Word-Gesture Typing in Virtual Realty

Bachelor Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Databases and Information Systems Group
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

I would like to thank Prof. Dr. Heiko Schuldt for the opportunity to write this thesis in the Databases and Information Systems Groups. Further, I would especially like to thank my supervisor Florian Spiess for giving me great advice, helpful feedback and clear answers to the many questions I had. Finally, I would also like to thank my family, friends and fellow students for helping me with the evaluation and giving me motivation to give my best for the thesis.

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 [6] 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 [4] 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” [4] 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. TODO:

DEFINE USER INPUTTED GRAPH 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. 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 is also a channel that brings language information into the calculation. Each channel alone from the system developed by Zhai and Kristensson [4] does not necessarily have enough power, but all the channels together can detect the right word. 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. Such a 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.

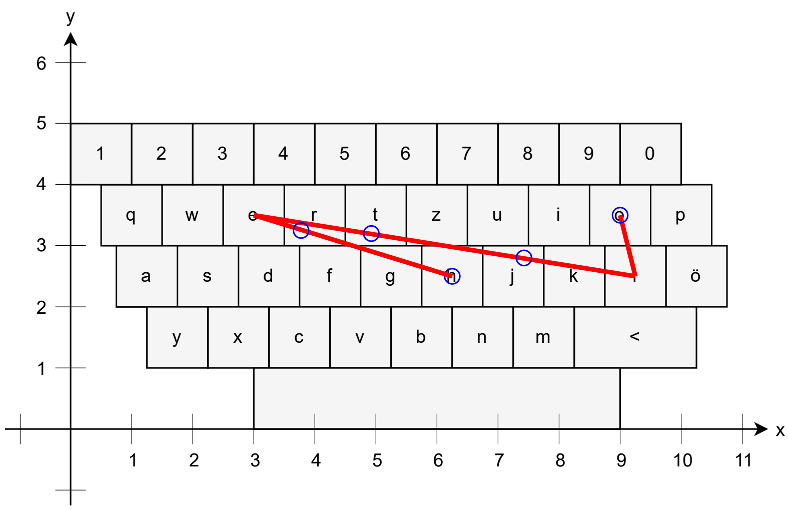


Figure 2.1: TODO: CAPTION

For all the words in the lexicon, also their perfect graphs have to be stored. Zhai and Kristensson [4] do not mention how they store these, but it would make sense, to only store N points for each graph, with N being the number of points that samples the graph. This can be justified by the fact, that for later calculations with the graphs, only these N points are needed and not more.

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 in shape and location. If one of these two distances is bigger than a given threshold, the checked word will be discarded and not further considered.

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.

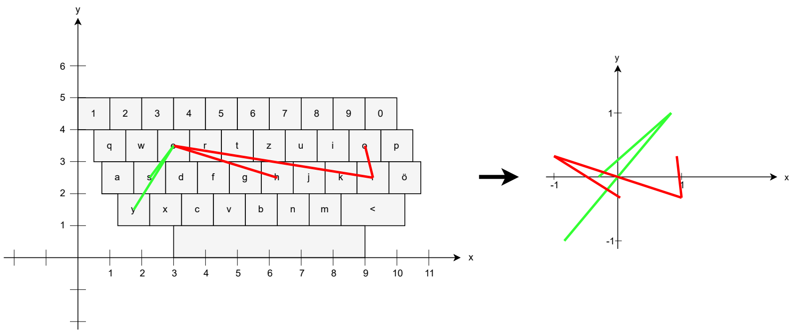


Figure 2.2: TODO: CAPTION

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 [4].

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 the 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 1'117 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.

2.3.1.4 Location Channel Recognition

To avoid a lot of these confusion pairs, the authors were using a second channel. This one is not for the shape, but for the location of the graphs. For the following formulas and calculations, the normalization of the graphs is 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.

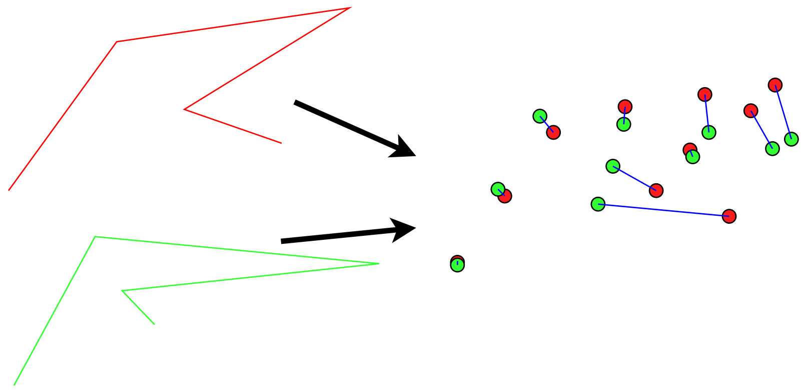


Figure 2.3: TODO: CAPTION

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 and $\alpha(i)$ with $i \in (1, N)$ are weights for the different point-to-point distances, such that $\sum_{i=1}^{N} \alpha(i) = 1$. These weights can be obtained through 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. Another way would be through training with a large amount of data. Lastly, $\delta(i)$ is defined through following formula:

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

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 on the keyboard 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 is the number of points used to sample a graph. 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 u is set to the sum of the spatial point-to-point distances. This means, N distances have to be calculated and summed up. These are the distances between the i-th point of u and the i-th point of t for $i \in \{1, N\}$

2.3.1.5 Channel Integration

With the two distances x_S and x_L, the most probable word, the user intended to write, can be calculated pretty well. 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 w, the probability, that w 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. They set $\mu = 0$ and σ can be obtained through training from large amount of data. Here, σ can be seen 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:

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

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

2.3.1.6 Further Steps To Improve The Results

There are also some further steps that can be made, but we did not implement in our algorithm. In this section, we want to discuss shortly about these.

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 use Fitts' law to calculate the normative writing time for a perfect graph. TODO IF NOT ALREADY: THEN COMPARE
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, which 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 a simpplified version of the word detecting algorithm used in SHARK^2.

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 as explained in section (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 inputted 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 (2.3.1.6). 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 normalize the graph's longer bounding box side to a fixed length. Therefore, a small graph gets stretched, 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.

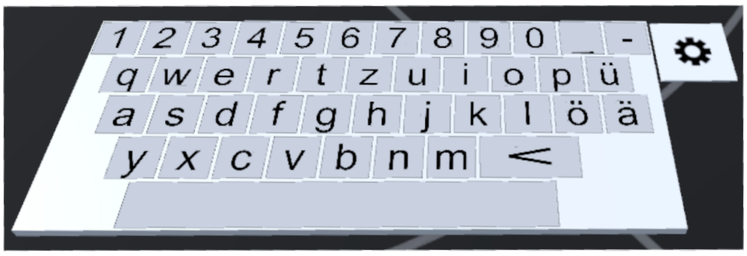


Figure 3.1: Word-gesture keyboard in its “normal” form without anything selected

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.

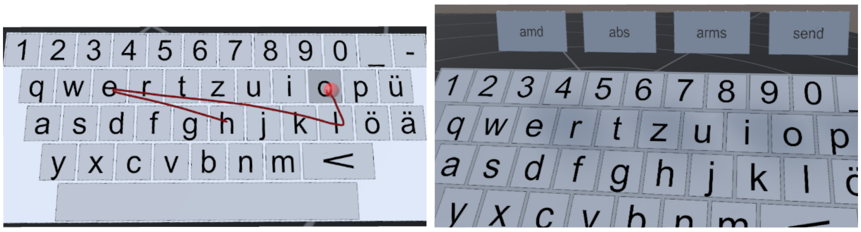


Figure 3.2: left: the user writes the word “hello” as gesture, right: user gets recommended words for “and”

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.

There are some challenges in VR we have to master. First of all, in VR the user is in a

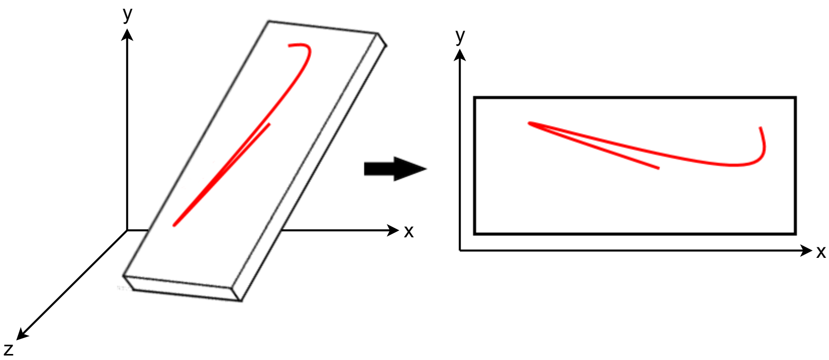


Figure 3.3: TODO: CAPTION

three-dimensional space. Thus, when they make a gesture, they could move up and down with their controller and do not have to stay on the keyboard. For this we need to look, that the line of the gestures still lies on the keyboard as if the user would draw on it as if it was paper, and they are using a pen. The next problem is, that a user can rotate the keyboard as they want in VR. Therefore, the drawn graph will most of the time not be lieing perfectly on the x-z-plane. This means, we have to transform all the points into the x-z-plane to be able to further process these gestures. Another problem of the inputting of gestures is the amount of points. Technically seen, every frame the user presses draws a gesture, a new point gets saved. Thus, we have to reduce this amount, because it would be just too many points, and it would slow down the program massively. We implemented a function that only TODO: ÜBERPRÜFEN. keeps points of they have either a certain distance between itself and the previous point, or there is a big enough angle.

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 technically seen 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 avoid 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 are spaces. The proper way to do this would be, if a word of more than one character was written with a gesture, for the next input, either if it is a single character or a whole word, a space is put before it. Some exceptions where the space would be omitted is for example if after a word a ".", "," or "?" is written. Also, at the start of a sentence, it should not set a space. If a single character was written, for the next word it should set a space before it, but if again a single character follows, it should not put one.
In our implementation, when the user writes a word that consists of more than one character, a space gets put behind the word. This means, if they write a second word or a single character after, they do not have to put a space manually. But if they write a single character, no space is put automatically after it. We do it this way, because the keyboard does not know what was already inputted in a textfield. For example, if the user changes the textfield but wrote into another a word before, the keyboard stores, that there is a word written and would put a space before the first word, even though this new textfield was

empty before.
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. While this would not be a necessary function to reach the core goal of inputting text, it can still help users to feel more confident using the keyboard. For example, if we only implement the qwertz layout, a user that is only using keyboards with the qwerty, ATOMIK or any other layout than qwerty, it can be cumbersome for them to use this keyboard. The problem is, that we do not know every layout all users are using for non-VR text input. Therefore, it is good that a user can decide by themselves what layout they like most to use and simply create it for our word-gesture keyboard.
TO achieve the creation of new 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 3.4. 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

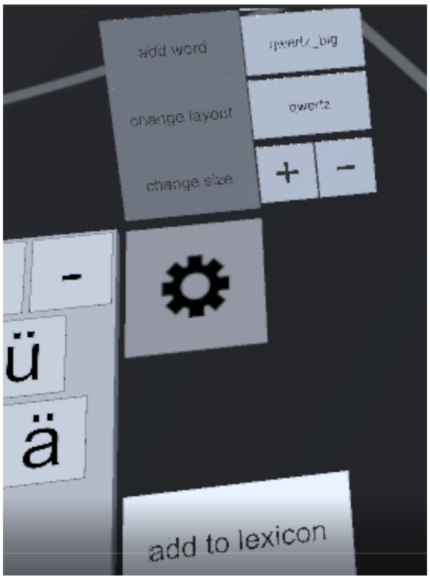


Figure 3.4: The options button with all the sub-buttons activated

Figure 3.5: “vitrivr” written with single character inputs, can now be added to the lexicon

single characters. It will be displayed as seen in Figure 3.5. 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 Moving and Rotating

Finally, we will briefly 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.
This function can become handy, if the user wants to reposition themselves in the VR space. They can just take the keyboard with them. Also, they can choose to have it closer or futher away, which can lower the cumbersomeness to use the keyboard. Rotating the keyboard can also help this purpose. Some users might prefer to have it more tilted than other do.

4 Evaluation

In this section we talk about the evaluation as a whole. We take a look at the phrase set we used, how we carried it out, the results that can be observed and compare it to existing results.

4.1 MacKenzie Phrase Set

For the evaluation we used the MacKenzie Phrase Set[5]. This is a set of 500 phrases. According to MacKenzie and Soukoreff [5], such a phrase set should use phrases of moderate length, that are easy to remember and representative for the target language.
The phrases of the MacKenzie Phrase Set 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 [5]: 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 phrases from the MacKenzie Phrase Set. As they are not in a specific order, e.g. alphabetic order, we decide to take adjacent phrases and not random ones.
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 tone sounds, such that the participants know when they finish one specific phrase. After that, a new phrase appears until 15 phrases are correctly inputted. If an incorrect word is entered, the user either can use the word suggestions (fig

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

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 briefly 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 twelve questions. The first ten are from the system usability scale and the last two are for the previously mentioned functions they test. Per question, there are five possibilities to choose from. 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: questionnaire as appendix.

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 did not change much for the participants. In fact, only the controllers and their buttons differ a bit.
The participants consist of volunteers including computer science students, friends and family members. In total, 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.
Participants were provided the following information:
If they are close enough to the keyboard, then the color gets a bit brighter, and they are in the keyboard's hitbox. 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. While pressing they can make a gesture over the characters of the keyboard an release it to write a word. If they do a full gesture and a word longer than one character is written, a space is automatically put behind the word. Single characters can be written by clicking on a key of the intended character. Then, no space is put, and they have to do it on their own. In the English language, this is particularly important for the words "I" and "a".
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 not the intended 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.

They were told, that they 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.
In total for the eleven participants, 165 phrases were used. The average word length was 4.16 with the minimum length of 1 and a maximum length of 12. In total, 856 words were written, and it consisted of 460 unique words. We also made sure before the evaluation, that all the words in these phrases are in our word lexicon, such that the participants could write every word with a gesture.

4.4 Results

Now, we 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

In Figure 4.1 green fields mark positive responses to questions, red ones negative responses and yellow ones neutral feedback. All in all, we can see a lot of green fields, which means the feedback to the SUS questions is quite good.
Question 9 got the worst score, but with it being a 4.09 out of 5, it is still fairly good. From the average values for each question, 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 1 has to be subtracted from the average score. For the even numbered questions, the average score needs to be subtracted from 5. At the end, these ten newly calculated values have to be summed up and multiplied 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.
We do also have two additional 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 these 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.

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

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

In Table 4.1 you can see how fast on 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.

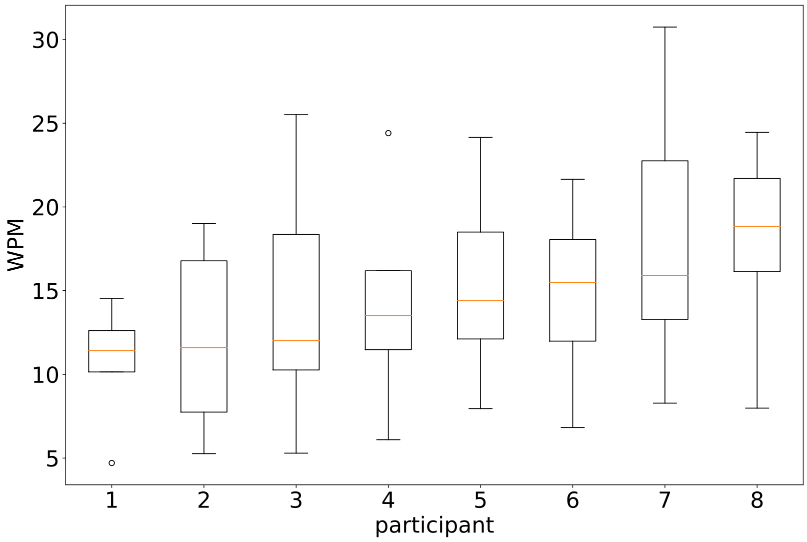


Figure 4.2: for participants 4-11: lowest wpm, 25% quantile, average wpm, 75% quantile and highest wpm

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, most of 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 shorter, the writing speed is more consistent.

We cannot find any clear trend that combines writing speed and consistency by looking at our measurements.

As mentioned above, if the participants only recognize an error at the beginning of the phrase when they almost finished it, they have to use a lot of backspaces and more or less have to write the phrase a second time. These can pull down the average wpm a lot. Therefore, we calculate the average wpm for the participants 4-11, where we only look at "perfect" phrases, that were written without using a backspace. First of all, the average wpm for these eight participants over all phrases is 12.94. The average wpm of these "perfect" phrases only is 16.53, which is about 28% higher.

Next, we want to find out, if the writing speed of the users has something to do with their experience in, on one hand VR writing and on the other hand word-gesture keyboards.

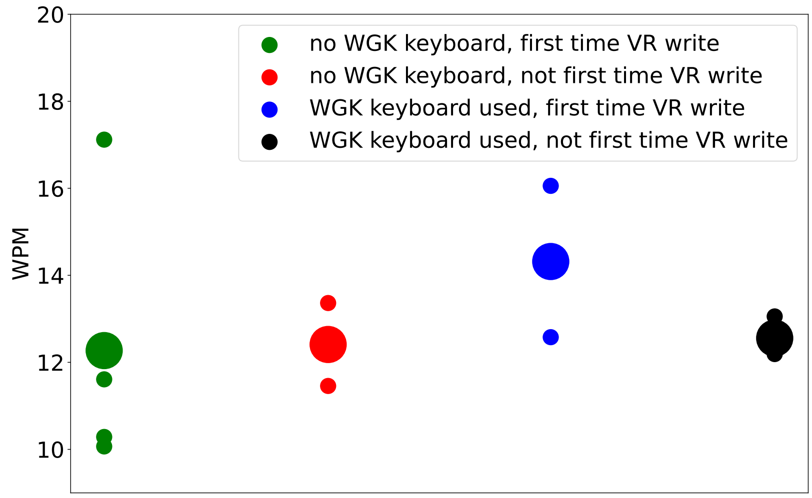


Figure 4.3: Each small dot shows the average wpm for a user. The big dots represent the average wpm per group. The colors are for the different groups of experience in VR writing and word-gesture keyboards

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 due to the small group sizes this is not really representative. Therefore, we can not conclude much about this, but it could have had a more interesting result.

Another thing we wanted to analyze is the writing speed compared to the age:

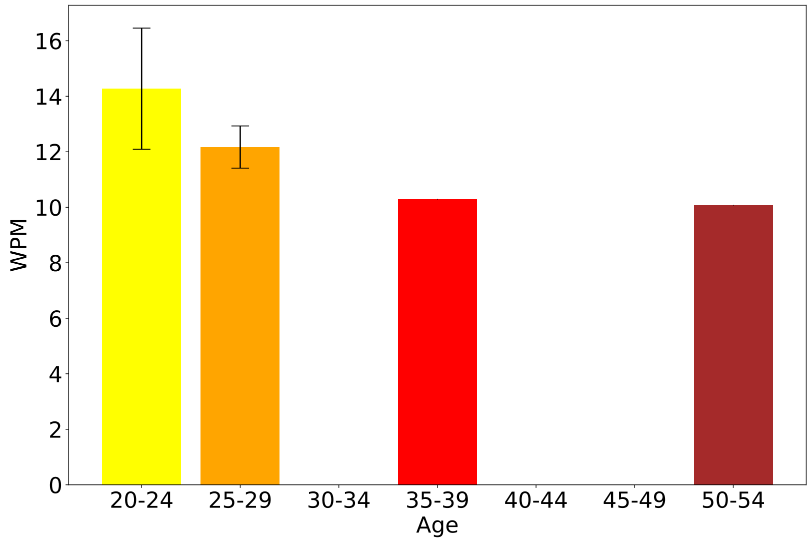


Figure 4.4: wpm values sorted by age groups of five years. The 20-24 and 25-29 groups also have a standard deviation error bar, because they contain of five and four participants. The 35-39 and 50-54 groups do not have an error bar, since each of them only contains of one participant

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

In this section, we will look at different error rates. First of all, we want to find out, which words caused the most errors. Then we investigate some errors caused by the participants. At the end of this section, we take a look at the errors mostly caused by the system's condition.

4.4.3.1 Most Frequent Error Words

We look 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 keyboard's word suggestions. The top ten such words are:

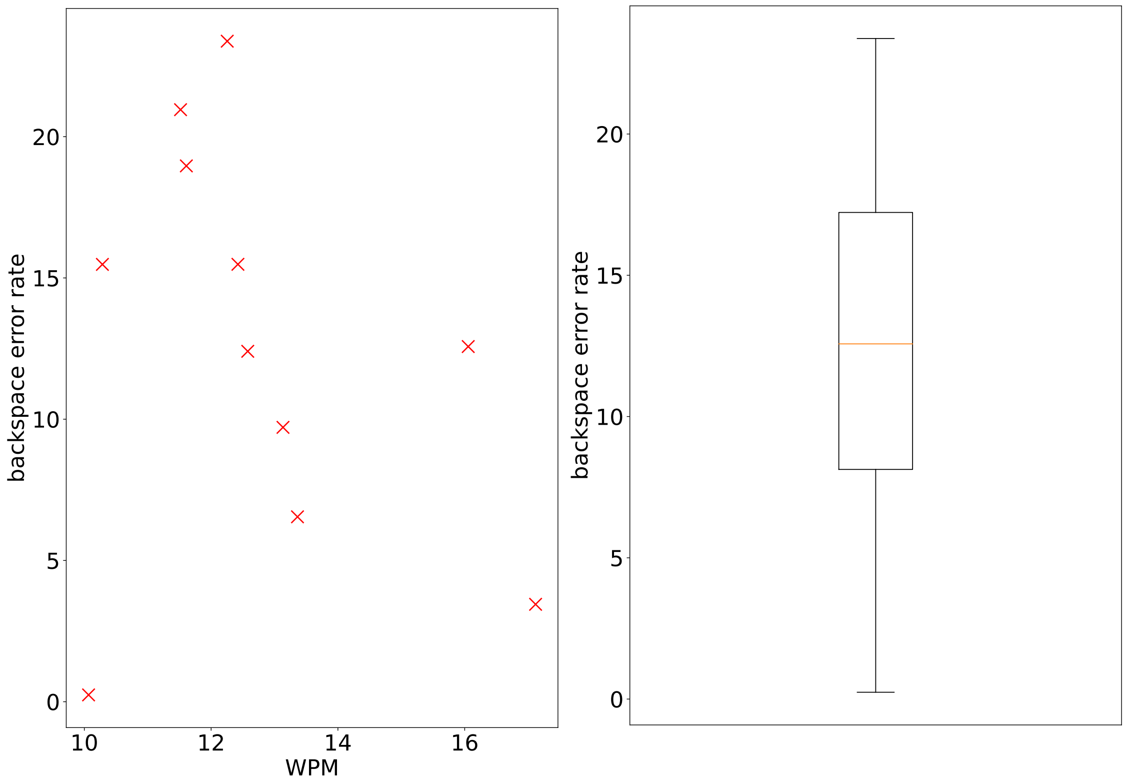
Table 4.2: most frequent error words

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

As we can see in Table 4.2, the word that is responsible for most of the errors is “the”, which caused 53 of them. In this list, we can also see other words whose errors can be avoided by a better implementation. For example “ìn” and “more”. The best match, when a participant wanted to write these words, were “thee”, “inn” and “moore”, which are words, that do not appear that much in plain language. For other words like “to” or “of”, whose best matches were “too” or “off”, it is maybe not as easy to get the right word. Because all of them appear fairly often in plain language.

4.4.3.2 Backspace Error Rate

The erroneous keystroke error rate (EKS ER) [1] measures the ratio of the total number of erroneous keystrokes to the length of the phrase that has to be written. For our calculations, we will take a similar formula but not exactly the one for the EKS ER. In another paper, Chen et al. [3] also used a variant of EKS ER. Instead of the erroneous keystrokes, they took the number of times the backspace key was used. We call this the “backspace error rate”.



(a) backspace error rate compared to wpm

(b) a boxplot for the backspace error rate over all participants. The orange line is for the median, the rectangle shows the 25% and 75% quartiles, and the upper and lower whiskers show the maximum and minimum.

Figure 4.5: shows the backspace error rate in two different ways. (a) with a comparison to the wpm of each participant and (b) a boxplot combining all participants' backspace error values.

In Figure 4.5(a) we show which wpm value was reached at which backspace error rate. We can not find much of a correlation. 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 keyboard suggestion. The participant with the highest wpm has the second-lowest percentual usage of backspaces. Therefore, they seem to get used to the system and its suggestion 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.
In Figure 4.5(a) we can see that the median value is around 12.5%. In fact, the average backspace error is 12.65%. This means, for every input of an amount of words with 100 correct characters about 13 backspaces were used.

4.4.3.3 Total ER

TODO: IF = CORRECTED WORDS, WRONG CHARS OF THEM The next thing we calculate is the Total ER according to Soukoreff and MacKenzie [7]. First of all, we have to clarify, that for this calculation we take another look attention the transcribed text than we did in the execution of the evaluation. In the Section 4.2 we mentioned that the phrase, a participant wrote, had to correspond exactly to the given phrase, otherwise it would not have gone to the next one. For the measure of the total ER value, we take the transcribed text as if a participant would not have been able to press the backspace. With only one exception, the backspace would have been allowed, if neither the best match nor the word suggestions had the right word and a wrong one is written. For the total ER value, we take following formula:

$$Total\ ER = \frac{INF + IF}{C + INF + IF} \times 100\% \tag{4.2}$$

Where C is the number of correct characters in the transcribed text, INF denotes the number of incorrect and not fixed characters in the transcribed text compared to the given phrases, identified with the minimun string distance and IF denotes the keystrokes in the input stream, that are not in the transcribed text and not editing keys.
(However, we will not take these values exactly as stated above as Soukoreff and MacKenzie [7] did. For C we take the number of correct characters of each 15 phrases a participant had to input. This means, we take the amount of characters that had to be written and subtract one for every character that is not in the previously defined transcribed text, but should be. One thing counting into this are missing spaces most of the time after words with one character as "I" or "a". Also, some characters from not immediately corrected words can count into it.
Next, for INF we do not count every character of an error word as error, but we calculate the minimum string distance (MSD) between the error word and the word it should be. The MSD value is the number of single character insertions, deletions or substitutions to get from the wrong word to the right one. For example, to get from "thee" to "the", we need one deletion, therefore the MSD value would be 1.
Finally, for IF we count the amount of backspaces a participant used to delete a wrong word, where the correct one was not even in the keyboard's word suggestion.) The following table shows our calculated results:

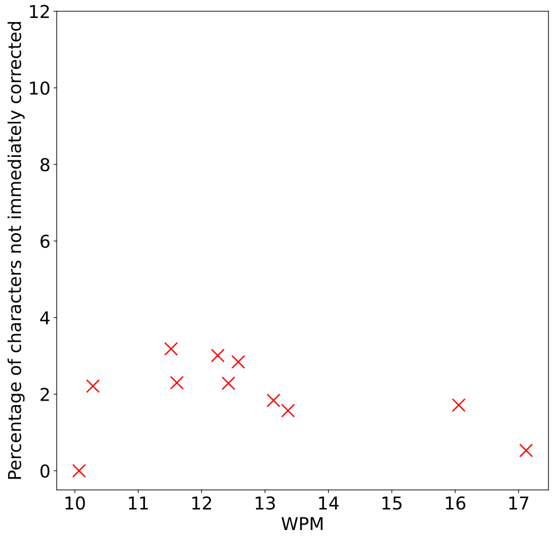
Table 4.3: Total ER per participant

participant	total ER
1	3.15%
2	2.95%
3	1.82%
4	2.54%
5	3.06%
6	2.18%
7	2.73%
8	2.25%
9	1.81%
10	1.05%
11	0%
average	2.14%

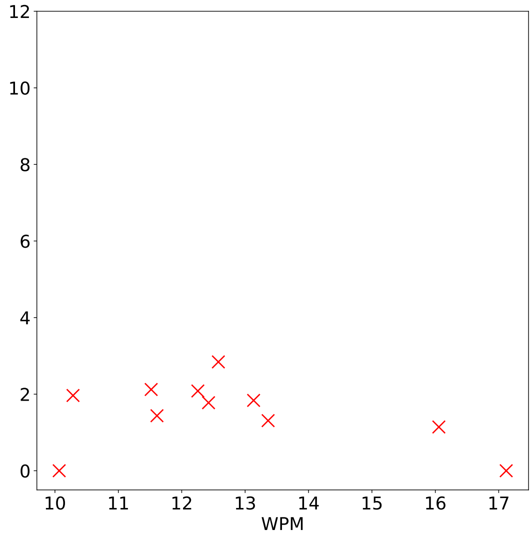
As we can see in Table 4.3, the calculated total ER values are quite low. The averge is only 2.14% with a standarddeviation of 0.91. Right now, we can not tell much about this, but later, we will compare these values with others from another paper.

4.4.3.4 Participant Conscientiousness

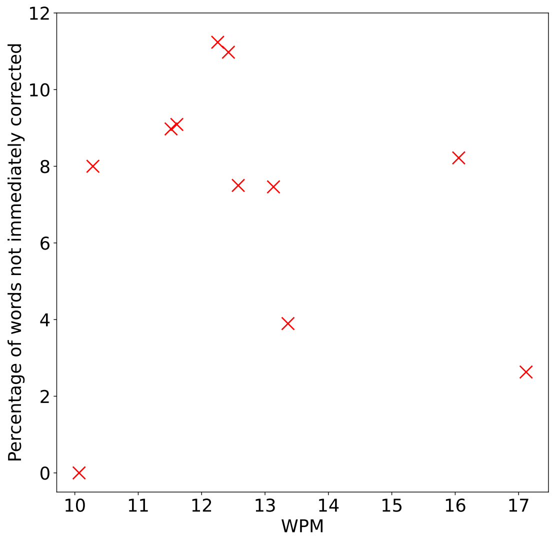
For the next calculation, we investigate the percentage of not immediately corrected words together with the wpm. Firstly, we will do this with characters and then with the whole words. For the calculation with the characters of an error word we take the MSD values as we did in the previous calculation with the total ER. Then we want to compare these two with the same approach, but without considering the errors happened because of “the”, “in” and “more”. We decide to remove these three words, because they could be avoided by a small change in the best match calculations.



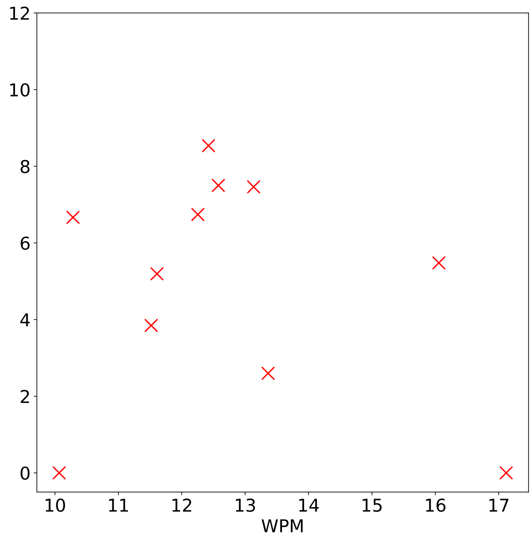
(a) Not corrected characters with wpm



(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: (a) shows the percentage of characters calculated with the MSD from wrong words, that were not immediately corrected with the word suggestions. (b) is the same as (a) but errors from “the”, “in” and “more” are ignored. (c) is the same as (a) but not with characters, but with the words as whole. (d) is the same as (b) but not with characters, but with the words as a whole

In Fig. 4.6(a) and Fig. 4.6(b) we calculate the MSD of the words, a participant did not correct, although they had the chance to do it with the keyboard’s 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. 4.6(a) we look at all wrong words. But a lot of the errors came from the words "the", "in" and "more", that could have been prevented as mentioned before. Therefore, in Fig. 4.6(b) we look at the errors without considering these three words.

We can see that the percentage of characters that are not corrected lies between about 0-3.5%. Overall we can observe that the percentages go a bit down from Fig. 4.6(a) to Fig. 4.6(b). Looking at the wpm, we can say that the writing speed does not really affect the percentage of characters not immediately corrected and is to some extent the same for every participant.

In Fig. 4.6(c) and Fig. 4.6(d) we do the same, but without the MSD of the wrong words but just look at the amount of them.

For these two figures, we can observe that the percentages also drop if we do not consider the error of the three previously mentioned words. Here it makes a bigger difference than in Fig. 4.6(a) and Fig. 4.6(b). Therefore, it seems that these three words do not contribute much to the total MSD value. Another thing we can see, is that the percentage of not corrected words lies between 0% and about 12% which is much higher than the percentages for the characters. This is because we work with the MSD. If a word is wrong, most of the time, our keyboard will not write a totally wrong word, if the gesture is right to some extent. It will consist of similar letters as the right word. Therefore, not the whole word's characters will be counted into this value.

To take another look at the participants' attention, we want to calculate the value of participant conscientiousness. As for the total ER, we also take the formula for the participant conscientiousness value from the paper of Soukoreff and MacKenzie [7]:

$$Participant\ Conscientiousness = \frac{IF}{INF + IF} \tag{4.3}$$

Normally, they would take the values of IF and INF as mentioned in Section 4.4.3.3 (not as we took them, but as Soukoreff and MacKenzie [7] took them). For us, the problem is that we emanate from the scenario, where a participant is not able to use backspaces apart from the special case. Therefore, there will not be any characters in the input stream that are not in the transcribed text besides the special case words and the according backspaces. To get a better value for the participant's conscientiousness we decide to work with words only, not characters. Thus, for INF we count all the words not immediately corrected by a participant and for IF the words that were corrected by using the keyboard's suggestions. We think, that the calculation makes more sense this way when looking at our keyboard, that writes most of the time whole words and not single characters.

A score of 100% would be perfect. This would mean, that a participant corrects every word with the word suggestions, if it is possible. The average value we get here is 66.28%, which means about two third of the words, that can be corrected with the suggestions, are corrected this way. The worst value is 33.33% and the best value a participant achieved is 100%. In our opinion, this score is not too bad. A lot of the participants are not used to word-gesture-keyboards and do not work in or use VR that much. We think, if they

used the keyboard over a longer period of time, they would get used to the words, that are sometimes harder for the keyboard to get as best match, and would pay more attention to use the word suggestions then.
Interesting is, that if we let away the errors caused by “the”, “in” and “more”, the average value decreases to 63.73%. This is an indication of the fact, that the participant seemed to get used to the errors caused by these three words and fixed them by using the word suggestions.

4.4.3.5 System Error Rate

Here we do not want to see if a participant was paying attention or not, but rather if our algorithm finds the right words most of the time. We count all the words that were neither the best match nor in the word suggestions, such that a participant had no chance to get this word right the first time. If this happened multiple times to a word, we count it in every time. We decide to name this error rate the system error rate, 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

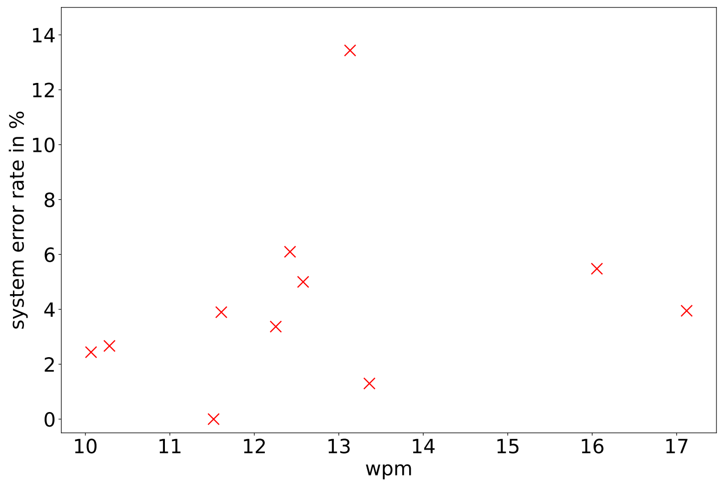


Figure 4.7: percentage of words that were not found by the system neither as best match nor as suggestion

In Figure 4.7 we can see what percentage of the words were not found by the system, neither directly as best match, nor as word suggestions, which a participant could choose from.
We can see, that the system error rate, does not have a correlation with the wpm. We can not see a tendency, that either the ones with high wpm or the ones with low wpm have fewer problems with not being able to write the right word. The average system error rate is 4.33%, which means that about every 23rd word made a problem.
On one hand, the average length of a word of the phrases we used is 4.16, on the other hand,

the average length of a word the system has not found in the first try is 7.03. This tells us, that words with a length way bigger than the average one, are more error-prone. We think this makes sense, because the longer the word, most of the time, the longer the gesture. In a bigger gesture, deviations from the perfect graph are harder to prevent than in a small gesture and the system might not output the right word.

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 visualization of spaces or the current position are another thing that is frequently addressed. Some participants find it unclear where the cursor is, hence if a space is missing or not. They suggest to implement some kind of visual indication, that indicates, where the cursor currently is. We can say about this, that this problem existed in the evaluation environment, but in vitrivr-VR, the program that our keyboard is manly developed for, has these kinds of cursor visualizations.
We also got a lot of praise and most of the time we saw a cheerful face when the VR headset got taken off. Some find it surprisingly intuitive, and others have a good feeling about using it.

4.5 Discussion

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

4.5.1 Result Comparison

First, we begin with the wpm. Boletsis and Kongsvik [2] 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 letters 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.
Another paper we want to compare the wpm with is from Chen et al. [3]. One of the keyboards they evaluate is a word-gesture keyboard where a user points with a ray from the controller to it and makes gestures like this.

In Table 4.4 we listed the results of Boletsis and Kongsvik [2], Chen et al. [3] and our measurement of the wpm value. Both 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,

Table 4.4: wpm for different VR text input methods

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 raycast keyboard	16.43
Word-gesture keyboard	12.75

but also not the one with the highest.

Another comparison we can do with the results of Boletsis and Kongsvik [2] is the comparison of the total ER value.

Table 4.5: total ER for different VR text input methods

text input method	wpm
Raycasting keyboard	11.05%
Drum-like keyboard	12.11%
Head-directed input keyboard	10.15%
Split keyboard	8.11%
Word-gesture keyboard	2.14%

The total ER for our keyboard seems to be much better than for the other input methods. A simple explanation could be that a user can make an error with every input. For the first four keyboards in Table 4.5 one input is equal to one character. Thus, if a user wants to input a phrase, they need to make inputs equal to the number of characters the phrase consists of. With a word-gesture keyboard, a user only has to make inputs equal to the amount of words in the phrase, which is usually much lower. And if the word is not the correct one, most of the time, it will not differ a lot from the word, that is intended to write. Therefore, in this case, there will not a lot of characters be added to the INF value for Equation (4.3).

The last comparison we want to make is with the EKS ER value from Chen et al. [3]. They state in the paper, that they used the EKS ER, but without the erroneous keystrokes, but the number of backspaces used. Therefore, this is defined the same way as our backspace error, and we can compare the values.

Table 4.6: total ER for different VR text input methods

text input method	backspace error
Word-gesture keyboard raycasting	15.64%
Our word-gesture keyboard	12.65%

In Table 4.7 we can see, that the error values are pretty much the same, although our value is a bit better. A reason for this might be, that they possibly did not implement, that a word can be deleted as whole. For example, if a word is wrong, with our implementation the user can just press the backspace key once and delete the whole word (only the first one, afterwards it deletes single characters).

Something we find interesting is the fastest possible wpm we could reach. In [4] Zhai and Kristensson made this experiment with four different phrases. We take the same phrases and compare their results with the ones we get. The four phrases are:
1. The quick brown fox jumps over the lazy dog
2. Ask not what your country can do for you
3. East west north south
4. Up down left right

Table 4.7: total ER for different VR text input methods

Phrase	$SHARK^2$ Author A	$SHARK^2$ Author B	Author of this work
1	69.0	70.3	47.78
2	51.6	60.0	48.48
3	74.4	72.9	70.22
4	74.1	85.6	58.04

We can see, that we can not reach the same maximal wpm, but especially for phrase 3 we can get close to their value. The word-gesture keyboard is obviously faster if it is on a small screen and a finger is used. The distances that have to be covered are much smaller than in VR, hence it makes it easier to write fast.

Overall we think, that the word-gesture approach we implemented might not be the fastest way to input words. At least this can be said for beginners. But it seems, that it has a lower error rate by far compared to single letter input methods.

4.5.2 New Implementations

Because of the feedback and some observations, we decide to do one more implementation.
It is not a big change in the code, but can still be fairly useful. We have the problem, that for example, "thee" is prioritized before "the", which does not make much sense, because "the" is used far more in plain language. Because the word list we are currently using is ordered by the frequency of the words, we decided to let this influence the best match and word suggestions. Now, if two or more words have the same highest score, the one that is highest in the word list will be taken, followed by the second highest and so on, until a word with a lower score comes next. This will prevent the problem with some words listed in Table 4.2.

5 Conclusion

This is the body of the thesis.

5.1 Results Discussion

5.2 Future Work

Bibliography

[1] Ahmed Arif and Wolfgang Stuerzlinger. Analysis of text entry performance metrics. pages 100 – 105, 2009.

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

[3] Sibo Chen, Junce Wang, Santiago Guerra, Neha Mittal, and Soravis Prakkamakul. Exploring word-gesture text entry techniques in virtual reality. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI EA '19, page 1–6, 2019.

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

[5] 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.

[6] 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.

[7] R. William Soukoreff and I. Scott MacKenzie. Metrics for text entry research: An evaluation of msd and kspc, and a new unified error metric. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '03, page 113–120, 2003.

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 .