact" keyboard)**

**Figure 4: The two types of keyboards used in our study.**

### Experimental Tasks & Procedures

We recruited 10 participants (2 females and 8 males, all right-handed, aged between 19 and 40, M=27). The participants had a various background including software engineers, project managers and attorneys. Half of the participants used full-size keyboards primarily, while the other half used compact keyboards more often.

To understand how the key events generated by gesturing on the keyboard are different from those by regular usage, our experiment involves two tasks: *gesturing* and *typing*. For brevity, we use typing to refer to regular uses of keyboard such as text entry using alphanumeric, punctuation, and the space keys and key combinations served as shortcuts, such as Ctrl+C/Ctrl+V for copy/paste, and meta characters, such as delete, home/end keys, and platform-dependent meta keys such as Macintosh's Command key. Each participant was asked to use both types of keyboard for typing and gesturing. The order of the keyboard use is counterbalanced. In each keyboard condition, a participant first completed a typing task and then a gesturing task.

Our typing task captures a typical text entry scenario. Participants were asked to type a plain-text document in English, and then follow the instructions to stylize the document using various keyboard shortcuts (see Figure 5). Participants were instructed to complete the typing task as naturally as possible, with their natural typing speed and shortcut habit, e.g., the preferred modifier location such as left versus right "Shift" key and the landing order in a key combination such as Ctrl+Alt+Del versus Alt+Ctrl+Del.

The typing task used a different document (with a similar complexity) in each keyboard condition such that participants do not enter the same document twice. Each document had about 1000 characters that involved about 50 unique characters and 4 different formatting styles from a set of bold, italic, hyperlink, center-aligned, underline and right-aligned. We provide a cheat sheet of 14 shortcuts for text editing and formatting to the participants.

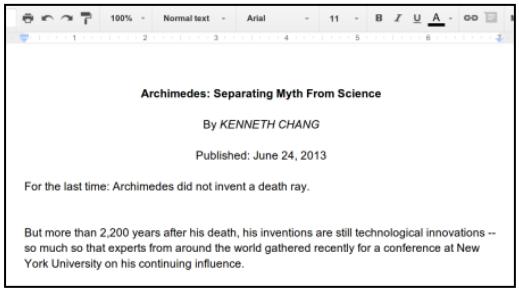


**Figure 5: An example of our typing task.**

In the gesturing task, participants were asked to perform keyboard gestures, by following an on-screen stimulus that shows one of the 16 gesture in Figure 3. Participants were informed that the presentation of the gesture in the stimuli is for the illustration purpose only, and they should perform the gesture in the way they are most comfortable with, regarding the size, speed, and location of the articulation on the keyboard. The visual presentation of the gesture is hidden once participants start gesturing to refrain them from precisely replicating the gesture. Participants were allowed to redo a trial if a mistake happens. The occurrence of the 16 target gestures is randomized with five repetitions for each gesture in total.

### Experimental Apparatus

A desktop computer with Intel i7 processor and 16G RAM running Ubuntu 12.04 LTS operating system was used in the study. A low-level key logger recorded all the keyboard events throughout the study by reading from the /dev/input/event* file, which Linux uses to expose input events. The timestamps for the beginning of each trial and gesture stimuli were recorded, which allow us to segment and label the continuous keyboard event stream as chunks of typing and gesturing data as well as corresponding gesture categories in post processing.

### Results

In total, we collected 1600 gesture samples (10 participants x 16 gestures x 5 repetitions x 2 keyboards), from which we excluded 6 samples that participants mistakenly entered (clearly belong to other categories).

We conducted a paired two-sample T test on the task completion time. On average, a typing task took 4162 seconds (SD=142 seconds) on the full-size keyboard, and 4607 seconds (SD=162 seconds) on the compact keyboard. $t(9)=3.0$, $p=0.015$. In contrast, for gesturing, the compact keyboard allowed a significantly faster gesture articulation speed (M=768ms, SD=385ms) than the full-size keyboard

(M=979ms, SD=604ms), t(9)=2.941, p=0.016, after log-transformation. The effect of keyboard conditions on the number of key presses generated from each gesture is not significant (p>0.05), with a mean of 10.2 keys, SD=4.2 (see Figure 6), which roughly reflects the resolution of these keyboards on sensing gestures.

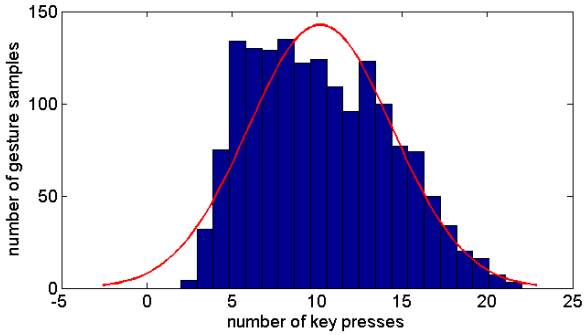


**Figure 6: The distribution of the number of key press events generated from each gesture.**

*Gesture Articulation Characteristics.* To understand how participants perform these gestures on the keyboard, we analyzed the articulation characteristics of the gesture samples. Although the four straight-line gestures have a designated direction, we deliberately left the articulation direction of the rest gestures undefined. For example, a circle can be drawn in either a clockwise or a counter clockwise fashion. This gives us an opportunity to observe users' preferences in performing these gestures on a physical keyboard.

We model the profile of gesture articulation direction by projecting each gesture stroke segment (a primitive finger motion as sensed by the keyboard) of collected samples onto three separate dimensions: horizontal, vertical and angular. For example, in the horizontal dimension, a movement can be left to right, right to left or no movement.

We found that gesturing tended to involve more left-to-right (59.4%) than right-to-left motion (40.2%), more downward (31.9%) than upward motion (21.3%) and more clockwise (58.9%) than counter clockwise motion (34.3%). Because all the participants in the study were right-handed, this observation implied that the participants preferred to contact the keycaps using the soft finger pad, instead using the tip or back of the finger.

We observed that the participants adopted various alternative articulation strategies in performing the gestures, such as using the soft pad on the tip of the index finger for a downward gesture, and using the back of the nail for an upward gesture. Several participants used the non-gesturing hand to hold the keyboard for stabilization while gesturing, while others completed the gestures single-handedly.

These observations have several implications for designing a keyboard gesture set. First, since participants tend to use their finger pad in most cases, it is important to have handedness in mind. For example, a left-to-right straight gesture might be easy for right-handed users, but uncomfortable for left-handed users. Second, it is advisable to design for gestures without much curvature in the path, since curvature would require the use of different parts of the finger to perform the gesture, which could be less comfortable. Third, some gestures might be easier to perform with alternative articulation strategies, such as using the thumb. It is useful to make users aware of these alternative strategies for increased performance and comfort.

*Subjective Feedback.* We solicit the feedback from participants to understand their perspective on the potential use of gesturing on the keyboard. Overall, the participants were excited about being able to issue gestures on a regular keyboard. In particular, the participants suggested several scenarios where GestKeyboard can be useful, such as shortcuts for editing a document while in a rush, web navigation using directional gestures (e.g., backward or forward) and shortcuts for launching specific applications. This feedback inspired us to develop two applications to showcase GestKeyboard that we will discuss later.

All the participants preferred using the compact keyboard over full-size keyboard. They felt that they were more accurate, more comfortable, and less frustrated when performing gestures on the compact keyboard, which is consistent with the gesture completion time we discussed earlier. Two participants felt that these keyboard gestures will shoulder part of functionality of existing keyboard shortcuts. In particular, keyboard gestures can be performed in an eye-free fashion.

## DETECTING GESTURE OCCURRENCES

Because we intend to allow users to switch between gesturing and typing without explicit mode switching, we need to detect the occurrence of a gesture from a continuous stream of keyboard events. A naïve strategy is to assume every key event to be a potential start of a gesture. However, this will require us to run gesture detection on every key event that induces a constant overhead for event processing. Instead, we view the event stream as a sequence of typing and gesturing segments and between two consecutive segments, the keyboard is idle, i.e., no key is pressed for a certain interval. Ideally, the "heavy-lifting" of executing the detection algorithm should only happen at the start of each segment. Once the segment type is determined to be typing or gesturing, the subsequent keyboard events can simply carry the same segment type and be dispatched immediately until another idle stage occurs. Therefore, our first task in gesture occurrence detection is to segment the event stream.

### Segmenting the Keyboard Event Stream

We segment a keyboard event stream based on if the idle stage before a new event (signals the start of a segment) or after the previous event (signals the end) is larger than a certain interval. Based on the gesturing data collected from our user study, on average, a gesture contains 3.7 idle stages for the full-size keyboard, and 2.5 for the compact keyboard. Based on the cumulative distribution of all idle

durations (see Figure 7), we use the duration value at the 99% cut-off point on the empirical CDF as the timeout interval, which turns out to be 420ms for the full-size keyboard and 120ms for the compact keyboard—only 1% of the idle stages would lead to incorrect segmentation.

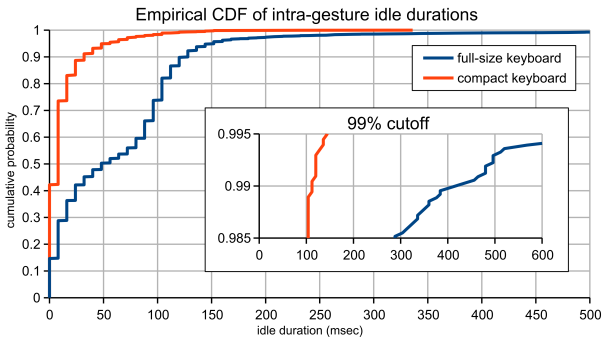


**Figure 7: The empirical Cumulative Density Functions of intra-gesture idle durations. Inset shows the 99% cutoff.**

These acquired timeout intervals will be applied to a continuous event stream at runtime for identifying the starting point of a segment and invoking gesture occurrence detection algorithms. For our offline analysis, we use these timeout intervals to segment the typing data into chunks, which are served as negative samples for training a gesture occurrence detector. Applying the timeout intervals on typing data results in 1,493 typing samples for the full-size keyboard, and 2,327 samples for the compact keyboard.

### Creating a Gesture Occurrence Classifier

With the typing and gesturing samples generated in the previous steps, our gesture detection task reduces to a binary classification problem. To create an efficient classifier, we first look into various features that characterize gesturing versus typing behaviors, including both spatial and temporal aspects.

#### *Keystroke Transition Distances*

One obvious observation about gesturing on a keyboard is that consecutive keystrokes (key presses) in the stream tend to be close in distance, as the fingers slide across keys. In contrast, two consecutive keystrokes in typing can be quite distant on the keyboard depending on the target word or shortcut being entered.

To capture this difference, we look at both the mean distance $M_K$ between consecutive keystrokes and its variance for the $K$ keystrokes that have been observed so far in an unknown segment:

$$M_K = \frac{\sum_{i=2}^{K} D(E_{i-1}, E_i)}{K-1}, \quad K \geq 2 \qquad (1)$$

where $E_i$ is the keycap name of the $i^{th}$ keystroke on the keyboard, and function $D(E_{i-1}, E_i)$ returns the segment-to-segment shortest distance between two keycaps, based on the specific keyboard geometry (see Figure 2). The variance, $V_K$, is in turn calculated as:

$$V_K = \frac{\sum_{i=2}^{K} |D(E_{i-1}, E_i) - M_K|^2}{K-1}, \quad K \geq 2 \qquad (2)$$

At this point, we can directly use these features to participate in the process for training a gesturing-typing classifier, based on our data. However, our typing data, from the two typing tasks in the study, is hardly inclusive for describing all possible keystroke transitions in day-to-day typing behaviors. To address this issue, we used an English language model (http://invokeit.wordpress.com/) to synthesize the typing data that can better reflect keystroke transitions in typing based on the follow procedure.

We start with the $N$-gram-frequency pairs from the language model, where the value of $N$ ranges from 2 up to 5. Then for each $N$-gram, we generated the corresponding keystroke sequences. It is possible that an $N$-gram could be entered in multiple ways, due to alternative modifier key locations. For instance, either of the two "Shift" keys might be used to enter uppercase letters such as "A" or "?" character. Lacking empirical data on the probability of such alternative keystroke sequences, we simply generated all possible keystroke combinations for each $N$-gram with an equal frequency, summing up to the $N$-gram frequency. Finally, for each such synthesized keystroke sequence, we computed the mean distance and variance for the first $K$ keys—$K$ ranges from 2 to 5.

Note that the synthesized dataset is only useful for capturing keystroke transitions and it lacks other properties of the behavior that can only be observed in actual typing, such as temporal aspects. As a result, we only use the synthesized dataset to determine an optimal threshold for $M_K$ and $V_K$, to discretize these features. We also computed $M_K$ and $V_K$ for $K$ from 2 to 5 for each gesture sample from a dataset that we collected for the 16 gestures in a pilot study—that is separate from the user study reported earlier.

For each $K$, we want to determine an optimal threshold that can best separate gesturing from typing based on $M_K$ or $V_K$. We achieve so through an ROC (Receiver Operating Characteristic) analysis on $M_K$ and $V_K$ for gesturing and synthesized typing (see Figure 8 for $M_K$ for the compact keyboard). Note that each typing $M_K$ and $V_K$ is weighted based on the frequency of their originating $N$-gram in the ROC analysis. Based on these thresholds, we discretize $M_K$ and $V_K$ as $M_k'$ and $V_k'$ by assigning 1 if they are above their threshold and 0 otherwise.

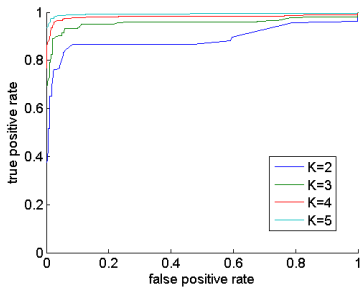


**Figure 8: The ROC curves for mean keystroke distances for gesturing on the compact keyboard.**

*Adjacency Rates*

Following on the keystroke transition distance feature family, we look into a similar but different aspect—the adjacency of two consecutive key presses. Note that two keystrokes that are spatially close are not necessarily adjacent. It is noticed that for gesturing, consecutive key press events often happen for adjacent keys on the keyboard, whereas it is not the case for typing. This spatial characteristic is captured as the adjacency rate, $A_K$, for the first $K$ key press events in a sequence:

$$A_K = \frac{\left|\left\{E_i \middle| E_i \in Adj_{i-1}, i \in [2,K]\right\}\right|}{K-1}, \quad K \geq 2 \qquad (3)$$

where $E_i$ is the same as Equation 1, and $Adj_{i-1}$ is the set of keycaps adjacent to the keycap of the $i$-1$^{th}$ key press event, which is acquired from the keyboard geometry. The numerator therefore denotes the number of pairs of consecutive key press events that are also spatially adjacent in the first $K$ key presses. Note that a keycap should not be considered adjacent to itself: we expect gesturing to have a high adjacency rate and repeatedly pressing the same keycap, which is rare in gesturing but common in typing, should not increase the adjacency rate.

*Revisitation Rates*

Typing frequently involves the repetition of the same key, such as the space bar, or highly recurrent letters such as letter "e" in "succeeded". Gesturing, in contrast, rarely involves revisiting previously pressed keys—only when a gesture stroke intersects itself, e.g., the pigtail gesture. The keycap revisitation rate, $R_K$, is defined as the number of unique keycaps in the first $K$ keystrokes of an unknown segment over the total number of keystrokes, i.e., $K$.

*Idle Rates*

Lastly, we observed that the keyboard is hardly ever at an idle state while gesturing, since the fingers would remain pressed on some keys. However, the dominant pattern for finger motion while typing is discrete striking of keycaps that presses and releases a keycap in a short time interval. This temporal characteristic is captured as the percentage of time that the keyboard is idle—the idle rate, $I_K$—during the first $K$ key presses.

*Combining Individual Features*

Consider features generated from the same featurization process but for different $K$ values a feature family. With the five feature families, $M'_k$, $V'_k$, $A_K$, $R_K$, and $I_K$, with $K$ ranging from 2 to 5, a total of 20 features (4 lengths x 5 feature families) are computed for each sequence segment. If a segment contains fewer than $K$ key press events, the feature $F_k$, $F \in \{M', V', A, R, I\}$, is assigned a default value -1, to indicate a missing value.

Based on these features, we used the C4.5 decision tree of WEKA to predict if an unknown segment is gesturing or typing from the 2$^{nd}$ to the 5$^{th}$ key press event since the start of a sequence, a separate binary decision tree model is trained for each, using all available features up to that event.

Because segmented typing samples are significantly more than the gesture samples for each keyboard condition from our user study, we balance the dataset by duplicating gesture samples to roughly have the same size as the typing samples. Based on this dataset, we tested each model with a 10-fold cross-validation. Figure 9 shows the mean accuracy for each individual feature family and for the decision tree combination of them.

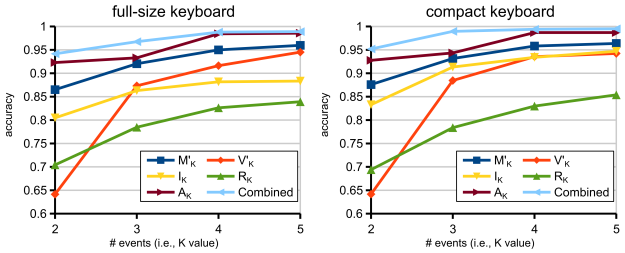


**Figure 9: The detection accuracy for the full-size and the compact keyboard for the number of key press events used.**

The results clearly show that adjacency rates ($A_K$) and mean keystroke distances ($M'_k$) are the more discriminative than other feature families. They require as few as two key press events to achieve over 85% accuracy. Idle rates ($I_K$) also consistently performed well. Combining all the five feature families with the decision tree outperformed each individual feature family with 94% accuracy based on only 2 key press events. By using 5 key press events and combining all the feature families, the decision tree reaches 99.5% and 98.9% accuracy on the compact and full-sized keyboard, respectively.

In terms of overall accuracy, using the full-size keyboard is almost as accurate as using the compact keyboard, which suggests the generalizability of the proposed features for different types of keyboard. One interesting observation is that, the type of keyboard has a more significant effect on the accuracy of the idle rate feature family than on other feature families. This phenomenon may be due to the large spacing and vertical travel distance for the full-size keyboards, which results in more idle stages even when gesturing, making it less distinguishable from typing.

**Making Time-Sensitive Decisions**

In the online situation, as keyboard events come in, our gesture detector should make a decision about if the user is gesturing or typing as soon as possible and at the same time minimize decision mistakes. The gesture detector is capable of making a decision once it sees two key press events, however, it needs to balance if it should make the decision now or wait until more events are observed.

A decision tree can give a confidence value for its classification result, and we need to determine an appropriate confidence threshold for accepting or ignoring a classification result. Although more events will lead to better accuracy (see Figure 9), it is important for the detector to resolve within a certain time limit. In addition, since keyboard events come in at an unexpected interval,

which is not time-bounded, we need to impose a resolution timeout in our decision process, i.e., if a resolution timeout is reached, the sequence is force-resolved to typing. For a given confidence threshold and resolution timeout, it is necessary to understand the performance of the gesture detector, with respect to elapsed time since the start of the sequence (see Figure 10).

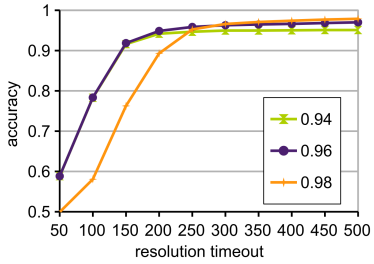


**Figure 10: The detection accuracy for different resolution timeouts and confidence thresholds on the compact keyboard.**

Each resolution timeout bounds the worst-case latency from the user starting to type to seeing the actions permanently committed to the system. We select 200ms resolution timeout and the 0.96 confidence threshold for our online detection process. This setting achieves 95% detection accuracy on both types of keyboard—on average 1 in every 20 detections is wrong, either a gesture sequence is detected as typing, or vice versa. Note that the 200ms latency only occurs when detection is triggered—a timeout followed by a keystroke, and could be reduced to tradeoff accuracy for responsiveness of the system.

## RECOGNIZING GESTURES

Upon identifying a gesturing segment from the keyboard event stream, we need to recognize the gesture. We first transform the keyboard events of the segment to a sequence of timed 2D coordinates and then classify the sequence with a template-based recognizer adapted from previous work.

If we consider each keycap as a binary sensor with two possible states: pressed or released, a keyboard event essentially describes a state change of one of these binary sensors. A key press or release event triggers the transition from one state to another, and a key repeat event has no change in the pressed state. In turn, the entire keyboard can be viewed as a low-resolution sensor array. Each individual sensor could be of large and irregular shapes. We use the keyboard geometry—the bounds of each keycap relative to the top-left corner of the keyboard—to transform a keyboard event stream into a timed 2D trajectory, i.e., an approximation of the finger's trajectory.

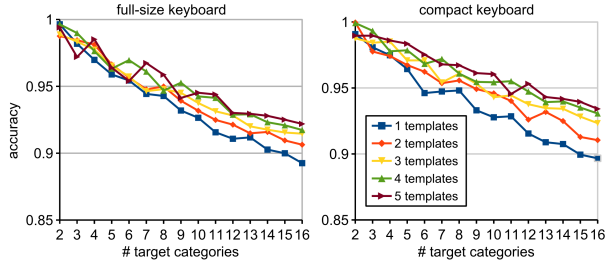


**Figure 11: The recognition accuracy for each keyboard.**

We maintain a set of currently pressed keys to reflect the current "footprint" of the user fingers, which is initially empty as we always start from an idle stage. Whenever a press or release event occurs, we add the key to the set or remove it from the set. For each update, we calculate the centroid of the footprint, $C$, as the following:

$$C = \frac{\sum \vec{p}_i a_i}{\sum a_i} \quad (4)$$

where $p_i$ is the bounding box center of the $i^{\text{th}}$ keycap in the footprint key set, and $a_i$ is its bounding box size. The centroid along the timestamp of the key event is appended to a 2D trajectory (see Figure 1).

We employ Protractor, a template-based gesture recognizer [9] for recognizing an acquired 2D trajectory, which matches a 2D trajectory against a set of templates. Protractor allows a developer to configure how sensitive the recognizer should be to orientation variation of a gesture. Because our gesture set involves gestures that would be indistinguishable from each other without considering orientation, e.g., four straight lines or gesture "M" versus "W", we set Protractor to be orientation sensitive.

Although several gestures in our target set are irreversible such as straight lines, many of our gestures are reversible, e.g., gesture "M" can be drawn from left to right or right to left. We could ask participants to perform additional gestures to capture both articulation directions. However, this would lengthen the study time and increase user fatigue and frustration. Instead, for each sample of a reversible gesture, we synthetically add its reverted trajectory to the template set. This allows the recognizer to support both types of gestures simultaneously without needing additional data. To find out the recognition accuracy of these keyboard gestures, we conducted a cross-validation splitting on participants—picking one participant's data for training and the rest 9's for testing and repeating the process for each participant. For each round, we varied the number of templates (training samples) from 1 to 5 and the number of target gesture categories (classes) from 2 to 16 (see Figure 11). Gesturing on the compact keyboard seems to be more accurate (M=95.3%, SD=2.5%) than on the full-size keyboard (M=94.5%, SD=2.6%), $t(df=74)=9.260$, $p<0.005$. For the most difficult classification condition (16 target gestures and 1 template gesture), the accuracy is 89.3%. When 5 templates per class is used to classify the 16 gestures, the accuracy is significantly increased to 92.2%. These results indicated that it is feasible to sense gestures on a physical keyboard.

## EXAMPLE APPLICATIONS

We developed two applications using GestKeyboard: one allows a user to invoke applications anytime and the other allows a user to edit text, using GestKeyboard gestures.

### System-Wide Deployment: GestRun

We developed an application launcher, named "GestRun" (see Figure 12), as a system-level service. Without going to a specific application (or the Desktop screen), a user can launch an application anytime, by performing a keyboard gesture. We leveraged shape mnemonics to ease learning of the gestures, such as the "circle" gesture for the Chrome browser, the "rectangle" gesture for terminal and gesture "V" for the Vim editor. GestRun provides a system-wide access to common applications, without consuming screen estate or requiring long click-throughs or search to find a specific application.

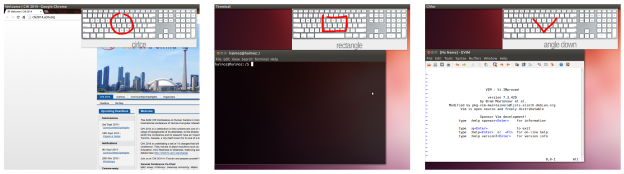

**Figure 12: The GestRun application launcher. The recognized gesture is shown above the corresponding application for the illustration purpose.**

### Application Specific Deployment: GestEdit

We also used GestKeyboard to enhance text editing. We developed GestEdit that allows a user to use keyboard gestures for common styling tasks (see Figure 13), such as vertical swipes for super or subscripting, and angle gestures for font style manipulation. Many of these styling operations have no shortcuts defined in existing text editors.

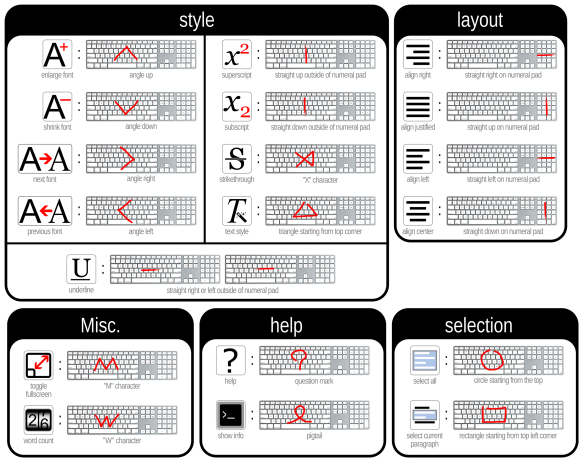


**Figure 13: The gesture commands used in GestEdit.**

In case a user is unsure which gesture to use, the user can pause the articulation by holding down the current keys. After a short dwell time (e.g., 500ms), an in-place popup menu is shown to present the gesture set and associated functions. At this point, the user can continue sliding her fingers to finish the target gesture. Alternatively, the user can cancel the ongoing gesture by pressing a key that is not adjacent to the current footprint,

GestEdit also leverages in which area of the keyboard a gesture is performed and invokes different functions for each area, e.g., the straight-line gestures map to alignment functions if they are performed on the numeric keypad.

## LIMITATIONS AND FUTURE WORK

GestKeyboard allows a user to perform a set of useful gestures on a regular physical keyboard that is equipped on every desktop and laptop computer. However, the ghosting and masking effect with most existing keyboard hardware prevents us from exploring gestures that require multiple simultaneous touch points, such as multi-touch or shape gestures [10]. In particular, shape gestures such as rolling the palm or fist across the keyboard can be easier to perform than stroke gestures that involve lateral friction.

Before gesture detection is resolved, there are two options to process the key events that have occurred for an unknown segment. GestKeyboard can hold on these events until the decision is made, and then dispatch them as well as the following events to either typing or gesturing, which continues until another segmentation timeout occurs and gesture detection is invoked again. This would incur a maximum of 200ms latency in interaction. Alternatively, GestKeyboard can dispatch each event right away but at the same time inform the application GestKeyboard has not yet made a decision. This removes the latency but shoulder the effort for handling uncertainty to application developers.

GestKeyboard itself is currently implemented as an application based on Linux. It is more appropriate for it to run at the core of keyboard event processing in the operating system, to take full control of the keyboard status. It is also worthwhile to implement GestKeyboard in a web environment, e.g., as a browser plugin so that it becomes instantaneously available to user on various computing devices that have a physical keyboard.

One important component in GestKeyboard is the keyboard geometry about the bounds of each key on the keyboard, which is often keyboard-specific. To automatically extract the geometry from a keyboard, we can potentially use computer vision and OCR to recognize each keycap from a photo of the keyboard.

Aside from the technical aspect, GestKeyboard's practicality should be further examined in the context of existing shortcut mechanisms, particularly in comparison with hotkeys. Keyboard gestures that are enabled by GestKeyboard have several potential advantages, including eye-free interactions, easy to learn and flexible to perform. Performing these gestures does not require the user to adapt for the specific keyboard layout or semantic designations of the modifier keys, which tend to significantly impact how a user performs hotkeys. Future research on these characteristics of keyboard gesture would provide direct implications on the usage scenario for GestKeyboard and further reveal the advantage and shortcomings of keyboard gestures.

## RELATED WORK

Using physical key presses to sense trajectories has briefly

been demonstrated previously. Jannotti [5] presented a scheme to simulate the shape of characters on the numeric keypad for text entry on a reduced keypad such as a TV remote. EdgeWrite uses four keys (arranged as a square) to capture spatial sequences for text entry [12]. All this prior work inspired us to explore gesturing on a physical keyboard in depth.

Extensive work has been conducted on gesturing on a soft keyboard for text entry based on a touchscreen or a stylus-enabled surface [14]. Although relevant, our work is vastly different because we focus on a different sensing device—an ordinary physical keyboard, which raises different challenges. For example, separating gesturing and typing on a touchscreen keyboard would be trivial because a touchscreen provides a much higher resolution than a physical keyboard does.

Traditional computing devices, such as a desktop or laptop computer, enabled limited gesture behaviors. For example, Apple MacBook allows a user to bring up Expose (windows overview) through a three-finger swipe gesture on the trackpad. However, stroke gestures are generally not supported on traditional computing devices that lack a dedicated gesturing device.

Previously, various efforts have been devoted to augmenting a keyboard with additional hardware [1][2][3]. In particular, the Pressure-Sensing Keyboard [3] is manufactured with additional press sensors in imprinting but its spatial resolution is unchanged. Touch Display Keyboard [2] enable flexible ways of using the keyboard, such as reassigning the keyboard functions or changing keyboard layout, by augmenting the keyboard with capacitive touch sensors and a projector instrumented in the environment. Actuated Keys [1] augments existing hotkey behaviors by mechanically raising keycaps when a modifier key is pressed. All this work did not focus on gesture articulation on the keyboard. In contrast, we aimed at enabling stroke gestures on existing unmodified keyboards.

## CONCLUSION

We present GestKeyboard, a novel technique for gesture-based interaction on an existing physical keyboard, without any additional hardware or modification. It does not interfere with existing keyboard usage and allows a user to switch between the regular usage—typing or performing hotkeys—and gesturing in a modeless way. Based on a user study with 10 participants, we explored various characteristics about gesture articulation on a physical keyboard. We contributed a set of algorithms for detecting gesture occurrences. In particular, we designed features that can effectively separate gesturing from typing by using typing behaviors synthesized from a language model. GestKeyboard is able to detect gesturing from regular typing, with 95% accuracy with a maximum latency of 200ms. Based on the experiment as well as two applications using GestKeyboard, we found gesturing on a regular physical keyboard is useful and feasible.

## REFERENCES


1. Bailly, G., Pietrzak, T., Deber, J., and Wigdor, D. Métamorphe: augmenting hotkey usage with actuated keys. In *Proc. CHI 2013*, ACM Press (2013), 563-572.
2. Block, F., Gellersen, H., and Villar, N. Touch-Display Keyboards: Transforming Keyboards into Interactive Surfaces. In *Proc. CHI 2010*, ACM Press (2010), 1145-1154.
3. Dietz, P., Eidelson, B., Westhues, J., and Bathiche, S. A Practical Pressure Sensitive Computer Keyboard. In *Proc. UIST 2009*, ACM Press (2009), 55-58.
4. Dribin, D. Keyboard Matrix Help. http://www.dribin.org/dave/keyboard/one_html/.
5. Jannotti, J. Iconic Text Entry Using a Numeric Keypad. Unpublished, 2002. http://pdos.csail.mit.edu/~jj/jannotti.com/papers/iconic-uist02.pdf.
6. Kane, S., Avrahami, D., Wobbrock, J., Harrison, B., Rea, A., Philipose, M., and LaMarca, A. Bonfire: A Nomadic System for Hybrid Laptop-Tabletop Interaction. In *Proc. UIST 2009*, ACM Press (2009), 129-138.
7. Keyboard Ghosting Explained. http://www.microsoft.com/appliedsciences/antighostingexplained.mspx. Retrieved September 15, 2013.
8. Leap Motion. https://www.leapmotion.com/. Retrieved September 15, 2013.
9. Li, Y., Protractor: A Fast and Accurate Gesture Recognizer. In *Proc. CHI 2010*, ACM Press (2010), 2169-2172.
10. Wigdor, D., Benko, H., Pella, J., Lombardo, J., and Williams, S. Rock & Rails: Extending Multi-Touch Interaction with Shape Gestures to Enable Precise Spatial Manipulations. In *Proc. CHI 2011*, ACM Press (2011), 1581-1590.
11. Wilson, A. Robust Computer Vision-Based Detection of Pinching for One and Two-Handed Gesture Input, In *Proc. UIST 2006*, ACM Press (2006), 255-258.
12. Wobbrock, J., Myers, B., and Rothrock, B. Few-Key Text Entry Revisited: Mnemonic Gestures on Four Keys. In *Proc. CHI 2006*, ACM Press (2006), 489-492.
13. Wobbrock, J., Wilson, A., and Li., Y. Gestures without libraries, toolkits or Training: a $1.00 Recognizer for User Interface Prototypes. In *Proc. UIST 2007*, ACM Press (2007), 159-168.
14. Zhai, S., Kristensson, P.O., Gong, P., Greiner, M., Peng, S.A., Liu, L.M., and Dunnigan, A. ShapeWriter on the iPhone – From the Laboratory to the Real World. In *Proc. CHI EA 2009*, ACM Press (2009), 2667-2670.