University of Basel

Word-Gesture Typing in Virtual Realty

Bachelor thesis

Databases and Information Systems
Department of Mathematics and Computer Science
Databases and Information Systems
https://dbis.dmi.unibas.ch/

Examiner: Prof. Dr. Heiko Schuldt
Supervisor: Florian Spiess

Philipp Weber
phil.weber@stud.unibas.ch
19-051-697

20.06.2022

Acknowledgments

So Long, and Thanks for All the Fish. And the template.

Abstract

Text-entry is one of the most common forms of computer-human interaction and indispensable for many tasks such as word processing and some approaches to multimedia retrieval. The conventional keyboards everybody knows have long-established as the main text input method for desktop and laptop computers and even for touchscreen based devices they are very useful. But when it comes to virtual reality (VR) and augmented reality (AR), conventional keyboards might not be the best solution. With today's technology, VR and AR lack of tactile feedback and accurate finger tracking. As a result, text input for VR and AR is still an area of active research.

In recent years, word-gesture keyboards (also called slide-to-type keyboards) have been introduced in most major smartphone operating systems. Word-gesture keyboards look more or less like a conventional keyboard. But instead of tapping on the different keys, words are written with gestures. Now, the idea is, that they might also work well with VR/AR.

Table of Contents

1 Introduction

1.1 Motivation

Lots of virtual reality applications need some kind of text input method. While most of the time this function is provided by a conventional keyboard implemented for VR, this might not be the handiest one. To use this kind of keyboard, a user has to tap on single letter key to input single characters. Even though most of the text inputs in VR applications may not be very long, it can still be exhausting for our arms to input these. The conventional keyboard in VR applications often has a bigger scale than a normal keyboard in reality. That said, we have to move quite a distance with our arms and always move up and down to not accidentally hit a key.

Now, we want to develop and provide a keyboard, that is handy to use in VR. The idea, as stated in the abstract, is to develop a word-gesture keyboard. With a word-gesture keyboard, the text input could become more comfortable. We would not need to move our arms up and down, we could just move on a flat plane from one key to another. Such a keyboard could also be smaller, because the precision is not as important as it is for the normal keyboard. For example, on a conventional keyboard, if we would tap in the middle of two keys, we cannot really tell which one to take for the input. But with a word-gesture keyboard, where we work with distances and graphs (more on this later), it has not that much of an impact. Therefore, a smaller keyboard is possible, and we do not have to move our arms that much. Hence, it might be less exhausting to write with a word-gesture keyboard.

To show the power of such keyboards in a non-VR environment, we can make a comparison between the best possible performance on a conventional keyboard with the qwerty layout and a word-gesture keyboard with the ATOMIK layout. MacKenzie and Zhang [4] found, that after about 17 hours of practicing, the user of a conventional keyboard with qwerty layout could input about 45 words per minute. On the other hand, Zhai and Kristensson [2] had in their experiment with a word-gesture keyboard with the ATOMIK layout a record input speed of about 52 to 86 words per minute. This shows, that the potential of such word-gesture keyboards is high and one could really write faster after some training. Therefore, we may ask ourselves if this could also be an efficient text input method for VR/AR.

1.2 Goals

For this thesis, we have two main goals. The first one is to develop a word-gesture keyboard. It has to work with vitrivr-VR and has to be available as open-source. It should also be available as a Unity package, so other developers can use it in their Unity projects as well. The second goal is to evaluate said keyboard. The evaluation will be conducted according to current research standards with the usage of the MacKenzie phrase set.

2 Background

In this chapter we introduce the environment the word-gesture keyboard is mainly developed in and developed for, some things about conventional and word-gesture keyboards in general and SHARK[2].

2.1 vitrivr-VR and UnityVR

vitrivr[1] is an "open source full stack content-based multimedia retrieval system"[2]. It supports video, image, audio and 3D collections. It also features a very broad set of query paradigms that are supported. vitrivr was developed by the Database and Information Systems group[3] (dbis) of the university of Basel. For our thesis, we use the VR part of vitrivr, namely vitrivr-VR. This is being developed in Unity[4].
Unity is a tool for developers, where one can create projects in 2D, 3D and VR. To a certain degree, Unity is free to use. Developers can provide assets and Unity packages. These can be either free to use or have to be bought. Another developer then can import and use these in their own Unity projects. The main language used in Unity is C#. A developer can write such C# scripts and if needed attach them to objects in a scene. These scripts can control the objects and what they are doing when a user interacts with them or something particular happens.

2.2 Conventional Keyboard

In this thesis, when we talk about conventional keyboard, we do not look at its type of construction, if it's a mechanical keyboard or not. We also do not consider a special layout, when we talk about conventional keyboards. We define the term "conventional keyboard" as the most used keyboard type, the one where a user has to input every single letter by pressing or tapping a single key.

[1] https://vitrivr.org/
[2] https://dbis.dmi.unibas.ch/research/projects/vitrivr-project/
[3] https://dbis.dmi.unibas.ch
[4] https://unity.com

There are two different kinds of conventional keyboards. One is the physical variant. This one is used for most desktop and laptop computers. The other variant is a so called soft keyboard. This is an on-screen conventional keyboard that is mostly used with phones, tablets and other touchscreen-based devices. It has the same functionality as the physical conventional keyboard, but instead of pressing a physical key, one has to tap on the screen at the right place.

2.2.1 Disadvantages in VR/AR

When it comes to VR and AR, it seems, that this is not the best method to input text. One reason for this statement is, that right now, it lacks of tactile feedback and accurate finger tracking. While this could be improved during the next years, yet it is not really there. Another reason is the size of such keyboards in VR. We have to tap on the keys with our controllers. If the keys are too close together, it might cause a problem in recognizing which key the user wanted to press. Therefore, there needs to be either bigger keys or bigger spaces between two adjacent keys. This results in a bigger keyboard, which results in more needed movement with the arms. If we have to move our arms a lot to input some text, this can quickly become exhausting.

2.3 Word-Gesture Keyboard

A word-gesture keyboard may look pretty much the same as a conventional keyboard described in the last section, but works quite different. First of all, it does not exist in a hardware version like the conventional keyboard does. It is more like the soft keyboard version on a screen.

Independent of the details of the implementation, every word-gesture keyboard works with gestures. That means, instead of tapping on single keys, the user has to draw one line or a shape on the keyboard. This will then be evaluated by an algorithm, that determines the closest word, the one with the most similar shape seen from different aspects, from a lexicon. For example, to input the word ”science”, the user has to put the finger on the screen, where the ”s-key” is displayed. Then they have to move, with the finger still on the screen, to the respective adjacent key with the correct character. At the end, the user has to take away their finger from the screen at the ”e-key”. If the gesture was more or less good, the algorithm behind should now be able to calculate, that ”science” is the word, the user intended to write. But if the gesture is done bad, it can happen, that the wrong word is being calculated.

2.3.1 SHARK^2

SHARK^2 is a ”large vocabulary shorthand writing system for pen-based computers” [2] developed by Shumin Zhai and Per-Ola Kristensson. It can compare the user inputted graph with a perfect graph of any word in a given lexicon. A perfect graph is the graph, that is produced, if we start from the center of the word’s first letter on the keyboard. Then we draw a straight line to the center of the next letter of the word and so on, until we reach the last letter. $SHARK^2$ then can find the word with the most similar graph compared to the user input. To achieve this, it uses a multi-channel recognition system. Each channel alone from the system developed by Zhai and Kristensson [2] does not necessarily have enough power, but all the channels together can detect the right word. The most important part are the two core channels, a shape recognizer and a location recognizer, where different aspects of the graphs get looked at. There are also some other tricks the system uses to achieve the best possible results.

2.3.1.1 Preconditions

The SHARK^2 system needs a lexicon that should be in the order of 10'000 words. The lexicon can be obtained through different methods. For example, the lexicon used to test SHARK^2 was mined from one of the authors' emails, but it could also just be a standard dictionary. For all the words in the lexicon, also their perfect graphs have to be stored. This is not mentioned by the authors, but it would make sense, to only store N points for each graph, with N being the number of points that samples the graph.

2.3.1.2 Template Pruning

First of all, $SHARK^2$ uses template pruning. It compares the start and end positions of the perfect graph of each word in the lexicon with the input gesture from the user, both being normalized. If one of these two distances is bigger than a given threshold, the checked word will be discarded and not further considered. With normalized, the authors mean normalized in both, shape and location.

2.3.1.3 Shape Channel Recognition

The next step is to apply the shape recognizer. It compares the shapes of the perfect graph for every word in the lexicon and the user inputted graph. For this, an amount of N sampling points has to be calculated for every graph. These N points need to be equidistant. Then they have to be normalized in scale and location. This means, that the graphs are all normalized by scaling the largest side of the graph's bounding box to a predetermined length L:

$$s = \frac{L}{\max(W, H)} \tag{2.1}$$

W and H are the width and height of the graph's bounding box. The middle point m of every graph's bounding box now has to be calculated. Then, m has to be subtracted from every point, because this sets the middle point of the bounding box to $(0, 0)$, thus normalizes it in location. Then, all points' positions have to be divided by s to get the normalized positions. Now the distance between the normalized user inputted graph and every word's normalized perfect graph has to be calculated. To do so, we use the following formula:

$$x_s = \frac{1}{N}\sum_{i=1}^{N} \|u_i - t_i\|_2 \tag{2.2}$$

where u_i is the i-th point of the user inputted graph and t_i the i-th point of a word's perfect graph. This is the so-called proportional shape matching distance [2].

2.3.1.4 Why using more channels?

Now, one could think, that this is enough and with the application of the template pruning and shape channel recognition, the word is perfectly determined. This is not the case. The authors stated, that words can have a similar or even same shape as other words. They call these word pairs "confusion pairs". They found out, that for example on an ATOMIK layout with a lexicon of 20'000 words, there were 1117 pairs of words that have an identical graph, if the starting and ending positions are not considered. If these are also considered with the shape, there is still a total of 537 confusion pairs.

2.3.1.5 Location Channel Recognition

To avoid these, the authors were using a second channel, not for the shape, but for the location. For the following formulas and calculations, the normalization of the graphs in not needed anymore. As the name states, it is more about the location, where the graph lies in a coordinate system, than about its shape.

They use an algorithm that computes the distance of the user inputted graph u to the perfect graph t of every word in the lexicon. The location channel distance is defined as:

$$x_L = \sum_{i=1}^{N} \alpha(i)\delta(i) \tag{2.3}$$

where N is the number of points used to sample a graph. $\alpha(i)$ with $i \in (1, N)$ are weights for the different point-to-point distances, such that $\sum_{i=1}^{N} \alpha(i) = 1$. $\alpha(i)$ can be valued in various ways. For $SHARK^2$ the authors used a function, that gives the lowest weight to the middle point-to-point distance. For the other point-to-point distances, the weight increases linearly towards the two ends. $\delta(i)$ is defined through following formula:

$$\delta(i) = \begin{cases} 0, & D(u,t) = 0 \wedge D(t,u) = 0 \\ \|u_i - t_i\|_2, & \text{otherwise} \end{cases} \tag{2.4}$$

where u_i is the i-th point of u and t_i the i-th point of t. D is defined as:

$$D(p,q) = \sum_{i=1}^{N} \max(d(p_i,q) - r, 0) \tag{2.5}$$

r is the radius of a key and d is defined as:

$$d(p_i,q) = \min(\|p_i - q_1\|_2, \|p_i - q_2\|_2, \ldots, \|p_i - q_N\|_2) \tag{2.6}$$

For all these formulas N has the same definition as for formula 2.3. The "trick" the authors use these formulas for is pretty simple. They state, that they form something like an "invisible" tunnel of one key width that contains all keys used to write a certain word. A perfect distance score of zero is given, when all the sampled points of the user inputted graph lie within the tunnel of t. If this is not the case, the distance score for t with respect to the user inputted graph is set to the sum of the spatial point-to-point distances.

2.3.1.6 Channel Integration

With the two distances x_S and x_L, the most probable word, the user intended to write, can be calculated pretty good. The authors assume, that the distance from a user inputted graph to the perfect graph of a word follows a Gaussian distribution. This means, if the user inputted gesture has distance x to a perfect graph of a word y, the probability, that y is the intended word can be calculated using the Gaussian probability density function:

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right] \tag{2.7}$$

One important thing here is, that this calculation has to be performed two times per word, because it has to be done for x_S and x_L. $\mu = 0$ and σ can be obtained through training from large amount of data. σ can be seen here as the sensitivity of a channel. If for example σ is equal to one key radius, the words, whose perfect graphs have a greater distance to the user inputted graph than one key width, have practically zero probability of being the intended word. For SHARK^2 the authors used σ as parameter to adjust the weight of contribution of each channel.

The authors also use σ for further pruning. They discard all candidate words, whose distance x is bigger than 2σ. For the candidate words $w \in W$, that have not been discarded until now, the marginalized probability to be the intended word is:

$$p'(w) = \frac{p(x)}{\sum\limits_{i \in W} p(i)} \tag{2.8}$$

Note, that this calculation also needs to be performed twice per word, once for the location channel and once for the shape channel. The last step of the channel integration is to integrate the probabilities from the two channels using Bayes' rule:

$$c(w) = \frac{p'_S(w)p'_L(w)}{\sum\limits_{i \in W_S \wedge W_L} p'_S(i)p'_L(i)} \tag{2.9}$$

$c(w)$ is now the confidence score for the word w.

2.3.1.7 Further Steps To Improve The Results

Dynamic Channel Weighting by Gesturing Speed

The authors say, that the user can draw a graph either on visual guidance from the keyboard (looking for the next letter of a word on the keyboard) or recall from memory. A graph drawn by visual guidance results in a higher location distance score than a graph drawn from memory recall. If a user draws a graph by memory recall the location distance score will be poor and the focus lays more on the shape. Therefore, they suggest a dynamic

weighting of the two channels, that is to adjust the weighting of the channels according to the time needed to draw the graph. In general, graphs drawn by memory recall are faster than visual guided ones. Hence, the gesture completion time should tell, how heavy the location channel should be weighted in the final selection. The time to complete a graph for a word obviously depends on its length and complexity. The authors then use Fitts' law
IS FORMULA NECESSARY HERE?
to calculate the normative writing time for a graph of a word. They use this result together with the actual graph production time to modify σ used in formula 2.7.

Using Language Information

The authors achieved quite good performance with the two core channels, but there still might be conflicting words. To prevent these, the authors suggest to also use language context. For SHARK^2 smoothed bigrams are used as the language model. It is then used to rearrange the N-best list of words received before (the N words with the highest confidence score).

3 Implementation

In this section, we will introduce how we implement a word-gesture keyboard using Unity, a Python script and the SHARK^2 system.

3.1 Word Graph Generator

As previously discussed in section 2.3.1, to make SHARK^2 work, the perfect graphs for all words in a lexicon must be precomputed. Therefore, we need a script, that either creates or overwrites a file for every available keyboard layout. It has to write the words included in the lexicon together with the corresponding N sampled points of their graphs. Per line, such a file contains a word, then a certain number N of points from the word's perfect graph followed by the same points, but normalized. Normalized here means the same as mentioned in the background section 2.3.1.3.

To run the script, the user has to provide the name of the layout that they want to create the perfect graphs for. Additionally, they also have to write the name of the text file containing all the words, which is the lexicon. The script then either creates a new file named "sokgraph_*layout*.txt" or if already a file with this name exists, it deletes its content. Then it fills the file line by line as mentioned above. In further sections, we will call a file of this type "layout file"

The script can only be executed for one layout at a time. Hence, if there are more available layouts for our word-gesture keyboard, the user has to run the script for every single one, which can take a bit of time. To calculate and write a file for all about 10'000 words in our used lexicon, it takes about four to five seconds.

One thing the script pays attention to is, if a word can be written with used layout. If there are words in the lexicon, that can not be written with the given layout, the script skips this one. Hence, there will be no line in the file for said word.

3.2 Used Algorithm

For our word-gesture keyboard we use a simplified version of the algorithm used in SHARK^2. This means, we do also work with two core channels, a location recognizer and a shape recognizer, but do not implement everything from the algorithm used for SHARK^2. The shape recognizer is to calculate the deviation from the user inputted graph and a perfect graph from a word with respect to their shape. The location recognizer is for the same thing, but not with respect to the shape, but rather the position on the keyboard. When looking at the shape, we have to normalize the graphs in a specific way (2.3.1.3), so the position, where they exactly lie on the keyboard, does not matter. When looking at the location, we look at the graphs as they are, without normalizing or changing anything.

As in the SHARK^2 system we also use the start and end positions of the graphs as a first pruning method. The difference is, that for SHARK^2, the authors chose to normalize all the graphs in scale and translation before comparing the start-to-start and end-to-end positions. In our algorithm, we do not normalize the graphs, but just look at the start and end positions of a user input graph and a word's perfect graph. As threshold we set the width of a key. That means, if either the start-to-start or end-to-end position is bigger than a key width, the word to which this applies, gets discarded.

Another thing we implement differently is σ. For the channel integration formula 2.7, they used σ in SHARK^2 as a parameter. They determine its value by the gesturing speed (2.3.1.7). We do not use the gesturing speed in our algorithm. For the location channel (in the formula 2.7) we use as fixed value for σ the radius of a key TODO: IS IT WIDTH OR RADIUS?. For the shape channel (in the formula 2.7) we use a variable value. We take a value that equals the radius of a normalized key. That means, a small graph will have a bigger σ than a big graph, because we do normalize the graph's longer bounding box side to a fixed length. Therefore, a small graph gets stretched by it, whereby a big graph gets contracted.

As mentioned in the beginning of this section, we use a simplified form of the algorithm used in SHARK^2. That being said, we do currently not use any language information nor dynamic channel weighting by gesturing speed. We do not use any language information, because that would have gone beyond the scope of this project. The dynamic channel integration was not implemented by us, because in our opinion the value of sigma, as we chose it, is fine for the purpose of our word-gesture keyboard.

3.3 Functions

In this section, we will present the functions our word-gesture keyboard provides.

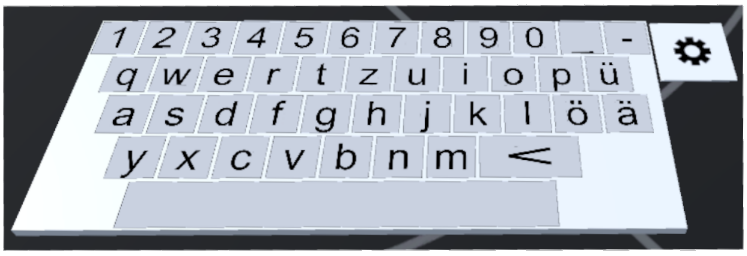


Figure 3.1: Word-gesture keyboard in its "normal" form without anything selected

3.3.1 Text input

There should be two text input methods for the user. One is the input of words with gestures, the other one is the input of single characters by tapping single keys.

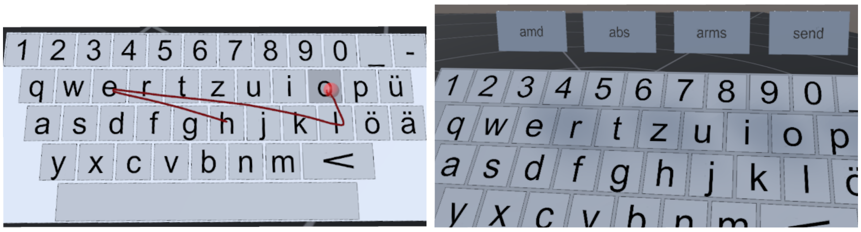


Figure 3.2: left: the user writes the word “hello” as gesture, right: user gets recommended words for “and”

3.3.1.1 With Gestures

The most important function our word-gesture keyboard provides, is the writing of words with gestures. A user can press and hold the trigger button of a VR controller inside the keyboard’s hitbox and start making a gesture on it. The user will see a red line, that is drawn on the keyboard where they gesture. This helps them to keep track of the line they drew. When the user wants to finish the gesture, they need to release the trigger of the controller. At this moment, our program starts to evaluate the 5 words with the best match to the user inputted graph. The one with the highest confidence score will be written into the text field. The other 4 are displayed at the keyboard (fig 3.2 right), such that the user can also choose between them. When they choose one of these 4 words, the word that has been written into the text field before, is replaced by the chosen word and the button, where the chosen word was written, will then display the replaced word.

3.3.1.2 As Single Characters

If the wanted word is not in the lexicon, there will not be any entry in any layout file. Therefore, our algorithm will not be able to get this wanted word as best match, hence it can not be written as gesture. For this case, we need a method to input single characters. Fortunately with our word-gesture keyboard this almost works without additional work. If the single letters are in the layout files, the user is more or less able to write single letters with a gesture. This “gesture” would just be a point on the right key, hence a single click at the right position. But there might be a little inconvenience. This is caused by the fact, that we work with distances. The distance from one letter to another is not big. And if for example the user wants to write an “e” but presses the key with the “e” on it on its left side and not perfectly in the middle, our system would also evaluate that aside from “e”, also “w” and “we” are words, the user might have intended to write (on a conventional qwertz or qwerty layout). To get rid of this, our system checks, if the user inputted graph’s bounding box is smaller than TODO: HOW MUCH SMALLER?. It recognizes, that the user wants to write a single letter, and then takes the best match. To get back to the example, “we” would be discarded and “e” would get a higher score than “w”, because of a smaller location channel distance. Therefore, the written “word” in this case would be “e”.

3.3.1.3 Spaces and Backspaces

Important when writing more than one word or characters are spaces. When the user writes a word that consists of more than one character, a space gets put behind the word TODO: MIGHT CHANGE WITH IMPLEMENTATION. This means, if they write a second word, they do not have to put a space manually. But if they write a single character, no space is put automatically.
When the user writes a word and then uses the backspace key, the whole word gets deleted. After this first word, only single characters will be deleted afterwards.

3.3.2 Create New Layouts

Another function is the creation of custom layouts. A text file for layouts exists, that contains all available layouts. The user can create as many new layouts as they want to. To create a new one, the user has to write the new layout’s name on the first new line. On the following lines they have to write the characters in an order, in which they want to have them on the keyboard. All the Unicode characters should be working, but two. In the current implementation, one whitespace is used to declare the position of the spacebar and the “<” character is used for the backspace key. This file gets read at the start of the program, so it cannot be edited while the program is running, or to be precise, the changes will not be recognized during runtime. One smaller thing we implemented is, that at the start of the program all characters used in the layouts not yet in the lexicon text file and layout files are being added. Without doing this, the user might not be able to input some single characters with their newly created keyboard, because the system simply would not find them in the layout files. During the runtime, the user then is able to switch between available layouts.

3.3.3 Options

To let the user use the other functions, we implemented an options button for our keyboard. It is marked with a black gear as seen in fig 3.3. The user can click on it with the trigger button of the controller when being in the hitbox of it. Then, three new buttons appear, each for one currently available option. One is to add new words, one to change the currently selected layout and the last one to scale the keyboard.

3.3.3.1 Add Words

The whole system works with a lexicon full of words and only these can be written with gestures. There will be words the user wants to write, that are not yet in said lexicon, hence they can not be written with gestures. For this case, we implemented a function such that the user can add new words. They can access it via the options button. Then they have to press the button where it says “add word”. When this button is pressed another button appears, the “add to lexicon” button. Now, the user has to input their intended word with

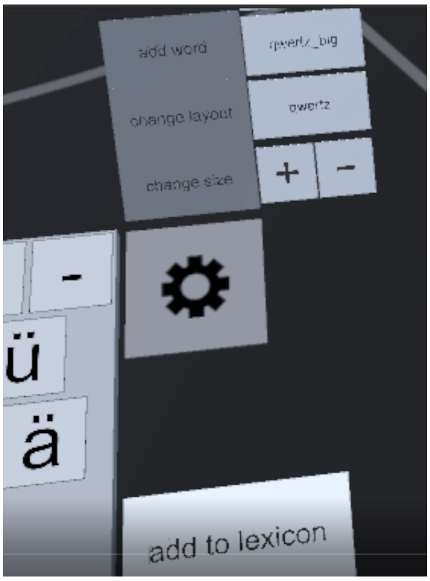


Figure 3.3: The options button with all the sub-buttons activated

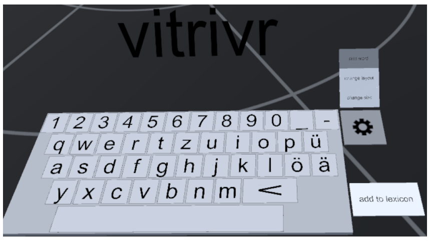


Figure 3.4: “vitrivr” written with single character inputs, can now be added to the lexicon

single characters. It will be displayed as seen in fig 3.4. Then, when they press the “add to lexicon” button, this displayed word will be added to the lexicon text file, if it does not already exist, and does not have a space in it. Additionally, for every available layout, the newly added word will also be added to the corresponding layout file, such that the user can, right after adding the word, write it with a gesture. One additional thing we implemented is, that the word only gets added in the text files, if it can be written with the layout the file corresponds to. For example, if a user wants to add the word “öffentlich”, but they made a layout without the letter “ö”, this word could never be written with this layout, hence it would be unnecessary to have it in the corresponding layout file.

3.3.3.2 Change Scale and Layout

There are two other functions that can be found under the options button. One is to change the size of the keyboard. When the user clicks on the “change scale” button, a “+” and a “-” button appear. By pressing the “+” button, the keyboard gets bigger, by pressing the “-” button, the keyboard gets smaller. This can help the user to make the keyboard more handy for them. Depending on how they want to move their arm, they can make it slightly bigger or smaller. If it is bigger, the gestures can be made more precise, if it is smaller,

gestures can be made faster.
The other function is the ability to change the layout. The user gets a list of all available layouts, when pressing the “change layout” button. They can then click on one of these layouts and the keyboard will change its appearance. The available layouts consist of the predefined ones and also of the newly added ones from the user.

3.3.4 Other

To end the implementation part, we will shortly talk about the last function, the ability to grab and move the keyboard. This means, the user can grab the keyboard, by being in its hitbox and pressing and holding the controller’s grip button. They can move it around in the room and rotate it as they want. If they release the grip button, the keyboard gets static and stays in the position. We were able to do this thanks to a script that already existed in vitrivr-VR TODO: HOW TO MAKE A REFERENCE TO THIS? and did not have to implement anything on our own.

4 Evaluation

In this section we want to talk about the evaluation as a whole. We want to look at the phrases we took, how we carried it out, the results observed and shortly discuss what all of this means.

4.1 MacKenzie Phrase Set

One precondition for the evaluation is to use the MacKenzie Phrase Set[5]. Basically, this is just a set of 500 phrases. According to the paper [3], such a phrase set should use phrases of moderate length, that are easy to remember and representative for the target language. These phrases do not contain any punctuation. Some of them use uppercase characters, but the authors mention, that participants can also be instructed to ignore the case of the characters.

Some statistics for the whole phrase set, also found in the original paper [3]: The MacKenzie phrase set consists of 500 phrases, that have a minimum length of 16, a maximum length of 43 and an average length of 28.61 characters. On the whole, 2712 words were used, which consist of 1163 unique words. A phrase consists of a minimum of 1, a maximum of 13 and on average of 4.46 words.

4.2 Task of the Participants

The task of the participants is to copy 15 ”random” phrases. They are not really random, but adjacent phrases from the downloadable MacKenzie Phrase Set (http://www.yorku.ca/mack/PhraseSets.ziphttp://www.yorku.ca/mack/PhraseSets.zip). As they are not in a specific order, e.g. alphabetic order, we decide to do it like this.

TODO: PICTURE OF EVALUATION SCENE. The participants see two text fields. On the top is the phrase to copy, on the bottom the words/phrase they write. If the given phrase matches the user inputted phrase, a sound sounds, such that the participants know when they finish one specific phrase. After that, a new phrase appears until 15 phrases

[5] http://www.yorku.ca/mack/PhraseSets.zip

are correctly inputted. If an incorrect word is entered, the user either can use the word suggestions (fig 3.2) or delete the wrong word and try to write it again. If a mistake is only noticed later on, the participants have to remove all words and characters up to and including the wrong word by using the backspace button.

After this first step, in the second step, we shortly explain two functions of the keyboard, which they can test afterwards. First the scaling buttons and then the function to add a new word. This is important, because we want to know if they find these functions useful and well implemented.

The last step of the evaluation is to fill a questionnaire. First it has some general questions about the participant's experience in VR. Then there is a block of questions in the form of a system usability scale. Per question, there are five possibilities to set the cross. From 1 (strongly disagree) to 5 (strongly agree). The questions are structured in such a way, that if the user is highly satisfied with everything, they would alternately make a cross at the 5 and 1. TODO: FRAGEBOGEN ALS ANHANG BEIFÜGEN.

4.3 Carry-out

To carry out the evaluation, we used two different VR systems. One was a setup with a HTC vive and HTC vive controllers. The other one included an Oculus Rift headset with corresponding controllers. Even though these are two different systems, it does not change much for the participants. In fact, only the controllers and their buttons differ a bit.

To find participants, we wrote an email to students from our university, and asked family members and friends. All in all, eleven people got in touch with us and participated at our evaluation. Every participant got the same explanation to give everybody the same foundation of knowledge.

We told them that if they are close enough to the keyboard, then the color gets a bit brighter, and they are in the keyboard's hitbox. We said, the keyboard is movable, if they press and hold the controller's grip button in the hitbox of the keyboard. If they release it, the keyboard gets static again and stays where it got put.

To write, they do also have to be in the hitbox of the keyboard but not pressing and holding the grip button, but the trigger button. Then they had to make a gesture over the characters of the keyboard to write a word. We also told them, that if they do a full gesture and a word longer than one character is written, a space is automatically put behind the word. We also said to them, that single characters could be inputted by clicking on a key of the intended character. If they did so, no space is put, and they have to do it their own. In the English language, this is particularly important for the words ”I” and ”a”.

We also told them, that if they made a gesture and a word was written, there may be one to four other choosable words. They could pick from them, if the word written in the text field is the wrong one. We especially mentioned the word “the”. All the time “thee” would be written as the best match, therefore they would have to correct it every time.

They were also informed about the backspace. So, that if they use the backspace button after writing a word, the whole word gets deleted and afterwards only single characters get deleted.

We also told the participants, that we have enough time and that they should not hurry, but rather look, that the inputted words are correct. Because if they are not correct, they have to use the backspace a lot of times.

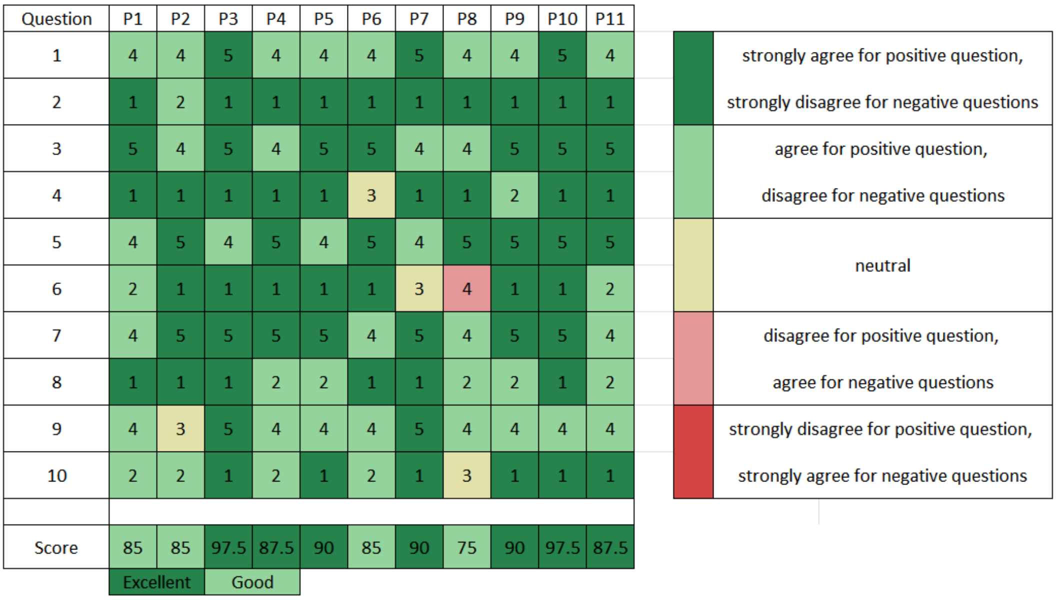
4.4 Results

Now, we want to talk about the results and some statistics we gained through the evaluation.

4.4.1 System Usability Scale

First, we begin with the results of the SUS questions. These ten questions were:

1. I think that I would like to use this system frequently when I work in VR
2. I found the system unnecessarily complex
3. I thought the system was easy to use
4. I think that I need the support of a technical person to be able to use this system
5. I found the various functions in this system were well integrated
6. I thought there was too much inconsistency in this system
7. I would imagine that most people would learn to use this system very quickly
8. I found the system very cumbersome to use
9. I felt very confident using the system
10. I needed to learn a lot of things before I could get going with this system



Question	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11
1	4	4	5	4	4	4	5	4	4	5	4
2	1	2	1	1	1	1	1	1	1	1	1
3	5	4	5	4	5	5	4	4	5	5	5
4	1	1	1	1	1	3	1	1	2	1	1
5	4	5	4	5	4	5	4	5	5	5	5
6	2	1	1	1	1	1	3	4	1	1	2
7	4	5	5	5	5	4	5	4	5	5	4
8	1	1	1	2	2	1	1	2	2	1	2
9	4	3	5	4	4	4	5	4	4	4	4
10	2	2	1	2	1	2	1	3	1	1	1
Score	85	85	97.5	87.5	90	85	90	75	90	97.5	87.5
	Excellent		Good								

🟩	strongly agree for positive question, strongly disagree for negative questions
▩	agree for positive question, disagree for negative questions
🟨	neutral
🟧	disagree for positive question, agree for negative questions
🟥	strongly disagree for positive question, strongly agree for negative questions

Figure 4.1: System Usability Scale with all the given points per question from every participant

Question 9 got the worst score, but with it being a 4.09 out of 5, it is still pretty good. From these values, we can calculate a usability score. Every question with an even number

is a negative one. That means, a score of one or “strongly disagree” is the highest possible. For the other questions, a score of five or “strongly agree” is the best possible score. So, from the odd numbered questions we have to subtract 1 from the average score. And for the even numbered questions we have to subtract their average score from 5. At the end, we have to sum up these ten newly calculated values and multiply them by 2.5. Our calculated usability score is 88.18. This is a high score, because from a score of 85.5 points, one talks about an excellent system usability. Therefore, we are really satisfied with the results of this.
We do also have two other questions about the scale and the “add word” function:
11. The function to add words is well implemented and easy to use
12. The function to scale the keyboard is unnecessary
Question 11 got a score of 4.55 out of 5 and question 12 got a score of 2.18, whereby 1 would be ideal. We conclude from these two questions, that the “add word” function makes a good impression whereas the scale function does not perform so well.

4.4.2 Writing Speed

One important thing of our evaluation is to find out, how fast users can write with our word-gesture keyboard. As unit to measure this values, we take the “words per minute” wpm. We calculate the wpm with following formula:

$$WPM = \frac{\mid T \mid}{S} \times 60 \times \frac{1}{5} \tag{4.1}$$

where T is all the phrases a participant had to write, hence $\mid T \mid$ is the number of characters a participant had to write. S is the time in seconds they used to write all 15 phrases.

participant	average WPM	lowest WPM	highest WPM
1	11.457	-	-
2	12.19	-	-
3	13.055	-	-
4	11.609	5.3	25.5
5	12.578	6.83	21.65
6	10.285	5.27	19
7	12.423	6.1	24.41
8	16.056	8.28	30.74
9	13.363	7.96	24.15
10	17.118	7.98	24.45
11	10.067	4.71	14.55
average	12.75	6.55	23.06

Table 4.1: average wpm, lowest wpm and highest wpm per participant. For the first three, we failed to get this data.

In Table 4.1 you can see how fast in average the participants were able to write their 15 phrases. We do also list the lowest and highest value. Everything is measured in words per minute.

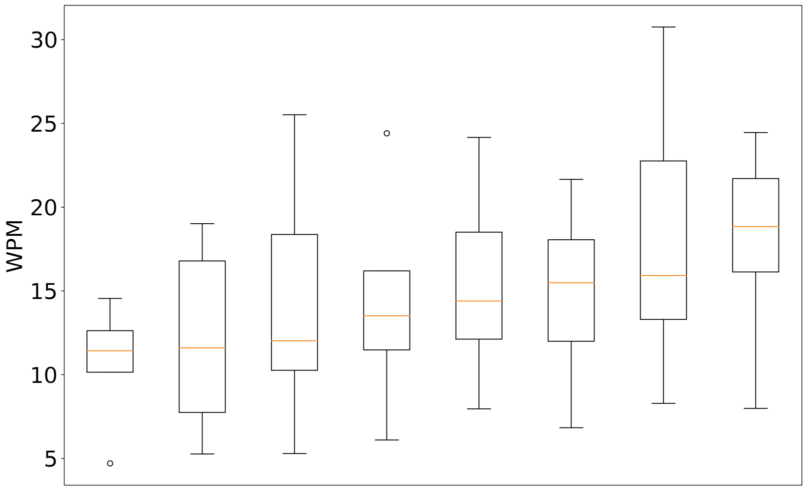


Figure 4.2: for participants 4-11: lowest wpm, 25% quantile, average wpm, 75% quantile and highest wpm

To understand the values of Table 4.1 a bit better, we make a so-called boxplot for every participant, for whom we have the data. We can see in fig 4.2, the higher the participant's median, most of the time they do also have higher lowest wpm value. The lowest wpm values mostly come about because a participant made a mistake and had to delete a lot and basically write the phrase two times. On the other hand, the highest WPM values come about because a participant made no mistake in writing the phrase. The rectangle in the middle of the two bars shows how consistent or inconsistent a participant's writing speed is. The lower bound is the 25% quantile, the upper bound the 75% quantile. This means, if the rectangle is less high, the writing speed is more consistent.
We cannot really find anything that combines writing speed and consistency.

Next, we want to look at the wpm values of the users and their experience in, on one hand VR writing and on the other hand word-gesture keyboards.

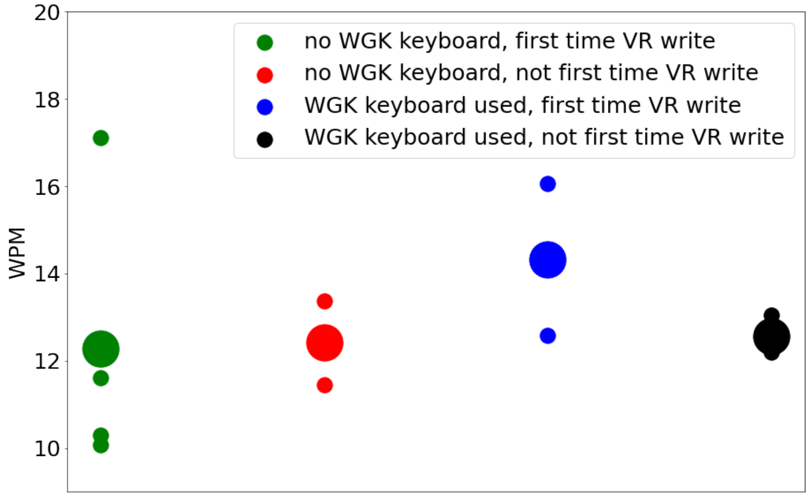


Figure 4.3: Average WPM per group of participant, grouped by their experience in VR writing and word-gesture keyboards

In Fig 4.3 we can see, that the prior knowledge, that some participants have, did not really help them to write faster. In fact, the fastest group was the one, that is experienced with word-gesture keyboards, but not with writing in VR. But we think this is due to the small sizes of the different groups.

Another thing we wanted to analyze is the WPM value compared to the age:

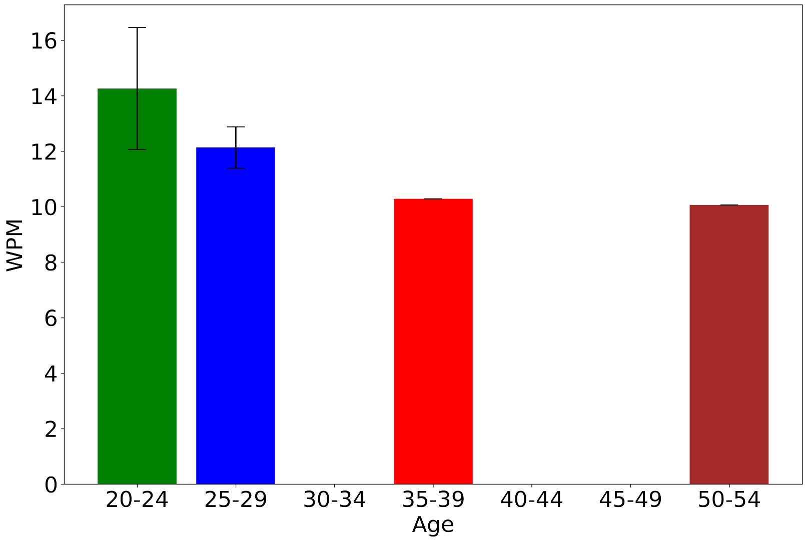


Figure 4.4: wpm values sorted by age groups of five years. The 20-24 and 25-29 groups also have a standard deviation error bar, because they contain of five and four participants. The 35-39 and 50-54 groups do not have an error bar, since each of them only contains of one participant

We can observe in Fig 4.4, that at least among our participants, the wpm value decreases with the increase in age. This means, the older participants were a bit slower than the younger ones.

4.4.3 Error Rate

For the measurement of the error rate, we calculate a user error rate and a system error rate. For the user error rate, we look at the amount of backspaces a participant had to use and the amount of words/character that were written wrong in all 15 phrases. For the system error rate we look at the words the system did not recognize at all, which means not as best match, nor as one of the four choosable suggestions and count their number of characters.

4.4.3.1 Most Frequent Error Words

We looked into all the words that were not the best match, so all the words a participant either corrected or not and also the ones that were not even in the suggestions. The top ten such words are:

Table 4.2: most frequent error words

word	times wrong
the	53
is	17
to	14
of	12
in	9
for	8
do	5
all	3
more	3
see	2

4.4.3.2 User and Backspace Error Rate

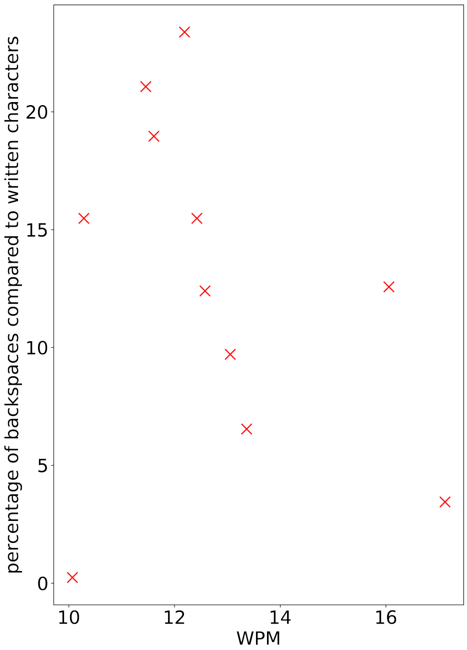
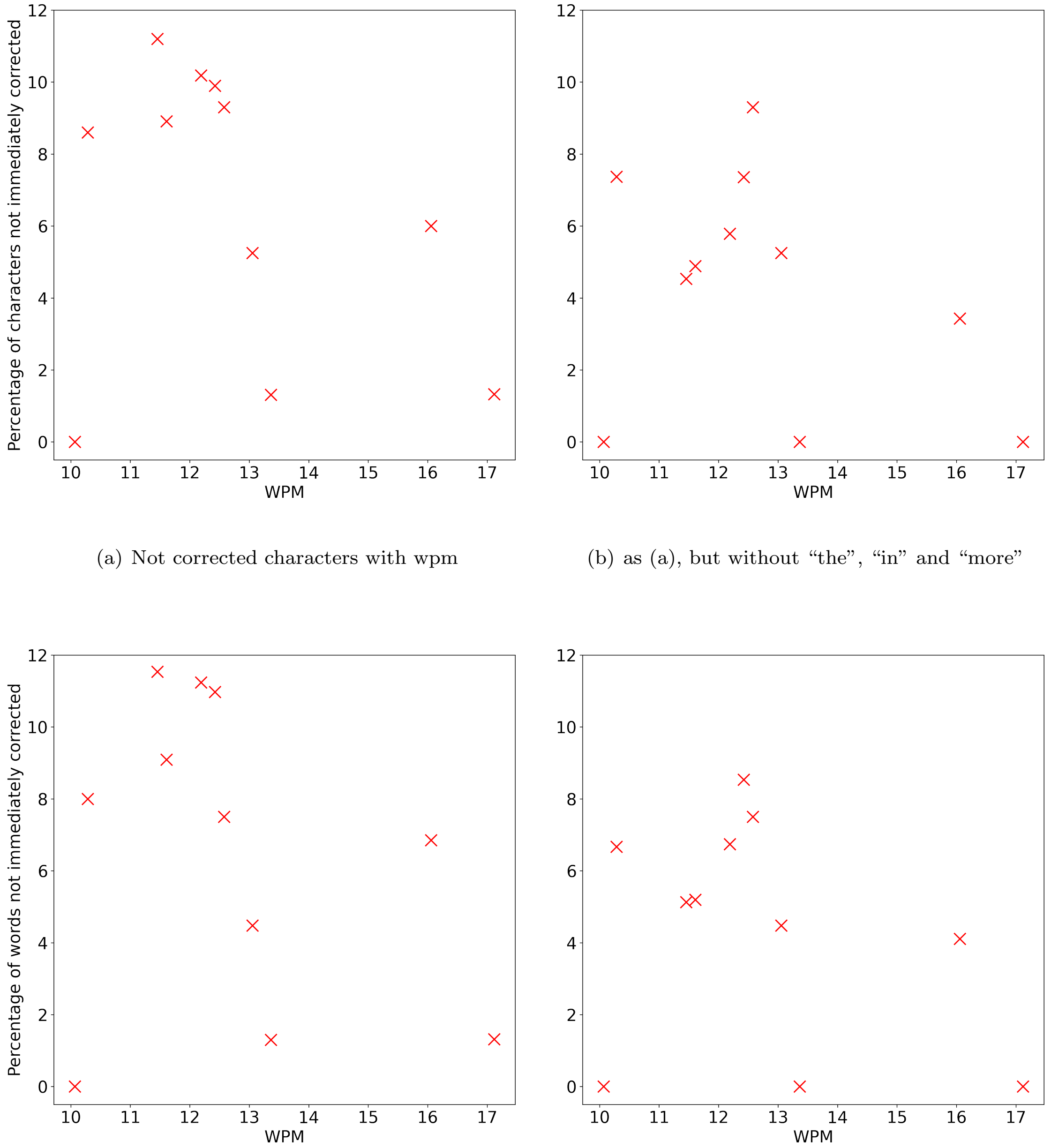


Figure 4.5: percentage of backspaces in relation to all characters in phrases compared to the wpm

In Fig 4.5 we show how many backspaces were used at which wpm value. We can not tell really much about it. But one thing that can be seen is, that the participant with the lowest wpm seemed to be very careful not overlooking wrong words that could be corrected by choosing the right suggestion. The participant with the highest wpm has the second-lowest usage of backspaces. Therefore, they seem to get used to the system and its suggestions words very fast and good.

Overall, there seems to be a little trend, that participants with higher wpm values had to use fewer backspaces than participants with lower wpm values.



(a) Not corrected characters with wpm

(b) as (a), but without “the”, “in” and “more”

(c) Not corrected words with wpm

(d) as (c), but without “the”, “in” and “more”

Figure 4.6: Plots of four Turing machines

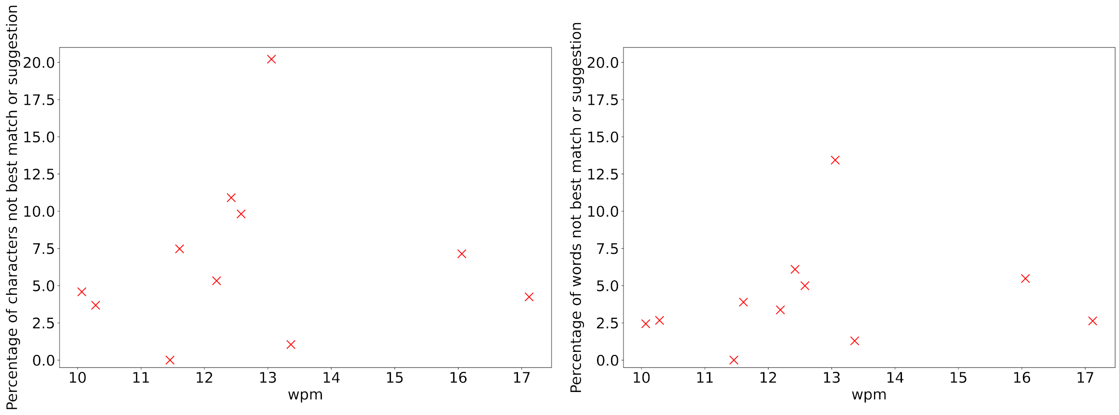
In Fig Fig. 4.6(a) and Fig Fig. 4.6(b) we count the characters of the words, a participant did not correct, although they had the chance to do it with the suggestions. On the y-axis, we look at the percentage, that these characters make up in comparison to all characters the participant had to write in all 15 phrases. In Fig Fig. 4.6(a) we look at all characters. But a lot of the errors came from the words “the”, “in” and “more”, that could have been prevented as mentioned in Section TODO: MENTION TO SECTION THAT HAS TO BE WRITTEN. Therefore, in Fig. 4.6(b) we look at the errors without consider these three words.

In the figures Fig. 4.6(c) and Fig. 4.6(d) we do the same, but without counting the characters of the “error words” but just look at the number of them.

One thing that can be observed is, that in the two lower figures of Fig Figure 4.6 the percentage values are almost the same as in the two upper ones. This means that the words that were not corrected, have on average about the same length as the average length of all words from the respective 15 phrases is.

One thing that can be observed at the left figures of Fig Figure 4.6 is that apart from one exception the percentage rate of not corrected words/characters is higher at lower wpm than it is at higher wpm. One obvious reason for this could be, that if words early in phrases were not corrected, a user had to delete the whole phrase back to the wrong word, and had to write all again. Therefore, it would make sense, that the faster participants made fewer errors, hence their risk of not correcting a word in the beginning of a phrase is lower. And then, they would not decrease their average wpm so much.

4.4.3.3 System Error Rate



(a) with characters

(b) with words

Figure 4.7: percentage of characters/words that were not found by the system neither as best match nor as suggestion

In Figure 4.7 we can see what percentage of the characters/words were not found by the system, neither directly as best match, nor as word suggestions, which a participant could choose. We decide to name this error the system error, although a participant could have done a really bad gesture which then would not really be the system’s fault to not find a word.

We can see, that in Fig. 4.7(a) the percentages are much higher than in Fig. 4.7(b). This means that the words, which the system did not find a word, need to be longer than an average word in the written phrases. A reason for this might be, that often gestures for long words did not succeed. If a participant made some little curves too much in a gesture for a long word, the chances are lower, that the system can detect the right word, whereas for a small word the chance is higher, that the word gets found. Because the gesture can be much shorter and therefore fewer inaccuracies from a participant will happen.
For the wpm and the percentages we can not detect any correlation. It seems, that these two values are not connected with each other.

4.4.4 Feedback

The full list including all verbatim feedback from every participant can be found in the appendix. Here, we just want to highlight the most frequently addressed points.
Some participants find the best matches and suggestions sometimes confusing. The most mentioned example is that “thee” is preferred over “the”. The spaces/ current position are another thing that is frequently addressed. For some participants, it was unclear where they were to write. They suggest to implement some kind of visual indication, that indicates, where the cursor currently is. We also got a lot of praise and most of the time we saw a cheerful face when the VR headset got taken off.

4.5 Discussion

In this section we want to compare our results to other works' results and talk about some changes we did because of the feedback and results.

4.5.1 Result Comparison

First, we begin with the wpm. Boletsis and Kongsvik [1] evaluate in their paper four different VR input methods, a raycasting, drum-like, head-directed input and a split keyboard. The first one is a keyboard where a user can select letter by pointing a ray with a controller on it. For the second one the controllers simulate drum sticks in VR and letters have to be pressed by them. For the third keyboard a user has to aim with the head for the letters and press a button on the controller to input. The last keyboard is one, that is split into two halves, one assigned to each controller.

Table 4.3: most frequent error words

text input method	wpm
Raycasting keyboard	16.65
Drum-like keyboard	21.01
Head-directed input keyboard	10.83
Split keyboard	10.17
Word-gesture keyboard	12.75

In Table 4.3 we listed the results of Boletsis and Kongsvik [1] and our measurement of the

wpm value. They had a similar approach to the evaluation as we did, with one difference. They did use the same ten phrases for every participant and keyboard type.
As we can see, our keyboard lines up in the middle. It is not the one with the lowest wpm, but also not the one with the highest.

4.5.2 New Implementations

5 Conclusion

This is the body of the thesis.

5.1 Results Discussion

5.2 Future Work

Bibliography

[1] Costas Boletsis and Stian Kongsvik. Controller-based text-input techniques for virtual reality: An empirical comparison. *Int. J. Virtual Real.*, 19:2–15, 2019.

[2] Per Ola Kristensson and Shumin Zhai. Shark2: a large vocabulary shorthand writing system for pen-based computers. In *UIST '04*, pages 43—52, 2004.

[3] I. Scott MacKenzie and R. William Soukoreff. Phrase sets for evaluating text entry techniques. In *CHI '03 Extended Abstracts on Human Factors in Computing Systems*, pages 754—-755, 2003.

[4] I. Scott MacKenzie and Shawn X. Zhang. The design and evaluation of a high-performance soft keyboard. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 25—-31, 1999.

A

Appendix

University of Basel

Faculty of Science

Declaration on Scientific Integrity

(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis:

Name Assesor: ____

Name Student: ____

Matriculation No.: ____

With my signature I declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Place, Date: ____________________ Student: ________________________

Will this work be published?

□ No

□ Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: ____

Place, Date: ____________________ Student: ________________________

Place, Date: ____________________ Assessor: ________________________

Please enclose a completed and signed copy of this declaration in your Bachelor’s or Master’s thesis .

August 2021