Swype Keyboard on Tiny Device Implementation Proposal

YUAN.Dengpan 20685430

December 20, 2019

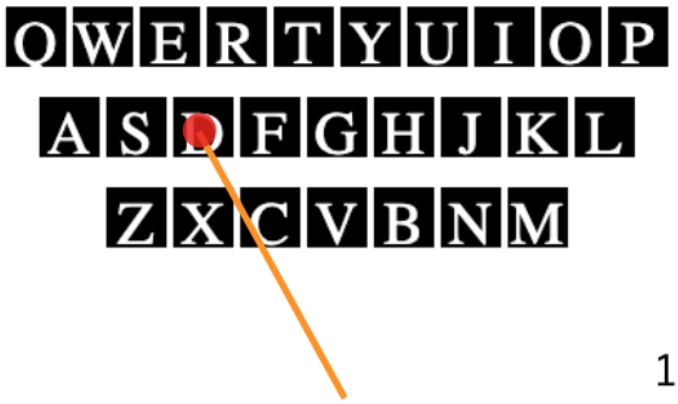
1 Introduction

This proposal will introduce the implementaion of the Swype keyboard on the usage of the tiny device. Four parts will be highly mentioned: DTW(Dynamic Time Wrapping) algorithm, the interaction of the existing keyboard layout, Data Retrieve, and words suggestion

2 Interaction with Current Keyboard

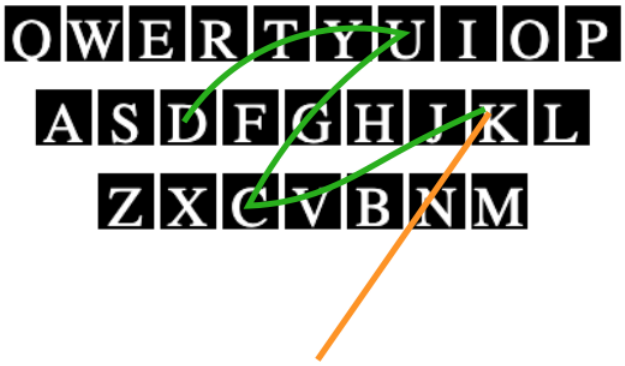
To make the illustration more easier, we will use the example of **duck** in this proposal if the user want to input through Swype keyboard.

2.1 Start Point and End Point



1

As a result of discussion, we agree that it should have a begging point selection process, when user control the bar at the beginning and stay at a certain key for 1 second, then the point turns red like figure1 (However, the point might not be located exactly on the perfect area of that certain key, this will be disccused in the following part *Point Estimation*). This indicator means that the keyboard has choosen the initial point and ready to draw a line.



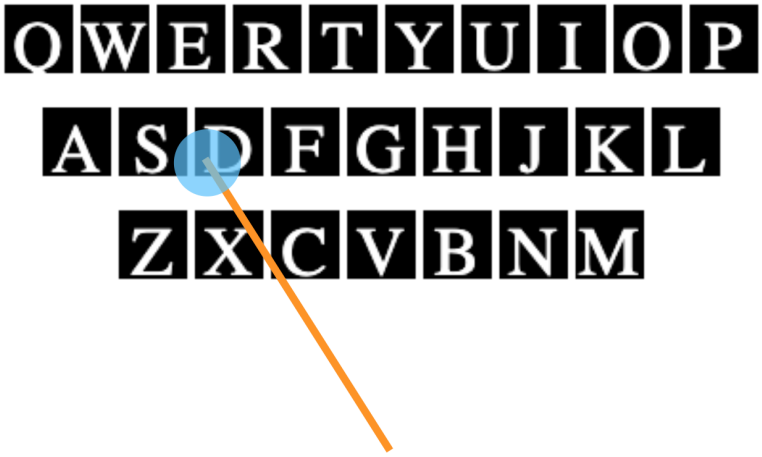
2

After the drawing, it should prsent a trace(with better transparency) for user to indicate an emperical path. And the end point is exactly when user's finger left the button/screen. P.S.,

the trace should have a strong inclination of the arc curve tendency because the operation is controlled by the bar with force. The predicted stoke is present on figure2.

2.2 Point Estimation

As a real problem, the tip of the bar mostly cannot fall on the perfect point (Filgure 3, center point of the diagonal of the key). So estimating which key the user actually want to input is important for providing the candidate keys. Besides, we should also prescribe the center of (x,y) position for 26 letters, these points will be servered to simulate the expected path later.



There should be an area as a continer of the error for user to input as acceptable erros. Figure4 is a visible blue circle error container to estimate the key inputs, the radius (9px) is the half long of the any key box, which is reasonble from the asepct of the distance. So this is the scenario of choosing the beginning point of **duck**, but tip locates on the left-down corner of **D**. As it is covered by container, we should also consider **S** as a candidate key.

Similarly, we should also monitor the points when swiping the path and add the new covered point as candidate key to form the candidate word for applying DTW Algorithm.

3 Algorithm

3.1 DTW Algorithm

DTW Algorithm is basically compring the simliarity of two sequences, but unlike other distance alogrithm, DTW has the advantage of comparing the sequence with the highest interaction parts. Technically speaking, this algorithm is a practical usage for suggesting words.

3.1.1 Avalibility in the Empirical and Expected Path

It should be informed that we will get a lot of the candidate words when finished the swiping, the only way we could comparing the path is the points (by converting the path into points will be explained later in *Data Retrieval*).

DTW Algorithm is avaliable when we are going to suggest words by comparing path. DTW Algoirhtm takes two list of points, one from emperical points, antoher one is expected points. we should apply DTW Algorithm for every single candidate word points with swipe points and getting cost, then ranking the cost as an evidence for suggesting words.

3.1.2 Algorithm Analysis

The cost of DTW is $O(MN)$, M is for the input size of emperical pairs, N is for the input size of candidate pairs. For M, the worst case is when user swiping all over the screen(suppose the resolution of the device is same as the iWatch 272 x 340), but the maximum input for N must be less than M, so it's roughly 8 billion computation. Even for the worst case, the current portable devices is capable to run such a polynomial complexity algorithm.

3.1.3 Simple Implementation

The algorithm in the code is pretty easy to understand and implement, it needs a cost funciton to compute the similarity (the lower, the better). And iterating every point pair with the cost function to generate a cost matrix. The right-down corner of index is the minimum cost, which means it is the minimum cost between the candiate path and the swipe path.

```
// Cost Function for Computing Similarity
function EuclideanDistance(pair_one, pair_two){
    var distance_square = Math.sqrt(Math.pow(pair_one[0] - pair_two[0], 2)
     + Math.pow(pair_one[1] - pair_two[1], 2));
    return distance_square
}

function DTR(expected_plist, real_plist){
    var row = expected_plist.length;
```

```
    var column = real_plist.length;
    var cost_matrix = [];
        cost_matrix.push([]);
    // Suppose the normal screen size is 1980X1080 px
    for(var i=1; i <= row; i++){
                cost_matrix.push([]);
        cost_matrix[i].push(100000);
    }
    for(var j=1; j <= column; j++){
        cost_matrix[0].push(100000);
    }

    cost_matrix[0].unshift(0);

    for(var i=1; i <= row; i++){
        for(var j=1; j <= column; j++){
            cost = EuclideanDistance(expected_plist[i-1], real_plist[j-1]);
            // Generate Cost Matrix
            cost_matrix[i][j] = cost + Math.min(cost_matrix[i-1][j],
            cost_matrix[i][j-1], cost_matrix[i-1][j-1])
        }
    }
    return cost_matrix[row][column];
}

// Test code
var expected_plist = [[3,4], [3,5], [2,6], [1,4], [2,2]];
var real_plist = [[4,4], [2,7], [1,4], [0,3]];
DTR(expected_plist, real_plist); // will final get a cost with 5
```

3.2 Candidate Words Generation Algorithm

As it mentioned, we should record all the candiate words for DTW algorithm, the candidate word could be formed with the candidate key. Here is a basic solution:

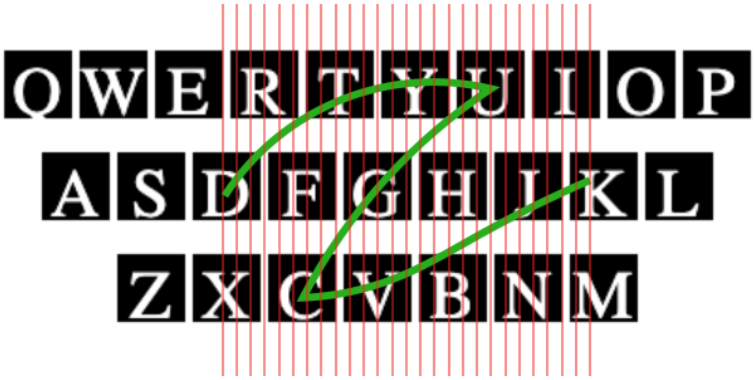
From the observation of the figure6, this is the middle process of swiping from **D** to **U**. There are simply 6 error containers to estimate the candidate words in order. There are two situations that candidate keys should be treated differently

For example, the container1 covers **S** and **D** initially, so it should create two brachs for the sequence. Another example is circle7, it only contains **U**, just simply add it to the sequence of previous formed candidate word **SDFRTY** and **DSFRTY**. The pseudocode could be written as:

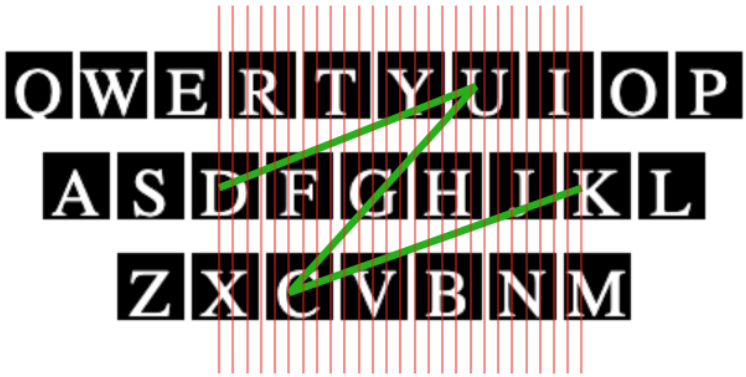
```
candidate_word_list <- []

while swipe not end:
    current_poi <- js.get_poistion()
    // get_candiate() should check the neighbor keys within coverage
    candidate_key_list <- get_candiate(current_poi)
    for i in candidate_key_list:
        for j in candidate_word_list:
            /* if the last letter of existing word
                doesn't equal to the candiate key,
                that means we meet a new key. Should
                be added to the end of the existing key.
            */
            if candidate_word_list[j][-1] != i:
                candidate_word_list[j] += i
```

4 Data Retrieve

The data in the keyboard is the path, which could be decompsed to the pairs of the points. However, how to convert the curve and the path into the pairs of points into the current layout of the keybord is a question. Here is a suggestive method:



It should be noticed that the pixel of the tip of the bar is 1px in the code, so we could use 1 px as the unit to form the pairs from the path. Figure5 is visulizing the technic of "cutting" the path into pairs, and interaction of red line and path should be recorded for later use. Recording the position of the point is fairly easy to do with javascript.



Different from the swiping path, we should convert all candidate words into points pairs. And then to compare the similarity.