UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
A Prototype for Text Input in Virtual Reality with a Swype-like Process Using a Hand-tracking Device

Permalink
https://escholarship.org/uc/item/43r3t7m8

Author
Jimenez, Janis Gisel

Publication Date
2017

Supplemental Material
https://escholarship.org/uc/item/43r3t7m8#supplemental

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Prototype for Text Input in Virtual Reality with a Swype-like Process Using a Hand-tracking Device

A thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science

by

Janis Gisel Jimenez

Committee in charge:

Jurgen Schulze, Chair
Ravi Ramamoorthi
Nadir Weibel

2017

Copyright
Janis Gisel Jimenez, 2017
All rights reserved.

The thesis of Janis Gisel Jimenez is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

____

____

____

Chair

University of California, San Diego

2017

TABLE OF CONTENTS

Signature Page iii

Table of Contents iv

List of Images v

List of Supplementary Files vi

Abstract of the Thesis vii

Chapter 1 Introduction to Virtual Reality 1

Chapter 2 Related Work in VR Text Input 5

Chapter 3 Backend 10

Chapter 4 User Interface 16

Chapter 5 Issues 21

Chapter 6 Conclusion and Future Work 25

Bibliography 26

LIST OF IMAGES

Image 1.1: A picture of the Sword of Damocles head mounted display. 2
Image 1.2: A picture of Nintendo‘s Virtual Boy console. 4
Image 1.3: A picture of the Oculus Rift headset and optional Touch controllers. 4

Image 2.1: A VR demo using a physical keyboard with the LEAP Motion Controller. 7
Image 2.2: A VR demo with a 3D keyboard and hand motion tracking where the user must press down each key. 7
Image 2.3: A VR demo with an upright 3D keyboard and hand motion tracking, which tracks key hovering rather than key pressing. 9
Image 2.4: A non-VR demo with Swype and hand motion tracking on an Android keyboard. 9

Image 3.1: The Oculus Rift DK2 headset with the LEAP Motion Controller attached. 11
Image 3.2: An example of a BK-tree sorted by the Levenshtein Distance between the words. 13
Image 3.3: Two matrices showing the Levenshtein Distances between the substrings of “kitten” and “sitting” on the left, and “Saturday” and “Sunday” on the right. 15

Image 4.1: A screenshot of “gaze” mode showing the letters keyboard. The user is typing out, “This is my thesis project!” 18
Image 4.2: A screenshot of “gaze” mode showing the symbols keyboard. 18
Image 4.3: A screenshot of “pointer” mode. 20

LIST OF SUPPLEMENTARY FILES

File 3.1: oculus_leap_keyboard.zip 15

ABSTRACT OF THE THESIS

A Prototype for Text Input in Virtual Reality with a Swype-like Process Using a Hand-tracking Device

by

Janis Gisel Jimenez

Master of Science in Computer Science

University of California, San Diego, 2017

Jurgen Schulze, Chair

Text input in virtual reality (VR) is a problem that does not currently have a widely accepted standard method. As VR headsets have become more commonplace, text input has also become more of a need. Using a physical keyboard is not possible with a head-mounted display that blocks the user's visual field. The two most popular solutions for text input in VR today are a virtual keyboard interfacing with VR controllers and voice recognition. However, they either require a handheld controller or a quiet environment. 3D-tracked controllers with a virtual keyboard can simulate a real keyboard to an extent, but they suffer from a lack of tactile feedback that makes typing slow and

unintuitive. A more intuitive solution is a Swype or SwiftKey-like algorithm, where the path that the user's finger travels is used as input, as opposed to individually pressing each key. I have implemented a prototype with the Oculus Rift and the LEAP Motion Controller that combines a novel Swype-like backend with hand gestures to demonstrate an all-purpose, intuitive method of text input. To compare it to state-of-the-art VR keyboard input, I implemented the virtual keyboard approach for hand-directed typing and head gaze typing.

Chapter 1

Introduction to Virtual Reality

Virtual reality is still a relatively new technology, but it organically grew from advancements in other fields. Even back in the 19th and early 20th centuries, the visual trickery that would eventually make VR a reality was already gaining traction in the early 1800s, as stereoscopy became popular [Unk96]. Stereoscopes, which resembled binoculars, worked by inserting two pictures (taken from slightly different angles) in such a way that each eye could only see one of the pictures. This created the illusion of 3D.

However, there was no way to gauge user input. Virtual reality‘s inspiration for user input came from a variety of fields, including the creation of flight simulators like the Link Trainers first constructed in the late 1920s [Low15]. These systems were utilized by the army to train pilots, using eye tracking and “force-feedback joysticks” to immerse them in the scenes painted in front of their test cockpit (which in later years, eventually incorporated film). The push to make better, more accurate simulators also catapulted computer graphics research forward. The two fields finally met in 1968 with the creation of the Sword of Damocles [Axw16]. This was the first headset to hook up to a computer, though it had to hang down from the ceiling in order to support its weight. It only showed

wireframes of basic shapes, but it was a step towards the virtual reality we know today at a time when the phrase “virtual reality” had not been popularized. It wasn‘t until 1987 that the phrase was coined in order to describe the controllers the Visual Programming lab was creating, such as their VR goggles and gloves [Cra16]. Still, the tech was very expensive and largely remained exclusive to research labs and enthusiasts until the 1990s.

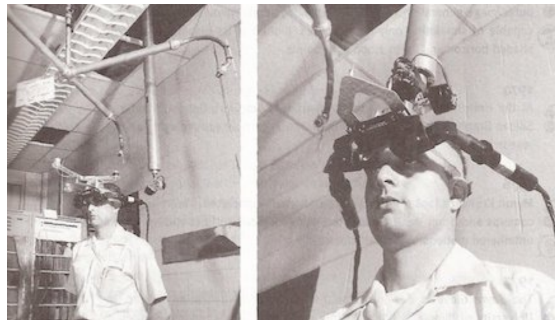

Image 1.1: A picture of the Sword of Damocles head mounted display.

In the 1990s, video game companies attempted to introduce VR to the consumer market, but their results were less than stellar. In 1993, SEGA revealed that they were developing a VR headset made for the Sega Genesis game console, but they had a number of critical production difficulties never prevented them from ever releasing the device. Nintendo was able to release the Virtual Boy in 1995, but it was a fairly uncomfortable system to play. It had to be placed on a stable chest-high surface [Kol08], and it could only display red and black, leading to eye strain if played for an extended amount of time. The headset was discontinued after only one year.

Skipping forward to 2011, there were still no major competitors in virtual reality. But the following year, Palmer Luckey launched a wildly successful Kickstarter to create the Oculus Rift [Luc12]. What started as a practical plan to make a few headsets for VR enthusiasts became the founding of a new company, which was later bought by Facebook. A consumer version of the Rift came out in March 2016, and by then, competitors had already made their own headsets to compete with it head-on.

Currently, VR is finally gaining traction in the consumer market as more companies release their own custom headsets, like the HTC Vive, Playstation VR, and the Gear VR, to name a few. Many consumers are barely buying their first headset, and more companies are rushing to beat out the market for the cheapest, lightest, or most user-friendly headset. Now that we have the hardware to work with, the next step will be to refine the software used for VR so that users can incorporate their headset in their daily lives.

Image 1.2: A picture of Nintendo‘s Virtual Boy console. [Edw15]

Image 1.3: A picture of the Oculus Rift headset and optional Touch controllers. [Ocu17]

Chapter 2

Related Work in VR Text Input

The main barrier to finding a comfortable and reliable way of taking in text input is the fact that virtual reality headsets block the user‘s vision. Some headsets are add-ons to existing hardware, so they have limited to no options for text input when they are being used in VR mode. This means they might not even come with a controller, choosing instead to rely on gaze selection (looking at a specific spot long enough to select it) along with any buttons that are already on the headset itself. A good example of this would be the Gear VR, which has circumvented the need for a controller with its voice recognition feature (currently in beta) [Alv15].

Those who do have a pre-existing controller follow the Playstation VR‘s example. Their solution is to use either the controllers that are already being used for the Playstation 4 or their Playstation Move motion controllers. Their system for typing messages is therefore the same as it is has been for most consoles: with a controller, the user uses the thumbstick (or touchpad) to slide over to the letter they want. With motion controls, the controller itself becomes a cursor in 2D space that the user moves over to the desired letter. Each letter must be individually confirmed with the push of a button [Shu16].

Since these headsets are mainly for playing video games, there is no pressing

demand to accommodate for text input, especially when the user has ways to input text on the existing system with their headset off. However, there is a growing demand when it comes to staying connected to one‘s phone while in VR. Some headsets allow the user to connect their phone through Bluetooth (or the headset display is their phone). The HTC Vive, for example, allows the user to see text messages sent by their contacts and call them with the built-in mic. If it‘s Vive for Android, the user can also send “Quick Replies“, which are pre-made messages [Rob16]. Generally, the user is expected to take off their VR headset to type a custom message.

For headsets hooked up to a computer, the straightforward solution is to use the keyboard. But again, these headsets completely block the user‘s vision. The Oculus Rift, HTC Vive, and Playstation VR all prefer to use their own controllers because they will always be in the user‘s hand. This mostly removes the issue of having to grope around in the dark for the right button once the user becomes familiar with the controller‘s button layout, and allows them to move around instead of having to sit at their desk. However, there are some independent projects that accepted this constraint and have made prototypes involving a physical keyboard. One project created a 3D replica of their keyboard and used the LEAP Motion controller to track where their hands are, so the user can see both their hands and the keyboard through their VR headset [Sle14]. It is an ideal solution when sitting in one‘s office and it is easy to learn, which will result in a fairly high word-per-minute performance compared to the one-letter-at-a-time input of most controllers, but not for general use.

In lieu of having to use a physical keyboard, some have opted to use a digital one. A big hurdle that this option has to overcome is the lack of tactile feedback, because there is no tangible controller to assure the user that their input was received. If there is no physical surface to press against, how can the user intuitively know when they are pressing something? For some, the solution is to create hardware that can simulate that

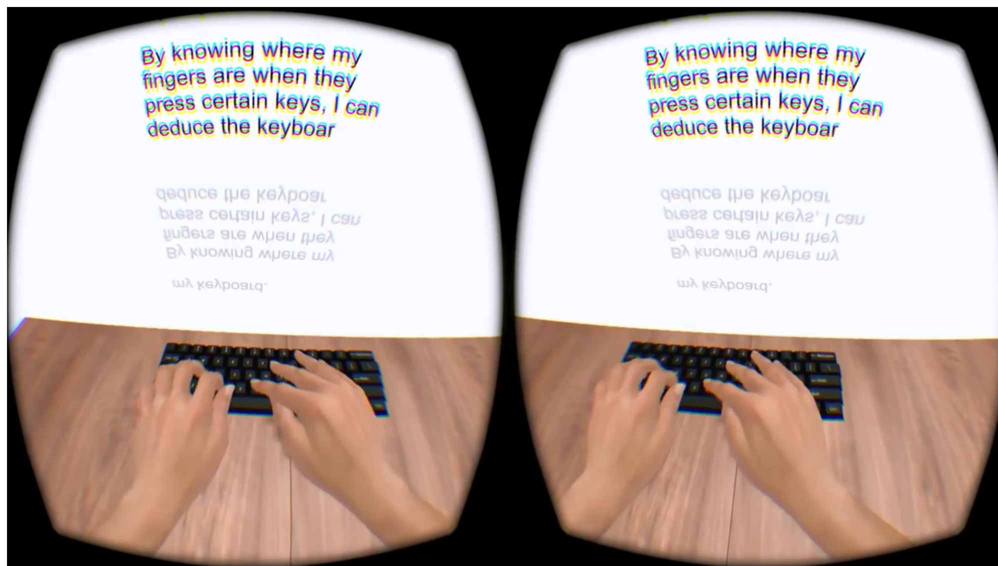

Image 2.1: A demo using a physical keyboard and hand motion tracking. [Sle14]

Image 2.2: A demo with a 3D keyboard and hand motion tracking where the user must press down each key. [Cor16]

feedback. A group of engineering students from Rice University created special gloves that use “bladders” that expand and contract to simulate the sensation of touch in VR [Wil15]. However, it is still in the prototype phase, so how well it performs with a 3D keyboard remains to be seen.

Some have made prototypes that provide ample visual and audio feedback to try to make up for the lack of tactile feedback. Students and VR enthusiasts have created

a variety of virtual keyboard designs that can interact with hand-tracking sensors (such as the LEAP Motion Controller or the Kinect): One has the appearance of a regular QWERTY keyboard on a mostly flat surface, with keys that move down as the user‘s fingers press them [Cor16]. Another has the keyboard directly facing the user, with keys lighting up and making a clicking sound when the user‘s finger is near enough to them [Wre14]. Some may also add in a “hover” check, where the user keeps their finger pressed on a key for a short amount of time to confirm that it is the desired key [Kin15]. These projects do free up the user‘s hands by not requiring a controller, but the lack of tactile feedback and the imperfect tracking mechanisms of the sensors make typing slow and error-prone.

There are other ways to type on a virtual keyboard, however. The difficulty in selecting keys partially stems from a depth-sensing issue, so a good way to circumvent this is to not rely on depth at all. Some keyboards rely exclusively on the aforementioned “hover” check. This is how the Myo armband, a motion control device, allows the user to search the web [Myo16]. But there is still a more efficient way that does not force the user to wait for every letter they want to type.

The Swype method of typing also does not need depth to work [Swy17]. On a phone, typing with Swype works as follows: the user places their finger on the first letter of the word, then drags their finger to each required letter without stopping. They only lift up their finger once the word is finished. Swype will then guess what word they were typing. If it guesses wrong, multiple auto-complete options would be available to choose from. A phone knows when to start and stop tracking input by sensing when the user‘s finger is touching the screen, but in VR, it could be controlled by a button on the headset or a hand gesture. There have been tests using the LEAP Motion with a Swype keyboard [Zug13], and there have been rumors of Swype looking into virtual reality [Seg11], but no big competitor has implemented this yet. So for my prototype, I chose to make a VR

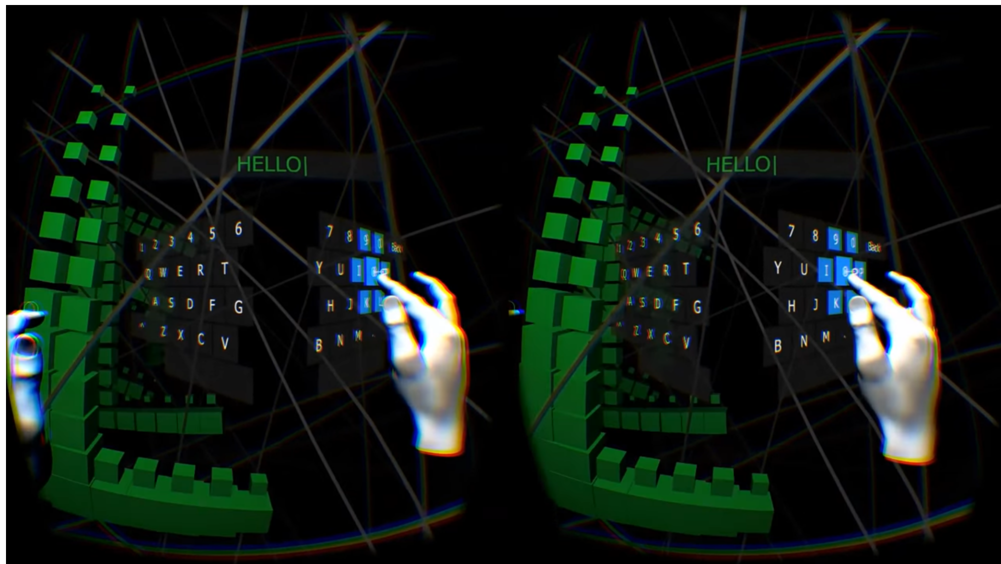

Image 2.3: A VR demo with an upright 3D keyboard and hand motion tracking, which tracks key hovering rather than key pressing. [Kin15]

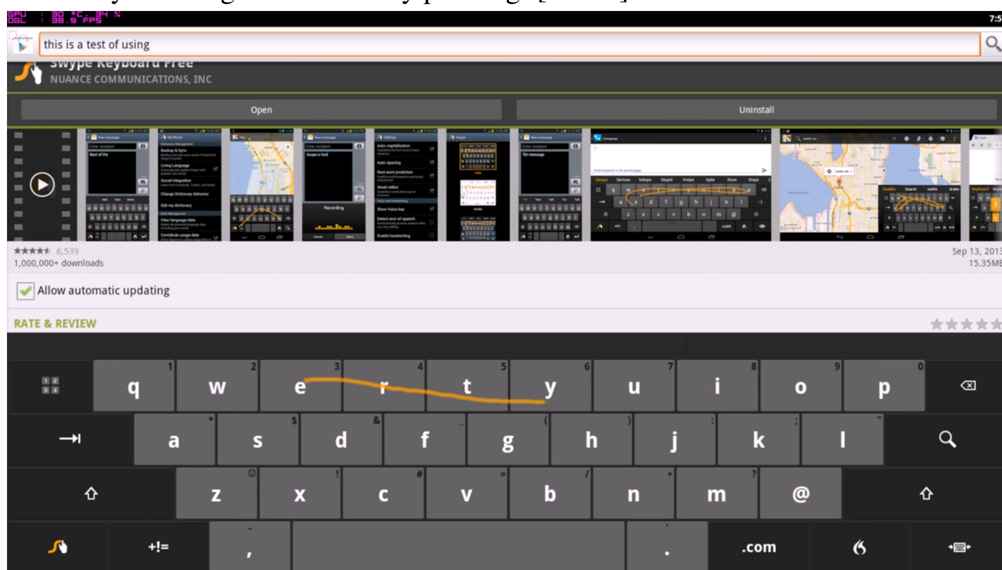

Image 2.4: A non-VR demo with Swype and hand motion tracking on an Android keyboard. [Zug13]

demo with hand tracking and a 3D keyboard like the examples mentioned above, but I combined it with a Swype-like method of text input to demonstrate how this method works in virtual reality.

Chapter 3

Backend

The prototype was created in Unity version 5.5.1f3 and written in C#. When deciding between Unity or Unreal Engine, two of the most popular game development platforms with VR compatibility, I chose Unity for a number of reasons. A large majority of developers use Unity instead of Unreal because of how quickly they can create a prototype [Jag16]. Many of the projects mentioned in the Related Work section used Unity to develop their prototypes. It was also the most convenient for me because I already have a few years of experience creating video game prototypes with Unity, so it was easy to pick up.

While there are many commercially available headsets, I focused on choosing a headset that is widely used and compatible with game-developing programs. I used a fairly high-end desktop, so I went with a PC-based headset. The Oculus Rift and HTC Vive are two of the most popular PC-based headsets, and are compatible with a variety of devices. Unity is mainly compatible with the Oculus Rift, Gear VR, the HTC Vive, and the HoloLens, though it continues to add support for more types of headsets [Uni17]. A DK2 headset was readily available, so I went with Oculus, though Unity should have the appropriate settings for this prototype work with the HTC Vive as well.

As for a hand-tracking device, the two main competitors in use are the LEAP Motion Controller and the second-generation Kinect (because the first-generation Kinect can't keep track of individual fingers). Both are compatible with the Oculus Rift, but the library that the LEAP Motion Controller comes with is much easier to work with when it comes to tracking individual fingers, while the Kinect V2 requires finding a third-party library or algorithm [Pte16]. The camera placement is also more convenient since it is placed directly onto the headset, so I ultimately chose the LEAP Motion Controller. The LEAP Motion SDK version for this project is 2.3.1.

I developed this on a Windows 10 operating system (OS). Besides the fact that it was readily available, it is the recommended OS by Oculus because they have stopped development on OS X and Linux for the time being [Bin15]. The LEAP Motion Controller SDK is also only available for Windows and Android [Mot17a].

Image 3.1: The Oculus Rift DK2 headset with the LEAP Motion Controller attached. [Mot17b]

For the "Swype-like" algorithm, Swype, Swiftkey, and Google Keyboard have no publicly available source code online, though Android phones automatically have a Swype keyboard. However, the Android keyboard has a limited API to work with that is

targeted towards touchscreens, and is quite difficult to interact with through scripting, so I decided against directly using this keyboard and instead made my own keyboard with Unity UI. There are many alternative open source projects available online, but I felt it was prudent to make my own. Most if not all lack the complexity that the commercial apps have under the hood, and my own backend would be easier to debug and build upon. For my prototype, I focused on creating an intuitive, responsive keyboard and determining effective criteria for filtering user input in the front-end and a fast, accurate way to compare a given word to a dictionary and auto-correct it on the back-end.

First, I needed a way to store words into a data structure that could be built and traversed in polynomial time. A hash table has quick O(1) access, but searching for the right word could lead to O(n) search time, which would be ill-advised for a large dictionary. Instead, I mainly looked at a ternary trie and a BK tree. Both essentually have O(logn) search time, so I looked at how the words were organized in each structure. A ternary trie has a structure similar to both Binary Search Trees and Tries. Like a trie, it would store words character-by-character, requiring w nodes for a word of length w. A trie, however, would have all 26 letters as its children at every letter, so it would be very space-inefficient with a large dictionary. Instead, a ternary trie is more like a binary tree, but with up to three children at every level instead of two. The left child has a value less than the current node, and the right child has a value greater than it. The middle child is for a node with a value that lies between the two other children.

The main issue with this structure was that it was suited more for spell-checking and auto-complete, not auto-correct. The first two rely on a sequential procedure, while auto-correct has to cover many more cases. When a user types a word, they could accidentally transpose two letters, or forget a letter in the middle of the word. They could even do this multiple times within the same word. A ternary trie orders the letters of words sequentially, though it can store a “distance” that would allow it to compare just

how different two words are from each other. With some tolerance to check against, it could still traverse the data structure and find all words that fit with in the tolerance, but this means that it would have to compare every substring it finds in its path, not just every word.

I ultimately went with creating a BK-tree [Joh07] [Xen13], a tree structure proposed by Burkhard and Keller that relies on sorting words by the distance between them. This means that each node is a word, not a letter. First, it picks an arbitrary node as its root. Then, to add a node, it finds the distance between new node and the root of the tree. It then traverses its children until it finds another node that has the same distance. If it does, then it traverses that node‘s children, and continues doing so recursively until it cannot find a node with the same distance. When it reaches this point, the new node is added as a child to the node it stopped on. Building the tree takes about O(nlogn): it takes O(logn) to place a single node into the tree, and it places O(n) nodes.

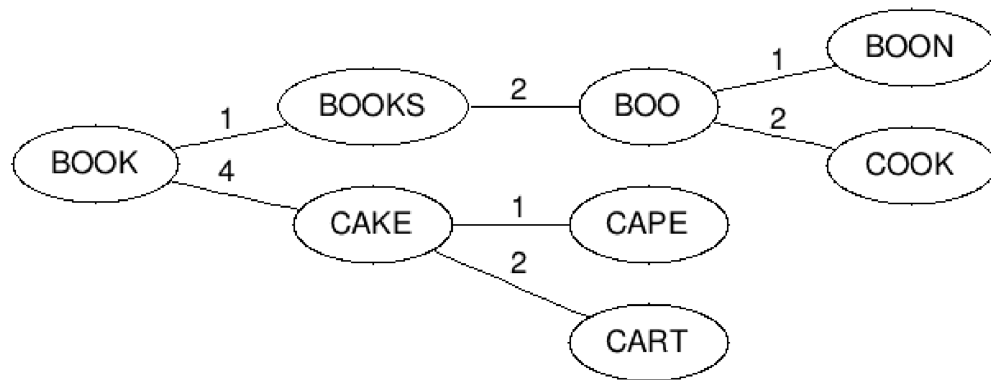

Image 3.2: An example of a BK-tree sorted by the Levenshtein Distance between the words. [Xen13]

The distance used by the BK-tree is the “Levenshtein Distance,” [Wik17b] a number representing how many changes need to be done to a word (deletions, insertions, or mismatched letters) in order to turn it into another word. I relied on this metric because it is a simple, discrete way to keep track of differences between strings that is commonly utilized for a BK-tree (and is sometimes used in ternary tries as well). For example, to

change the word “cat” to “gate”, the ‘c’ must be changed to a ‘g’ and an ‘e’ must be added to the end. Two changes must be made, so the distance between “cat” and “gate” is 2. The Levenshtein Distance code works by maintaining a matrix of integers holding the distance between substrings of the two words. It iterates through both words, letter by letter, and adds 1 to the count in the current index. Once it finishes iterating, it returns the value at the last index.

The original Levenshtein Distance equation does not take transpositions into account (“cat” and “chat” would have a distance of 3 without transpositions, and 1 with transpositions), so I coded in the transposition check made by the Damerau-Levenshtein Distance version [Wik17a]. In addition to this, I added an extra check: a “keyboard neighbors” check. When typing, it is not uncommon for the user to accidentally choose a letter neighboring the letter they actually wanted. To account for this, I have a dictionary storing the surrounding neighbors for every letter and symbol on the keyboard. Each time it makes a comparison between the two words, it will check if the two letters are neighbors of each other. If so, it will add 1 to the distance; otherwise, it will add 2. To balance this change, every increment that is not between two neighboring letters will be doubled. This means that each deletion, insertion, and non-neighbor mismatch will add 2 to the distance instead of 1.

The next step is using the tree to come up with auto-correct candidates. Once the user has finished typing a word, it traverses the tree to find candidates within a certain tolerance threshold. This O(logn) search process is as follows: it starts with a queue containing the root of the tree and finds the distance between the new word and the root word. If it falls within the tolerance threshold, it is added to a list of possible auto-correct candidates. Then it looks through its children and adds them to the queue if they fall between the range of the previously calculated distance plus or minus the tolerance. The current node is then dequeued and the process loops until nothing is left in the queue.

		k	i	t	t	e	n
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	3

		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

Image 3.3: Two matrices showing the Levenshtein Distances between the substrings of “kitten” and “sitting” on the left, and “Saturday” and “Sunday” on the right. [Wik17b]

The list of candidates is sorted by lowest to highest distance to the new word. The first three words in the list are then displayed in the scene as possible candidates that the user can select to correct the word they just typed. The results are then passed on to the front-end logic, which will then display the results to the user.

File 3.1: The Unity prototype is available on ProQuest as a supplementary zip file.

Chapter 4

User Interface

After looking through some demos online, I decided to build off of the "Unitys UI System in VR" demo and tutorial provided online from the Oculus site [Bor15]. The main component that was crucial to this prototype was their gaze pointing code, which has a cursor directly in the middle of the screen that follows the user‘s head motions. It provided a reliable backend for interacting with Unity‘s user interface (UI) with a cursor. I combined this with the LEAP Motion Controller in two different ways: camera "gaze" mode and finger "pointer" mode. In "gaze" mode, the cursor is still tied to head movement, while in "pointer" mode, the cursor is controlled by the pointer finger of the right hand. Both modes are augmented by hand gestures, which control keyboard interactions. There is also a third mode called "one-letter", which will turn off the Swype aspect of either mode and only take in one letter at a time. Note that here, when a button is "pressed", it means the user made the appropriate gesture while the cursor is above that button.

The LEAP Motion Controller SDK provides multiple default hand models. I chose the wire hand models because the colors make it easy to know which hand the computer considers "left" or "right" at any point. The wire model also blocks less of the

user's view, which makes it easier to see the rest of the scene.

The scene is a flat 2D keyboard panel in world space made with Unity‘s UI that has the same basic layout as a typical phone keyboard, mainly so that most users will already be familiar with the layout. It has the default ABC mode, and a SYM mode with some basic symbols. To the left of the keyboard is the text box panel where the user‘s text is displayed. The backspace key will delete the rightmost letter that has been typed into the text box if pressed. Note that if the shift key is pressed, then all subsequent words typed by the user (if not in "one-letter" mode) will be capitalized until it is pressed again. If the user is in "one-letter" mode, then every subsequent letter typed by the user will be capitalized until it is pressed again. Above these two panels is the auto-correct panel, which will update with the recommended auto-correct candidates. It also has buttons that the user can press to switch between the different modes, and a "Clear" button that will erase all the text in the text box panel when pressed.

I put the keyboard in World Space mainly due to the resulting size of my keyboard and text box. If I shrank the keyboard more, it became difficult to see the individual keys, and putting it in Screen Space would have forced it to this size so that the whole UI component would be visible. Although the text box panel would technically be on top of the keyboard in a typical Android keyboard layout, I placed it to the side of the keyboard to give the keyboard center stage in the scene, rather than push it towards the bottom. The spacing between the keys also imitates the Android Samsung S7 phone layout I was using as a reference, but it was also so that it would be harder for the user to swipe over a letter they didn't intend to include, due to the large size of the cursor.

The cursor itself was taken from the Oculus demo and repainted to match the colors of the scene. The size of the cursor is different depending on the mode the user is in. In gaze mode, the cursor is the same size as what the demo provided because it doesn't get in the way of the background but is also big enough to see. The circle shape

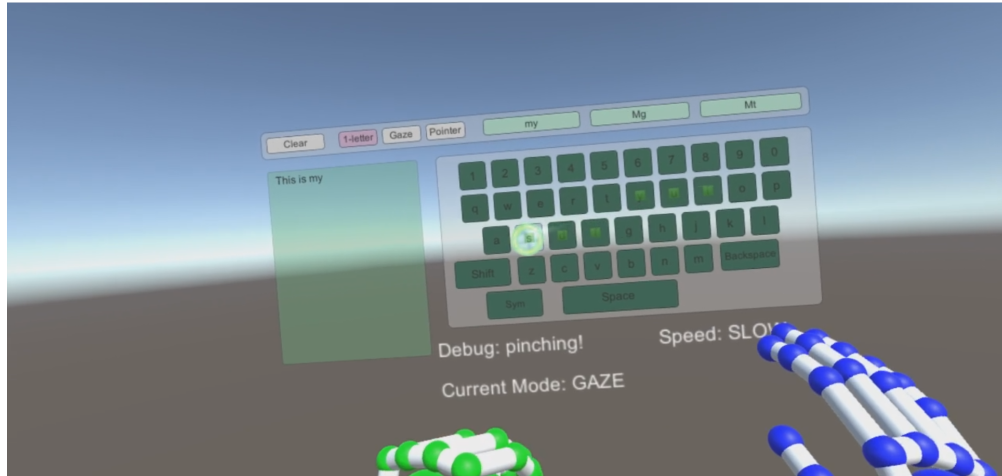

Image 4.1: A screenshot of “gaze” mode showing the letters keyboard. The user is typing out “This is my thesis project!”

Image 4.2: A screenshot of “gaze” mode showing the symbols keyboard.

with a hole in the center is also useful because the letter will be visible in the center when the user is hovering over it instead of blocking it. Pointer mode, however, slightly increases the size of the cursor, as seen in image 4.3. This is because pointer mode is harder to control and jitters a bit, so the larger cursor size helps to stabilize it.

The Oculus demo handles the logic behind how the cursor knows what button it is hovering over. Originally I had tried using 2D collisions between the colliders of the cursor and the keyboard buttons, but there was a peculiar invisible offset that the collider boundaries would not show, which skewed the input the keyboard received. The demo‘s

logic instead relies on casting a ray away from the camera towards the scene and keeping track of what it collides with.

Below the keyboard is some text kept for the purpose of debugging. The top left text states whether it correctly knows that the user is pinching or not. If the user is pinching, it will say "Debug: pinching!", otherwise it will say "Debug: not pinching". The text on the right keeps track of the user's speed. The threshold for what is considered fast or slow is currently determined by a hard-coded number in the code, so it will say "Speed: SLOW" if at or below that speed, and "Speed: FAST" otherwise. The bottom left text object states what the current mode is. At startup, the default mode is gaze, so the text will say "Current Mode: GAZE".

The main gesture used in this project is the "pinch" gesture. This gesture was chosen because most of the other gestures were removed by the Orion update to LEAP Motion to be reworked. The "pinch" gesture is actually a short method that checks if each finger is bent or not. If all five fingers are bent, then the hand is "pinching," otherwise it is not. When the left hand does a pinch gesture, the cursor will start reading in the letters it is directly hovering over. It will keep doing this until the left hand stops pinching. At that point, it will add the word to the text box, then calculate and display the possible auto-correct candidates. If it is in "one-letter" mode for either of these options, then instead the user will type one letter at a time. They can go to the desired letter, make the pinching gesture with their left hand, then stop pinching and it will add that letter to the text box panel.

As for the amount of auto-correct candidates, the Android phone also has three candidates. I attempted adding more, but that meant shrinking the buttons. Sometimes the length of the word would extend past the button, and shrinking the font made it difficult to read the word. It would be possible to comfortably add more if the UI panels were stretched out more to the sides, but I kept it at three to maintain the current layout style.

If the cursor just took in every letter that the user hovered over, it would be difficult to make out any comprehensible word, so there had to be a restriction on what letters were included in the final word. I had at first tried to weight the letters using a combination of velocity checking and the change in direction. Basically, as the cursor is moving, it would constantly update the current velocity of the cursor. Every few frames, it would create a vector in the direction it‘s currently facing. When it reached a letter, if the velocity was below a certain number, it would compare the current vector to the previous vector it calculated to find the change in angle. This angle would be used to give the letter a ‘weight,’ and only letters above a certain weight would be included in the final word. The rest would be handled by the auto-correct.

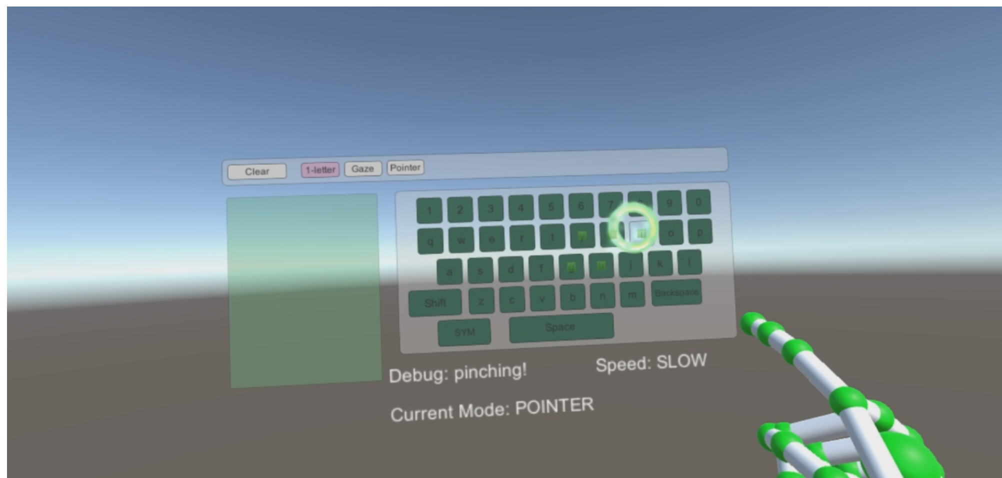

Image 4.3: A screenshot of “pointer” mode.

In practice, however, the change in angle did not seem to follow the desired behavior, and I instead decided to simplify so it would only do a velocity check. When the user is going across the letters, all of them are added to the word, but they have a negative weight. If the user slows down on a letter below some threshold, the letter‘s weight is updated to a positive value. Once the user stops making a pinching gesture, all the letters that have a non-negative weight are added to the final word and returned.

Chapter 5

Issues

Since this is a simpler prototype, there are several areas where it could benefit from further improvements. Hardware-wise, using a CV1 Oculus Rift headset for this project would be better than the DK2 headset I used. The DK2 was the best choice at the time, but at this point it is no longer supported, so any more recent updates made to the Oculus have resulted in strange intermittent problems. The DK2 must be able to connect to "Unknown Sources" to work with Unity, and has the habit of going dark without warning during play. Sometimes it even reverses the left and right screens, making it unusable until the demo is stopped and restarted. Concerning my own prototype, one major issue is how unstable and shaky the cursor is when in the second mode (using the right hand). This is due to the cursor's dependency on the movement of the head, and the shorter distance between the pointer finger and cursor (compared to the camera and cursor). Smoothing out the movement would likely improve the speed and accuracy of this mode.

Concerning the code, the most troublesome part would have to be the auto-correct. While it does contain the desired word most of the time, my UI only displays the first three results, weighted by distance. If there are more than three words that have the

same distance, there‘s a chance the desired word will not be displayed as one of the three options. For earlier tests, I used a small dictionary of a few hundred words to catch any larger-scale bugs that could appear. Then I switched over to a much larger dictionary, which is directly from Linux 14.04‘s dictionary file. Suddenly, the auto-correct‘s accuracy dropped significantly because there were many more words that fell into the same Levenshtein Distance tolerance. I attempted to fix this using two different approaches, and neither one passed what I called a “hello” test. This simply means that the user types out the word “hello” using the Swype-like method and looks at the three candidates that show up.

I chose the word “hello” because it is a fairly common word that has double letters in it. duplicate letters cannot currently be typed out with this prototype. I considered adding in a gesture that would indicate that a letter would be counted twice, but the more natural solution is that the autocorrect should be smart enough to account for it. So instead, I have a test for local repeats that decreases the overall distance by one for every immediate double found in the word (i.e. the two t‘s in “potted” count, but not the ones in “potato”). This could have unintended consequences on the calculated Levenshtein distances, however.

The first approach I wrote was to sort the candidates by their levenshtein distance to the user’s typed word. Those with the lowest Levenshtein distance would be at the front of the list. However, other words like “hell,” “well,” and “bell” would show up first simply because they were found as candidates before “hello” was found, and they have the same Levenshtein distance. Then I instead used a 5,000 word frequency dictionary from a word frequency site [Fre17] and sorted the words by frequency. Even then, words like “well”, “tell”, and “help” would come first because they were considered more frequent than “hello”. Even sorting by both methods combined would not be enough. Note that in both of these cases, “hello” is correctly found as a candidate, but it does not

reach the criteria to make it to the front of the list. The ideal solution would be to train it on user input so that it would learn what words are more commonly typed and offer more accurate suggestions, but I did not have the time to implement this.

In the back-end chapter, I mentioned that when calculating the Levenshtein Distance, it would check if the two compared letters were neighbors on the keybord. This "keyboard neighbors" dictionary stores the neighbors of letters as they appear in a typical Android keyboard, but any other keyboard would have to manually change this because the entries are all hard-coded strings of characters.

It is also missing some symbols that usually come with phone keyboards (curly braces and the tilde, to name a few). My current dictionary was filtered to leave out any umlauts, accent marks, and so on for the sake of simplicity. It also uses case-insensitive matching when calculating Levenshtein distance.

Concerning speed, the prototype would rarely experience low framerates except when I would purposely cause it to slow down and print out debug output. The one exception is when I used the Linux dictionary file, which contains around 99,000 rows of words. It would take around 6 seconds at start-up to build the dictionary, freezing the program until it finished. Perhaps multithreading this in some way, along with having a smaller dictionary of the most common words so the user can start typing immediately, could alleviate this drop in performance. After startup, however, searches were still fast and there were no significant framerate drops.

Other than the somewhat unreliable auto-correct performance, another issue that keeps it from truly imitating Swype is the fact that it weights letters by speed. If a user does not slow down for a few milliseconds at each letter, the program might not include that letter in the final word. At that point, it starts to resemble a "hover" technique more than Swype. It is still more efficient, because the waiting times for most hover-based programs is still much longer, but the user cannot simply remain at the same fast speed

the entire time and still obtain their desired word, so balancing the speed check with other factors would be critical.

Chapter 6

Conclusion and Future Work

For this project, I was able to create a prototype utilizing a Swype-like algorithm in virtual reality using a hand-tracking device to simulate typing on a 3D keyboard. I implemented both a gaze-based and a finger-based cursor control, and included a single-letter mode to compare with the Swype-like mode. A useful next step for this project would be to hook up the UI from thie project to the official Swype or Swiftkey backend and then test its performance. This project has several shortcomings that cannot quite measure up to a polished, widely-used back-end that has been optimized through several iterations, so having access to something more stable would be ideal.

If the current prototype back-end code was kept, the auto-correct would have to be greatly improved. As mentioned in the Issues chapter, training it on user input would be a great way to teach it to display more accurate suggestions. As for the hand-tracking, one way to enhance it would be to combine it with a controller with built-in haptic feedback, like the Hands Omni gloves from Rice University that inflate “bladders” [Wil15], or the GloveOne gloves that use vibrations [Glo15]. While my approach sidesteps the need for haptic feedback, it would be useful to test the Swype approach in VR with an experience that better resembles the sensation of using Swype on one’s phone.

Bibliography

[Alv15] Edgar Alvarez. Samsung made a web browser for the gear vr. https://www.engadget.com/2015/12/01/samsung-internet-for-gear-vr/, 2015. Engadget, Accessed: 2017-02-20.

[Axw16] Jon Axworthy. The origins of virtual reality. https://www.wareable.com/wearable-tech/origins-of-virtual-reality-2535, 2016. Wareable, Accessed: 2017-02-22.

[Bin15] Atman Binstock. Powering the rift. https://www3.oculus.com/en-us/blog/powering-the-rift/, 2015. Oculus Blog, Accessed: 2017-03-15.

[Bor15] Andy Borrell. Unity‘s ui system in vr. https://developer3.oculus.com/blog/unitys-ui-system-in-vr/, 2015. Oculus VR, Accessed: 2016-08-15.

[Cor16] Joshua Corvinus. Vr hex keyboard using leap motion. https://www.youtube.com/watch?v=ZERwYJVZOgk, 2016. Youtube, Accessed: 2017-02-20.

[Cra16] CrashCourse. The future of virtual reality: Crash course games. https://www.youtube.com/watch?v=BfcBjJ3c9lg, 2016. PBS Digital Studios, Accessed: 2017-02-15.

[Edw15] Benj Edwards. Unraveling the enigma of nintendo‘s virtual boy, 20 years later. https://www.fastcompany.com/3050016/unraveling-the-enigma-of-nintendos-virtual-boy-20-years-later, 2015. Fast Company, Accessed: 2017-03-1.

[Fre17] Word Frequency. Corpus of contemporary american english. http://www.wordfrequency.info, 2017. Word Frequency data, Accessed: 2017-03-07.

[Glo15] GloveOne. Gloveone: Feel virtual reality. https://www.kickstarter.com/projects/gloveone/gloveone-feel-virtual-reality, 2015. Kickstarter, Accessed: 2017-03-15.

[Jag16] David Jagneaux. Why epic‘s tim sweeney is fine with more devs using unity. https://uploadvr.com/tim-sweeney-on-unreal-vs-unity-priority-on-shipping/-first-and-foremost-with-ease-of-use-accessibility-being-second/, 2016. Upload (remove / after ‘shipping’ to use URL correctly), Accessed: 2017-03-15.

[Joh07] Nick Johnson. Damn cool algorithms, part 1: Bk-trees. http://blog.notdot.net/2007/4/Damn-Cool-Algorithms-Part-1-BK-Trees, 2007. Nick’s Blog, Accessed: 2016-08-15.

[Kin15] Zach Kinstner. Hoverboard vr interface. https://www.youtube.com/watch?v=hFpdHjA9uR8, 2015. Youtube, Accessed: 2017-02-20.

[Kol08] Patrick Kolan. Ign retro: Virtual boy revisited. http://www.ign.com/articles/2008/01/14/ign-retro-virtual-boy-revisited, 2008. IGN, Accessed: 2017-02-22.

[Low15] Henry E. Lowood. virtual reality (vr). https://www.britannica.com/technology/virtual-reality, 2015. Encyclopaedia Britannica, Accessed: 2017-02-26.

[Luc12] Palmer Luckey. Oculus rift: Step into the game. https://www.kickstarter.com/projects/1523379957/oculus-rift-step-into-the-game, 2012. Kickstarter, Accessed: 2017-02-22.

[Mot17a] LEAP Motion. Leap motion developer. https://developer.leapmotion.com/#101, 2017. LEAP Motion, Accessed: 2017-03-15.

[Mot17b] LEAP Motion. Oculus rift dk2 setup. https://developer.leapmotion.com/vr-setup/dk2, 2017. LEAP Motion, Accessed: 2017-03-15.

[Myo16] Myo. Connect with myo. https://www.myo.com/connect, 2016. Myo, Accessed: 2017-03-15.

[Ocu17] Oculus. Oculus. https://www.oculus.com/, 2017. Oculus, Accessed: 2017-03-15.

[Pte16] Vangos Pterneas. Finger tracking using kinect v2. http://pterneas.com/2016/01/24/kinect-finger-tracking/, 2016. Vangos Pterneas Blog, Accessed: 2017-03-15.

[Rob16] Adi Robertson. Htc‘s hands-free vr phone tool is as clever and frustrating as the vive itself. http://www.theverge.com/2016/4/6/11377740/htc-vive-vr-bluetooth-phone-notifications-hands-on, 2016. The Verge, Accessed: 2017-02-20.

[Seg11] Sascha Segan. Swype working on typing by waving hands in the air. http://www.pcmag.com/article2/0,2817,2380227,00.asp#fbid=bidQg4EBxv7, 2011. PC Mag, Accessed: 2017-01-25.

[Shu16] Sid Shuman. Playstation vr: The ultimate faq. http://blog.us.playstation.com/2016/10/03/playstation-vr-the-ultimate-faq/, 2016. Playstation, Accessed: 2017-02-20.

[Sle14] Lachlan Sleight. Leap keyboard demo. https://www.youtube.com/watch?v=ckAGpmf21a8, 2014. Youtube, Accessed: 2017-02-20.

[Swy17] Swype. Swype home. http://www.swype.com/, 2017. Swype, Accessed: 2017-02-20.

[Uni17] Unity. Vr devices. https://docs.unity3d.com/Manual/VRDevices.html, 2017. Unity Documentation, Accessed: 2017-03-15.

[Unk96] Unknown. The history of stereo photography. http://www.arts.rpi.edu/~ruiz/stereo_history/text/historystereog.html, 1996. Rensselaer Polytechnic Institute, Accessed: 2017-02-28.

[Wik17a] Wikipedia. Dameraulevenshtein distance. https://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance, 2017. Wikipedia, Accessed: 2016-08-15.

[Wik17b] Wikipedia. Levenshtein distance. https://en.wikipedia.org/wiki/Levenshtein_distance, 2017. Wikipedia, Accessed: 2016-08-15.

[Wil15] Mike Williams. Gamers feel the glove from rice engineers. http://news.rice.edu/2015/04/22/gamers-feel-the-glove-from-rice-engineers-2/, 2015. Rice University News and Media, Accessed: 2017-03-15.

[Wre14] Chris Wren. www.wrenar.com vr keyboard w/oculus rift and leap motion. https://www.youtube.com/watch?v=67Hyb2w1xFs, 2014. Youtube, Accessed: 2017-02-20.

[Xen13] Xenopax. The bk-tree - a data structure for spell checking. https://nullwords.wordpress.com/2013/03/13/the-bk-tree-a-data-structure-for-spell-checking/, 2013. Xenopax, Accessed: 2016-08-15.

[Zug13] Adrian Zugaj. Test: Using leap motion with swype keyboard. https://www.youtube.com/watch?v=_-4k0PnU46o, 2013. Youtube, Accessed: 2017-02-20.