Vrije Universiteit Brussel

FACULTEIT INGENIEURSWETENSCHAPPEN

# SpeeG: A Speech- and Gesture-based Text Input Device

Proefschrift ingediend met het oog op het behalen van de titel Master in de Ingenieurswetenschappen: Toegepaste computerwetenschappen, door:

Jorn De Baerdemaeker

Academiejaar 2010 - 2011

Promotor: Prof. Dr. Beat Signer
Begeleider: Lode Hoste

Vrije Universiteit Brussel

FACULTY OF CIVIL ENGINEERING

# SpeeG: A Speech- and Gesture-based Text Input Device

Dissertation submitted in partial fulllment of the requirements for the degree of Master in Engineering: Applied Computer Science, :

Jorn De Baerdemaeker

Academic year 2010 - 2011

Promotor: Prof. Dr. Beat Signer
Advisor: Lode Hoste

## Abstract

Nowadays, the input of text on TV set-top boxes or game consoles is often restricted and an unpleasant experience. Due to the limited space on a TV remote control, users are required to press the same button multiple times before the intended character is selected. Likewise, on game consoles, such as the Wii or Xbox, users select single characters from a virtual on-screen keyboard, which is also a very inefficient way of entering text.

Speech recognition input is recognised to be the holy grail of efficient text-input for many years. However, after numerous attempts in academia and industry, we are still not close to 100% recognition rates. Additionally, a speech recogniser typically enforces users to initiate a user-specific training session.

The goal of this thesis was to overcome both of these issues. We developed SpeeG, a prototype for text input via a combination of speech recognition and hand-gestures. Hand gestures are used to apply corrections of the speech-recognised text. This voice input and gesture-based input, combined with a fluent, intuitive zooming interface which we optimised for continuous voice input, is presented as the next generation of efficient text input for set-top boxes and game consoles.

Initial experiments confirm that SpeeG is a promising new input device as users draw positive conclusions from their experiences with our prototype application.

## Acknowledgements

This dissertation is made by one person but without the support and the help of other people this would not have been possible.

First of all, I would like to thank my excellent advisor Lode Hoste. The effort that he did was indescribable. He helped me a lot with pointing out the direction, and guided me whenever I was stuck.

The next person that I would like to thank is Professor Beat Signer. He was the one who believed that I was capable for switching a full thesis in the middle of the year. He was the person who helped me searching for a new topic and I could borrow his Kinect that I needed for this thesis.

There are a lot of people who supported me. The following person is someone whom I can call every moment of the day; that is at least if she does not let her mobile phone linger around, and that is my girlfriend.

Another person to which I would like to say thank you is Kevin Puttemans. He started the same studies with me, *Applied Computer Science*. I learned so much from him that if I were to write it all down, this dissertation would be twice as long. So, thank you Kevin.

# Table of Contents

# Figure List

# Chapter 1: Introduction

Today, almost all households have set-top boxes in their living room. Set-top boxes are used to program the television, to record some programmes, to connect to the Internet, to play some games or to retrieve more information of specific programmes.

Currently, interacting with a set-top box is done with a remote controller. Still, in the gaming console industry we see different ways how a user can interact with their game. A few years ago, the Wii of Nintendo was something revolutionary on the market. The Wii remote was different from the remotes that we used to see in the gaming console industry. With this remote you can point in a certain direction and the cursor is moved to point where you direct to.

*Figure 1: Wii Mote & Playstation Move*

Around September 2010, Playstation came to the market with an almost equal controller, called the Playstation Move. On the left-hand side of Figure 1, the controller of the Wii is depicted and on the right-hand side the Playstation Move controller is shown.

Recently, Microsoft released its Kinect controller. The Kinect is a novelty in the gaming industry. For the first time, a gaming console device detects complete body movements.


*Figure 2: Kinect*

## A. Problem Statement

We observe that there are a lot of possible ways to type in text. However, in the gaming console industry there seems to be a lack of good input devices. Today almost every device is connected with the Internet. In the near future even refrigerators will be connected with the Internet. Gaming console have the option to connect to the Internet, still the way of typing a word is really difficult and time consuming.

The problem is that in the gaming console world there is no good input device for inputting text. Most gaming consoles handle this problem by showing a virtual on-screen keyboard. Figure 3 shows an example of how this look likes on the Wii console system [12].


*Figure 3: Wii Keyboard*

With the Wii keyboard, the user points to the letter they want to type. In case the user wants to type a sentence this will take a long time.

The problem is that there is no device in which the user can type fast sentences or even words without having to hold a device or gaming console.

SpeeG is a very efficient way to input information by using voice recognition. Speech recognition is not always 100% correct. A lot of external factors can have a negative influence on speech recognition. Further, negative internal factors like noise of the hardware (e.g. microphone) can influence the result. Further, bad algorithms need a lot of computing power to recognise the speech which has a negative influence.

## B. Used Technolgies

SpeeG is a composition of three components. The first component that we used for SpeeG is the 3D camera of the Kinect. The second component is the Sphinx speech recognition system. The last component is Dasher on which the user interface is based.

The 3D camera sends a bundle of infrared light to the user who is standing in front of the Kinect. By registering the dots that spread in the area (see figure 4), the Kinect detects the presence of people, and more specifically, hand gestures like a wave movement or a push operation.

*Figure 4: Kinect IR dots*

For the navigation in Dasher, both the left and right hand are used. The left hand manages the initial start (push) operation and the speed of the movement, while the right hand guides the path.

The second component, Dasher, is used as the graphical interface of SpeeG. Dasher has the function of a zooming interface as shown in Figure 5.

The user starts with Dasher in the centre of the screen. At the right-hand side of the screen, the alphabet is placed in rectangles. Based on the English vocabulary, some letters have a higher probability to become for example the second letter when the first letter has already been chosen. In that case, these letters are placed in larger rectangles. So the rectangles with the letters ‘x’ or ‘y’ are smaller

than the rectangles with the letters ‘a’ or ‘b’. Further travel into the letter rectangles allows new rectangles to appear at the right-hand side of the screen, creating a continuous stream of words.

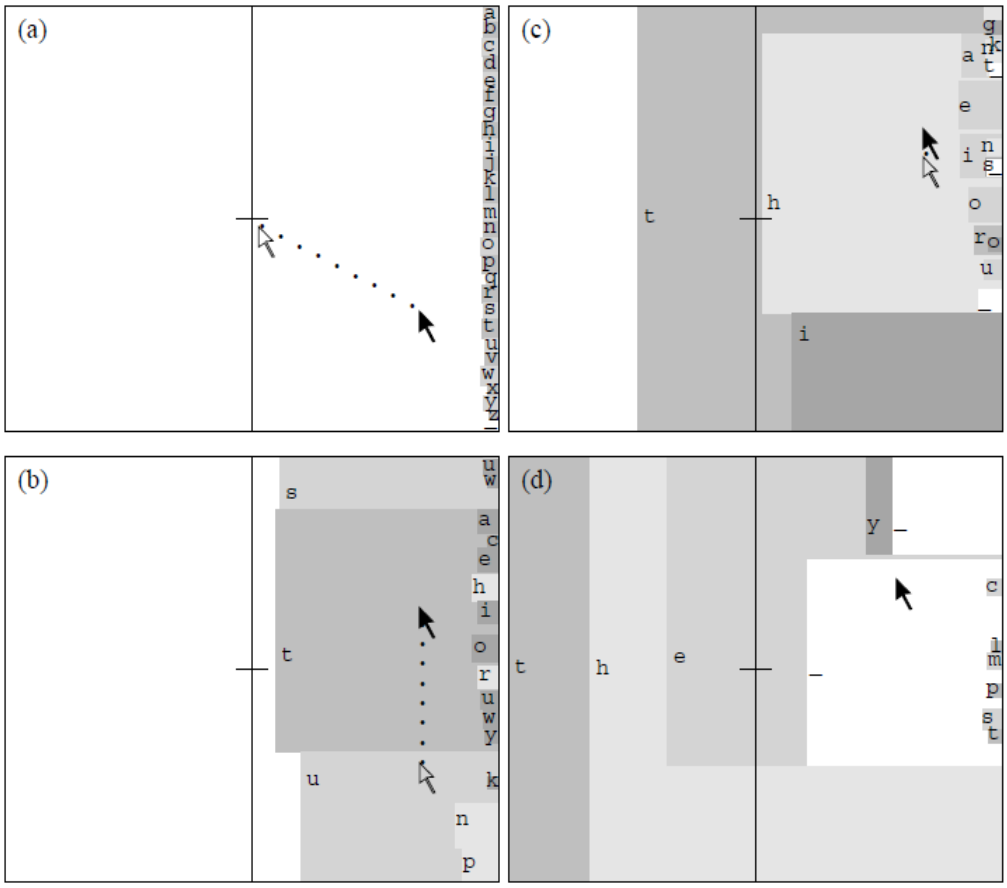



*Figure 5: Dasher Zooming interface*

The input to Dasher are words delivered by the SphinX speech recognition system. Sphinx is an open source speech recogniser written in Java or C. SpeeG has been realised based on the Java version of Sphinx.

SpeeG could be adapted for use in a home environment. Fast and accurate results are mandatory for use in set-top boxes.

## C. Motivation

SpeeG is a completely new idea. The uniqueness of SpeeG is that its real-time interaction.

In the first place, as we already discussed that there are many issues in text input. The motivation is to drastically speed up the text input for set-top boxes like gaming consoles.

To type in a single word on the gaming console is hard. To type in a sentence is too time consuming. Most of the time when the user has to type something, it is their name or a title to save the game.

At the moment, speech recognition is not 100% accurate. Most speech recognition systems use a dictionary; still those systems are not accurate enough. Creating a list of possible words that a speech recognition system can recognise is feasible. Today, there are no systems that use this sort of implementation where the user receives a list of possible words.

We will use the combination of Dasher and a speech recognition system. Dasher is a technology that was released in 2002. Dasher is created for people who want to type in a text fast. Further when people have to deal with medical problems, the eye-tracking option can help them type in text correctly.

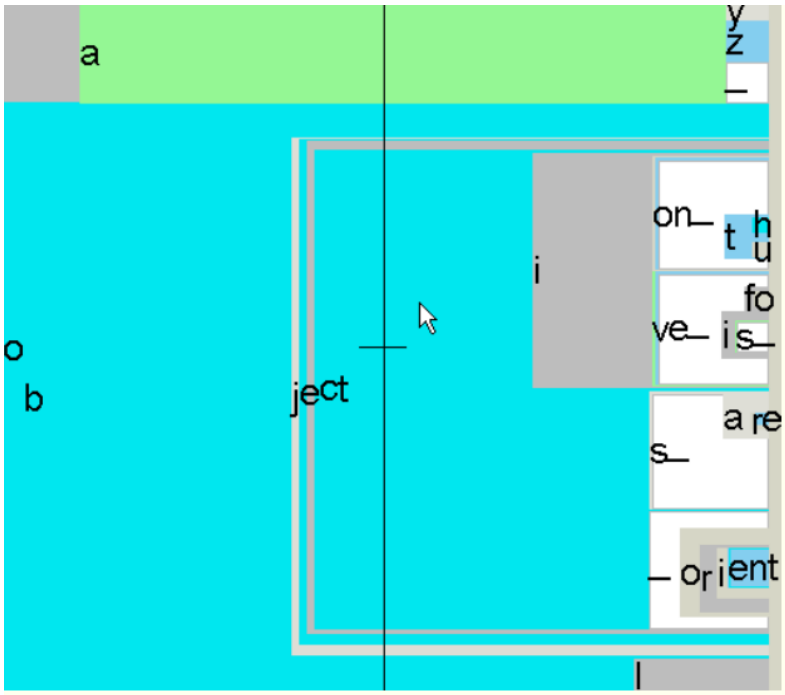



*Figure 6: Dasher*

Figure 6 shows the state of the Dasher interface while the user is writing the word 'objection'. Alternative words that could easily be written at this point include 'objective', 'objects_', and 'object_oriented'.

SpeeG is developed in combination with speed, speech recognition and selecting the correct words.

SpeeG works around the disadvantages of typing a text and the speech recognition issues. SpeeG brings a new way of text typing.

This thesis aims at creating an effective and useful application that solves the speed problems of typing on a gaming console.

Another objective is to have a solution for the problems that occur when the speech recognition is not 100% accurate.

The application is a combination of existing technologies that are adapted so they can all work together properly. Further, all the source code is open source, so others can continue to work on this project.

## D. Contributions

SpeeG is completely unique. No other system has the same features as SpeeG. SpeeG offers a unique combination between Kinect 3D cameras, speech recognition and Dasher. The reason why we targeted the 3D cameras is because it allows users to speak freely. The user does not have to obtain or hold a controller.

SpeeG is developed in a way that it is easy to replace the speech recognition or the input (Kinect). Dasher cannot be replaced because it is forming the core of SpeeG.

The most important contribution is the implementation of the idea. The experience is to have a real-time hand-assisted voice recognition. Further SpeeG can easily be modified for future work.

Several microphones were evaluated and compared. An overview will show the microphone that delivers the best result for the speech recognition system. Speech recognition systems are really weak for incoming noise. Hence, not all microphones are suitable.

Eventually, SpeeG has been tested by a selection of volunteers. From this user evaluation we derived some initial feedback and results.

## E. Structure of this thesis

Chapter 1: Introduction

In the introduction, an explanation of the problem is given, followed by the motivation why SpeeG is a suitable solution.

Chapter 2: Related work

The second chapter provides information on which solutions exist as an alternative for the application developed in this thesis. The chapter includes a report about the research of different speech recognition systems.

Chapter 3: SpeeG - Implementation

The fourth chapter gives an overview of the technology used for the implementation.

Chapter 4: User Evaluation

The one but last chapter contains the user evaluation of this thesis and the related results.

Chapter 5: Conclusion

The lessons learned from this thesis are discussed.

Chapter 6: Future work

Provides some recommendations for future work.

# Chapter 2: Related Work

In this chapter, state-of-the-art available text input methods will be discussed in more detail. The first subsection describes the most significant speech recognition systems, such as Dragon. Dragon recently became one of the major players, after a series of acquisitions of competing systems (e.g. IBM ViaVoice and Lernaut & Hauspie Recognition). Further, the Microsoft Speech SDK and the Sphinx recognition engine will also be described. The second subsection contains alternative text input mechanisms, such as on-screen keyboards, the T9 input method, Swype and the 8Pen. The third subsection discusses different modalities that are used in Human-Computer interaction which are related to this thesis.

## A. Speech Recognition Software

Today, there are a couple of major players on the market in the speech recognition software industry. In this section the most important systems will be discussed. The Dragon Naturally Speaking engine, licensed by Nuance. The Microsoft Speech SDK, which is also bundled with the Kinect SDK. and the open-source Sphinx system which is still under active development at Carnegie Mellon University.

### 1. Dragon Naturally Speaking

With a claimed accuracy of 99%, Dragon Naturally Speaking positions itself as one of the best speech recognition software systems on the market. By means of a set of mergers and acquisitions of smaller players, like ViaVoice of IBM (2003) [8], they have been able to gain the largest market share. Further, they use technology of former Lernaut & Hauspie (2001)[5]. Dragon Naturally Speaking is also used in other industries, like the medical world or the automotive industry [4]. Some cars – like Ford – are already equipped with the Dragon Naturally Speaking software. The API of Dragon Naturally Speaking SDK used to be open source, but is now only available via a developer licence .


*Figure 7: Dragon*

## 2. Microsoft Speech Recognition

Microsoft Speech Recognition SDK is a software development kit that allows the user to build speech engines and applications for Microsoft Windows. The SDK can be used a C++ and .Net environment.

Reviews show that Windows Speech has a good accuracy. The Windows Speech Recognition system is able to learn from previous mistakes, thus the more the user speaks, the higher the accuracy. Though, as a developer, the SDK is very limiting. There is no control over the training parameters and requesting interesting information, such as a list of alternative words is not supported. This is a big disadvantage for our setting. An additional disadvantage is that Microsoft Speech recognition is not cross platform.

## 3. Microsoft Kinect Voice Recognition

The Kinect leverages the Microsoft Speech SDK by providing an array of microphones to improve noise-cancellation. Sophisticated software, such as beam forming. is able to retrieve the location of the speaking user, which allows to focus on voice, and not the common in-house noises. The Kinect hardware performs best at a distance of 10 feet. This was decided after a thorough evaluation where Microsoft researchers went to about 250 houses to test with different positions for the microphone. Finally, they decided to place four microphones in a row, one on the left and three on the right.

Further, Microsoft researchers improved upon the recognition of dialects. The acoustic model of Kinect was constructed by analyzing hundreds of people from around the world, speaking English with notes from their own regional dialects.

For the moment the user can use the Microsoft's voice recognition system to give basic commands, such as to stop a game. In that case you just have to say 'Xbox Stop'. It is important to notify that the system only recognizes a pre-programmed and very limited vocabulary. These words are provided by the application developer.

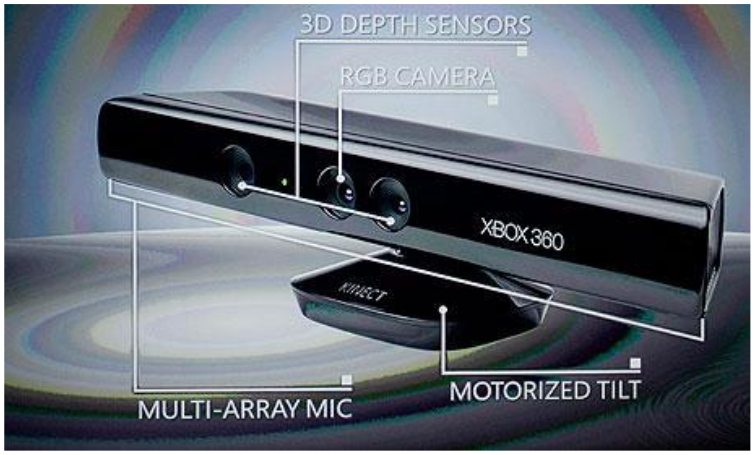



*Figure 8: Kinect Microphone*

## 4. Sphinx

Sphinx is speech recognition software that was developed at Carnegie Mellon University under DARPA funding. A few years ago, CMU released their software for free under a BSD license.

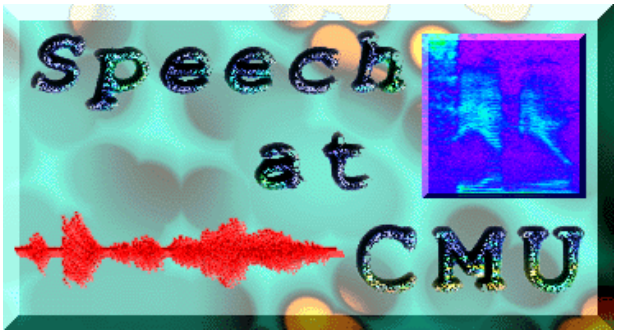



*Figure 9: CMU Speech*

Sphinx uses the hidden Markov model based system for large-vocabulary speaker-independent continuous speech recognition. The great difference between Dragon

Naturally Speaking and Sphinx is that Sphinx is open source and it does not require a user specific learning phase. After the installation, Dragon Naturally Speaking requires to learn the user's voice. Sphinx has an option that allows the user to train the voice data but it is not compulsory. Sphinx is implemented in two different program languages, C++ and Java, which is very developer friendly. Further, Sphinx allows the program to choose from a wide range of options, for example which language model has to be used. Additionally, PocketSphinx is also available for the cell phone or for computers that do not have enough processing power.

## 5. DICIT

DICIT project uses a distant-talking interface for interactive TV. DICIT system processes the user's input by speech and remote control. It was designed to understand natural language and command-and-control-style speech input. The most important feature of DICIT is the EPG (electronic program guide). It includes a program list and a filter for the criteria "channel", "genre", "day", and "time". Every program title that changed dynamically in the EPG can be spoken.

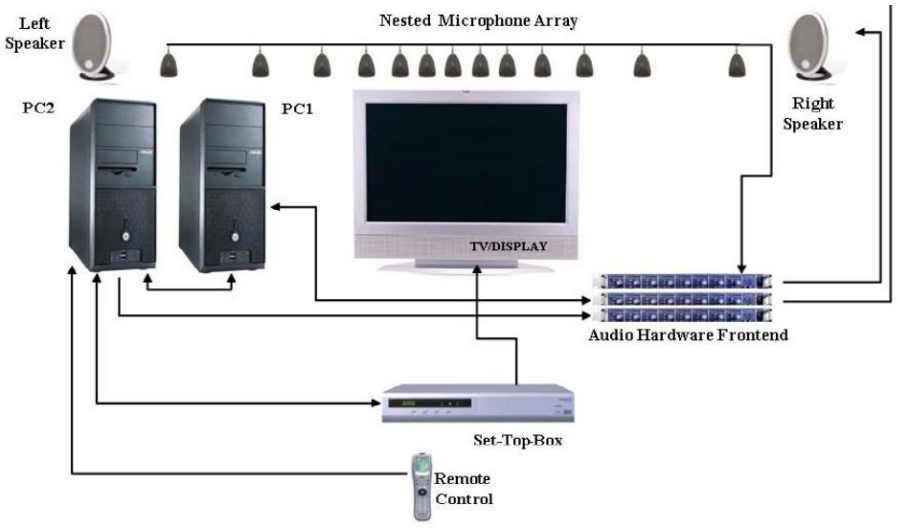



*Figure 10: DICIT Interface*

The speech recognizer works with natural language understanding. The NLU

module employs statistical models mapping recognized word chains into parameterized actions. The system is capable of learning about 100 different voice commands. The speaking distance goes up to five meters. The Remote by Amulet Device has a built-in microphone which transmits the speech signal. The DICIT system also doesn't require activation to open the microphone and start the recognition (i.e. no "push-to-talk" required). The DICIT system is however not applicable to our approach where we would like to provide a completely open and free input model [6], without the use of a limited (targetted) vocabulary.

## B. Alternative Input mechanisms

The way of typing a text hasn't changed much during the years. Begin 1900 the typewriter was invented and based on that the keyboard was developed. The keyboard layout is modelled exactly the same as the typewriter layout. The keyboard is currently still one of the most efficient text-input mechanisms today. However, a keyboard has its disadvantages. Small devices such as cell phones do not have sufficient size to lay down all the keyboard keys. Additionally, people do not want to clutter their living room with a keyboard for each device. Therefore, several new sorts of input typing devices were brought on the market. An example of this is the cell phones where people can type SMS's. More experimental approaches allow the user to stare with his eyes to indicate the letters. This was the main drive force for the development of Dasher. In this topic we'll discuss the alternative input mechanisms in more detail.

Although many controls are present in the living room today, the introduction of a more optimised input technology such as the T9 based keyboard has been an important event and we hope to vastly improve on this technology

### 1. On-Screen keyboards

The text-input on a Wii console is handled via an on-screen virtual keyboard. The user has to direct his Wii remote to the letter he wants to type and then press a button. Writing a sentence using this system is very inefficient and frustrating.

However, currently, this seems to be the best option for these kinds of devices. Similar techniques are used for other state-of-the-art game consoles, such as the PlayStation 3 and Xbox 360.


*Figure 11:Wii Mote & Keyboard*

## 2. T9 Input Method

Another way of inputting text is via a reduced keyboard layout, such as employed on typical cell phones. Although this method is more space-efficient, the set of only 9 buttons of the alphabet on a cell phone makes typing difficult. Getting the letter 'z' for example, requires to push not less than 4 times on the '9' button.

An improvement over this method is called T9. It was developed to minimize the effort of cell phone keyboard by guessing which letter is intended when a user hits a single key. It does this by combining all previous letters to form a viable (single word) combination. This input method also has a look ahead function: at any time, the T9 method provides the word the user is most likely trying to enter and provides word-completion [9].


*Figure 12: Standard Cell Phone*

Unfortunately, there are fewer keys on the cell phone keyboard than there are letters in the English alphabet. Further, there are words that have different spellings but the same input sequence. The words 'me' and 'of' for example have the same input sequence '63'. For this problem the user is required to use a third button. It even can be worse, 'home', 'good', 'gone', 'hood', 'hoof', 'hone' and 'goof' all have the input sequence '4663'. If the * key is used as the "next match" button, then the user will have to type in `4663******' to type in the word `goof', for a total of 10 buttons the user could have just as well used the default cell phone input method.

The biggest problem of the T9 input method is that it is sometimes even slower than the basic input method on a cell phone. The overuse of typing on a cell phone can lead to chronic muscle problems.

## 3. Swype

A novel way to input text on a screen is by means of Swype. Swype allows the user to draw a smooth continuous line on the screen, running through the letters that have to be typed. In the example underneath, the word "quick" was constructed. As you can see, the line also ticks off a whole bunch of other letters on the path, but that doesn't confuse the system. The build-in intelligence doesn't require a high accuracy. Because of this 'smart system', it is possible to input text

very rapidly. Tests show that a user can get speeds of more than 40 words per minute.


*Figure 13: Swype*

However, a disadvantage of this technology is that a touch screen is essential. It might be interesting to apply this to thin air gestures (e.g. using 3D cameras) although we argue that the acceleration of the hand cannot be measured with 100% accuracy. Therefore, it would be difficult to slide over the letters.

## 4. 8Pen

The 8Pen allows the replacement of the conventional keyboard on all devices capable of detecting gestures. In particular mobile phones with touch screens, digital cameras, modern remote controls or game controllers, including 3D cameras such as the Kinect. Its advantage lies mainly in the fact that it is possible to input text faster than using conventional layouts on small devices [3].

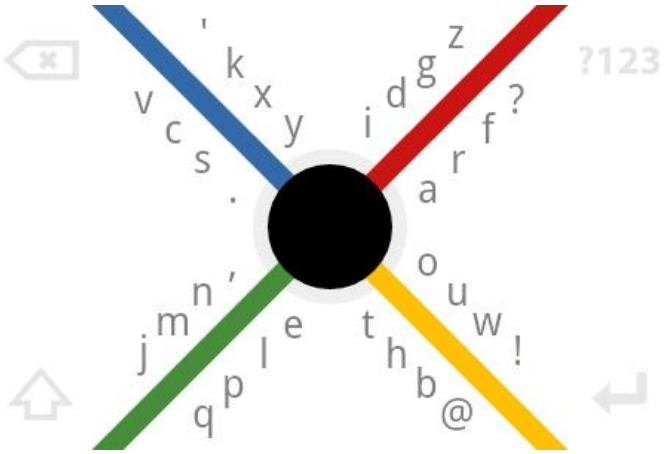



*Figure 14: 8Pen Interface*

The program is free for the Google Android device. The concept of the 8Pen is similar to the concept of the Swype for "drawing", but for the 8Pen there is a learning curve. The more the user practices, the faster the words are "typed". There is a possibility to learn the 8Pen programmable "gesture". Common words can be bind with a custom gesture. The speed that the 8Pen achieves is between 20 or 30 words per minute. Ceteris paribus, this is less than Swype.

## 5. Dasher

Dasher already exists for some years now (2000). The basic idea is to provide a zooming interface where users can construct words by zooming in on the intended letter at the right hand side of the screen. The graph underneath depicts the collection of letters as seen on a Dasher interface.

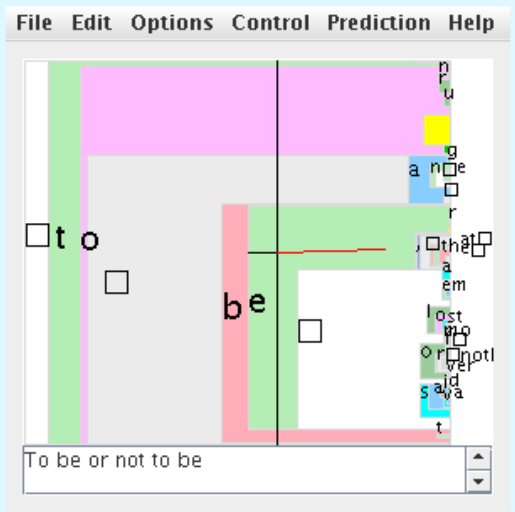



*Figure 15: Dasher*

The interface is developed in such a way that Dasher uses a language model to determine how much of the world is devoted to each piece of text. Probable pieces are given more space. The reason is that it is easier to select if the chunks of letters are larger.

Further, it is easy to train the model on any writing style. Any type of pointing device can also control dasher.

For describing this more in detail we use the example of entering the word 'the' in Dasher. In Figure 6, the start up of Dasher is shown with an alphabet of 27 characters displayed alphabetically in a column. The '_' symbol represents a space. The user writes the first letter by making a gesture towards the letter's rectangle. The trails show the user moving the mouse towards the letter 't'.

The display zooms towards the letter (figure 6.b) the user points. The rectangle of 't' gets larger and new characters inside this 't' rectangle appears. The possible letter combinations of 't' are 'ta', 'tb'…, 'tz'. The heights of the rectangle correspond to the probabilities of the string. In the English language 'ta' is more probable than the combination 'tb'. The software therefore allows easy travel to the combination 'th' (figure 6.c), and then to 'the' (figure 6.d).

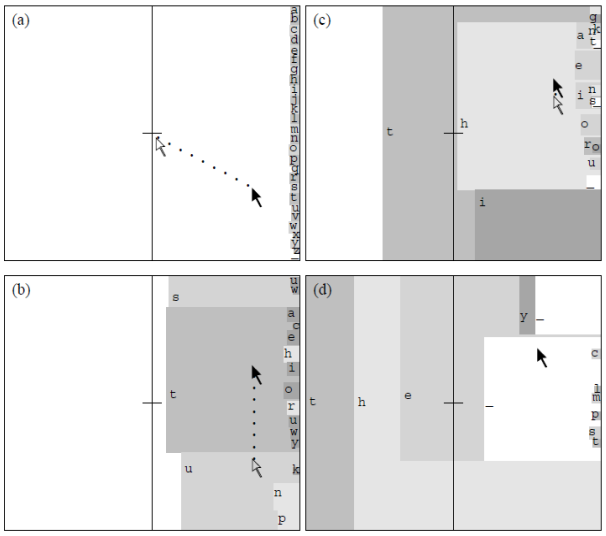



*Figure 16: Dasher Interface*

Novice users can quickly achieve speeds of over 20 words per minute [50].

## 6. #Dasher

The language modelling system in #Dasher could provide the basis of a language model to allow speech recognition of source code by providing the recognition system with the information it needs to disambiguate utterances.

Due to the high navigation and editing rate in software development it seems unlikely that a purely voiced based system would be practical, however in combination with a pointing based navigation system it might be possible to dramatically reduce the amount of keyboard usage necessary [22].

## 7. Speech Dasher

Speech Dasher is the combination of speech with Dasher in a two-step approach. First, the user speaks texts and then, when done, the 'recognise' button needs to be pressed. After the offline (non real-time) speech recognition, Dasher builds its dictionary on the recognised text. Users then navigate in the same way as in the original Dasher.

For the speech recognition Speech Dasher uses the PocketSphinx engine [50]. Users can choose between a US or UK English acoustic model to improve recognition rates. The trigram LM is trained on newswire text.

Tests show that people that use Speech Dasher can reach up to 40 words per minute. Even sentences that contain recognition errors can be written at 30 words per minute.

A disadvantage is that the developed software has not been released open-source (yet). This means that it is hard to convert the two-step input model to the continuum where SpeeG is based on.

Thus there are two reasons why we cannot reuse the existing Speech Dasher implementation for our SpeeG idea: Speech Dasher is not open-and is not developed for continuous input. We highly customized the Dasher implementation for our purposes as it was not built for continuous input too. More on this in Chapter 3: Implementation.

| Method | Words per minute (wpm) |
|---|---|
| **T9** | 45 |
| **8Pen** | 25 |
| **Swype** | 40 |
| **Keyboard** | 60 |
| **Dasher** | 20 |
| **Speech Dasher** | 35 |

## C. Human-Computer Interaction

Human-Computer Interaction defines the way the user interacts with his device. There are many ways to interact with a device. The most famous ones we all use daily are the mouse and keyboard. Other technologies, such as touch screens, are

also viable interaction methods. Other common devices are 3D cameras (e.g. Kinect) and eye tracking devices. In this part we discuss the possible interaction modalities related to our SpeeG interface.

## 1. Eye-tracking

Currently eye-tracking devices are tested to synergize with Dasher in order to enable hand-free text input. Think about the possibilities. This system would enable people with limited hand movement to actively engage in text writing activities. The working is simple. To write a string of text in Dasher, you focus your vision on the letters you would like to appear. A related technique that already exists is the one where the user has to consciously direct their gaze toward action targets. The problem is that the Eye-tracking Dasher uses extra, specialised hardware for tracking eyes.

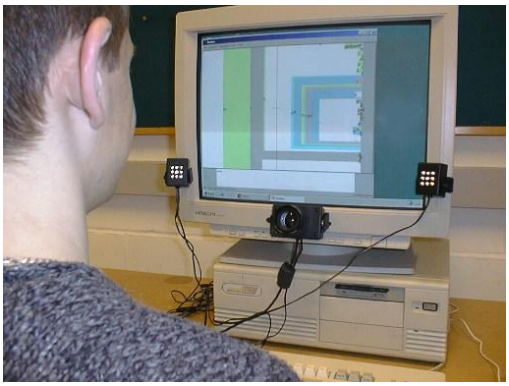


*Figure 17: Eye Tracking*

## 2. Touch Gestures

Touch is a new way to interact with a system. Tablets and some models of cell phone use this technology already. By touching the screen, commands are executed. Users can open a program by touching an icon or typing text on the screen (virtual keyboard). Today multi-touch gestures such as zooming objects on the tablet and turning objects are used. The only disadvantage is that the consumer price of a medium-sized touch screens is still quite high.

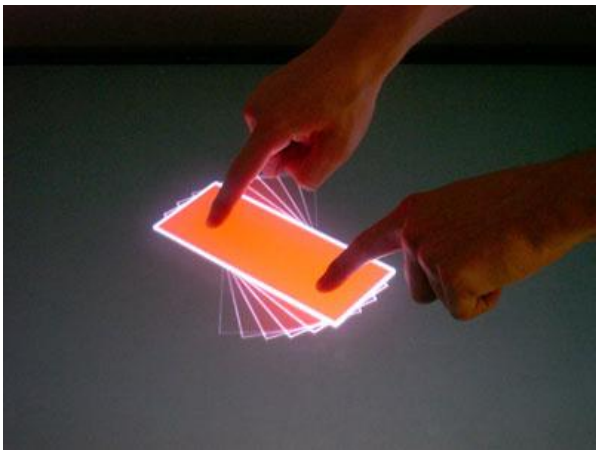


*Figure 18:Touch Gesture*

## 3. Hand Recognition

Hand Recognition is a technology in which hands are detected by a camera. Hand recognition means the initial recognition of a hand and then keeping track of it. There are many techniques to recognize hands. One of the techniques used to detect hands is the Hausdorff technique which calculates the distance between two sets of points [20].

Another technique uses a particle filter which tracks the hands in time, based on colour and motion cues. No initialization phase is necessary, even when there is failure. The hand segmentation is based on Gaussian Mixture Model and refined using a combination of spatial information [18].

*Figure 19: Hand Recognition*

Research is done to apply hand gestures in video games. The recognition should be in real-time response and systems should be used in unconstrained environments. The algorithm works in three main steps: hand segmentation, hand tracking and gesture recognition from hand features [19].

## 4. 3D Cameras

While webcam based approaches use a variety of complex analysis 3D cameras, like used in the Kinect, can recognize hands and the complete body with much greater ease and precision.

The Microsoft Kinect is a device that delivers the ultimate gaming experience. It has a camera and suitable software with which the Xbox360 of Microsoft can be controlled. The user has to carry a controller around. Movements, narration and the recognition of objects drive the Kinect.

The Kinect uses two cameras for tracking movement. There is one infrared (IR) camera for the depth detection and one for RGB colors using the 640x480 resolution. There is also a projector, an IR transmitting diode, which projects a well-defined pattern that can be recognised by the IR camera.. With this information, a 'depth field' can be created which is interesting to distinguish body parts, such as hands. For the gaming experience, the user stands in front of the

Kinect. In that case, he is the most nearby object. His shape will appear in brighter colours, like yellow or green, while the room itself, the sofa and the walls, will appear in shades of grey.

Alternative technologies, such as time-of-flight cameras can also be used to easily recognise body parts. SoftKinectic for example is the leading developer of time-of-flight 3D gesture recognition CMOS sensors and cameras. However, the low monetary cost of the Kinect is extremely interesting for our purpose.

The Kinect device is delivered with a PC compatible USB connector. However, by the start of this project there was no official Kinect SDK. Thankfully, the open-source community quickly developed their own drivers which can be used together with the OpenNI NITE package (from PrimeSense, the company that licenced it's technology to Microsoft).

Other drivers, such as the Code Laboratories drivers are also available online. Their SDK is programmed in Java. We evaluated this SDK but concluded that it did not provide the required algorithms (e.g for capturing push operations). OpenNI NITE does provide these algorithms and therefore SpeeG uses the OpenNI NITE.

The OpenNI NITE package is able to identify a person after he performed a recognition pose for a few seconds. In Figure 20 an example is showed after a successfully person detection.

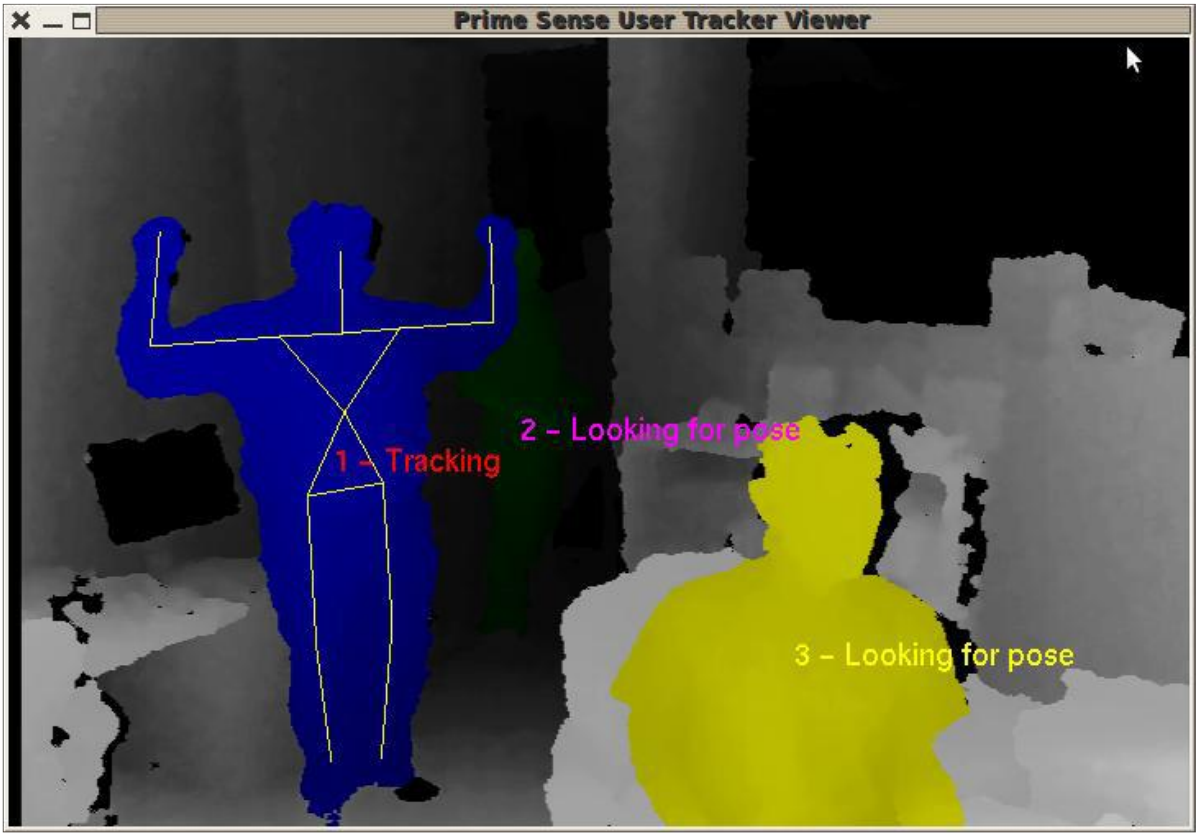



*Figure 20: Body Recognition Kinect*

After testing the algorithm, we discovered that the result of those algorithms did not require adaptation. Many other implementations were made on the Kinect OpenNI NITE drivers. In figure 22 for example, the Kinect is combined with the game Street Fighter. Street Fighter IV can now be played using simple motion based gestures assigned to key bindings using FAAST [15].

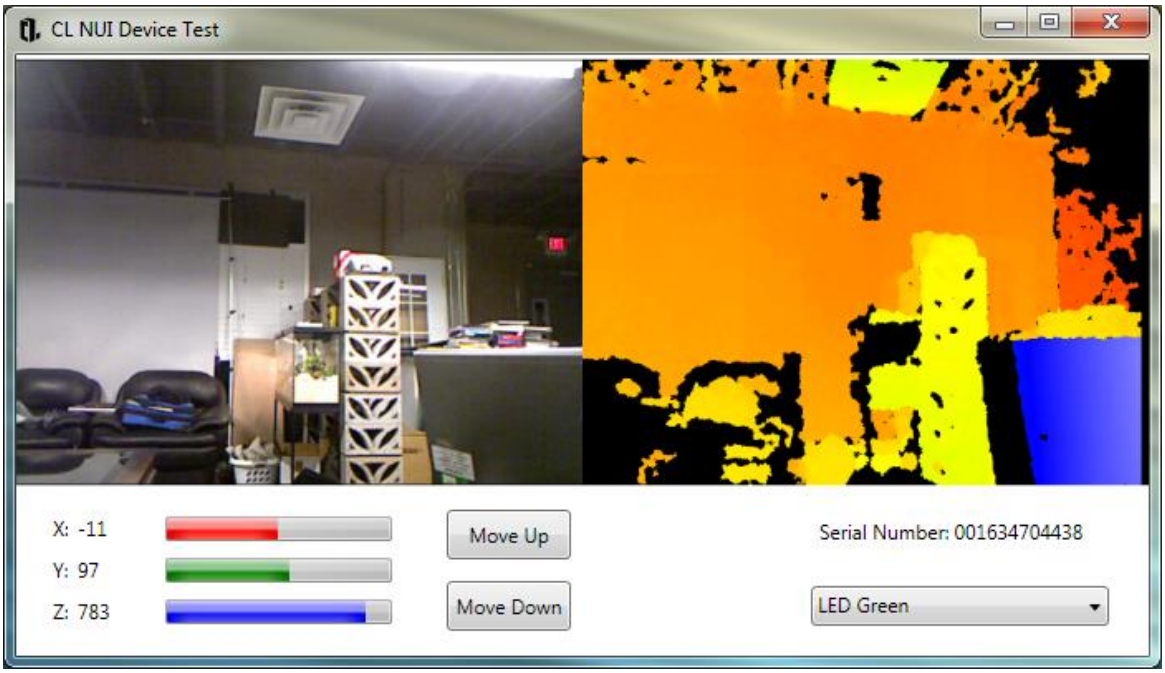



*Figure 21: Code Laboratories Example*

Another example is that the user can learn to recognize objects. In figure 22 the user pronounces the word 'dog' and Kinect learns that this object is a dog. Normally, if the user removes the dog and then places it in front of the Kinect again, the Kinect should recognize that the object is a dog. In both cases the Kinect device is used.


*Figure 22:Kinect (StreetFigther & Recognize Objects)*

## D. Speech Recognition Hardware

To achieve good results with speech recognition engines, the selection of the employed hardware is very important. The most important parts are the sound card and the microphone. A sound card that is not appropriate adds a lot of noise, which results in a bad quality. The type of microphone also strongly influences the result of the speech recognition in a similar way.

### 1. Sound Cards

Speech requires a low bandwidth. A normal sound card of 16 bit is enough to option a good quality. Of course, the conversion from analog to digital should be as clean as possible. Electrical noise can influence this signal drastically. This noise can be caused by monitors, PCI slots, hard disks, etc. Still, the audible noise coming from computer fans, squeaking chairs or heavy breathing have a stronger influence on the results than electric noise.

### 2. Microphones

There is not a predefined answer to which is the best microphone. Some microphones were developed to recognize a few intentional words, for example the Kinect Microphone. Others need to be able to recognize more complex word constructions and continuous sentences. The best microphone for speech recognition is without doubt to be the Sennheiser ME-3, but there are other microphones that give a similar result for the speech recognition.

*Figure 23: Sennheiser ME-3 Microphone*

USB microphones also exist and have the advantage that the process of conversion is outside of the computer so no electrical noise can influence the signal. Another advantage is that the correct voltage needed for the conversion is better than the one inside a computer.

Of course, other extern background noise like for example music, car engines, ... can influence the output of the system, so the speech recognition result in bad results [7].

Figure 24 gives an overview on the accuracy ratings using different microphone. This accuracy is test on the most popular speech recognition systems [16].

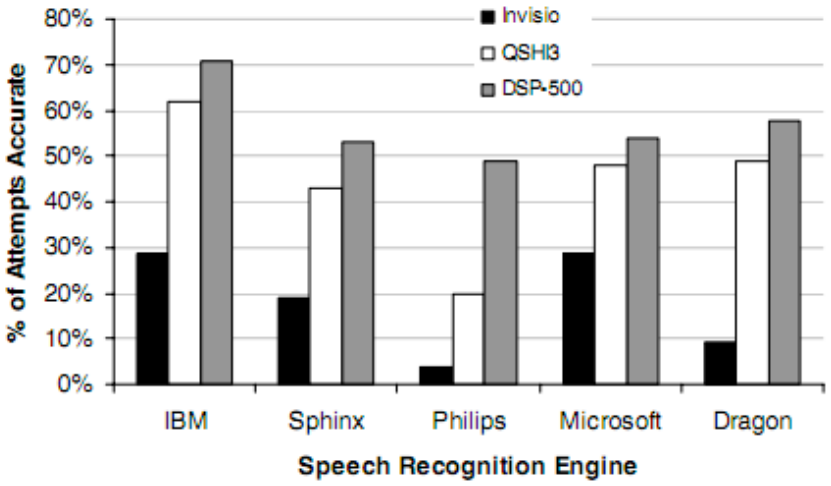



*Figure 24:Accuracy rate according to speech recognition engine and microphone*

# Chapter 3. Implementation

The implementation of SpeeG is based on a combination of different components. Figure 25 shows an overview of our architecture. We have four major parts that work with each other.

First, we begin with the user. He controls the text-input method by means of voice and gestures. Whenever a user is detected by the Kinect camera, our SpeeG prototype starts listening for input by enabling the Dasher (2) and the Sphinx speech recognition engine (3).

The user is now free to speak words, which will be recognised by the voice recogniser (4). Every word which is successfully recognised, will be send to the Dasher interface, together with a list of alternatives (5). Dasher is responsible for compiling the word and its alternatives into an easy selectable visual interface (6).

The process is finalised by letting the user point to the word(s) he wants to type. This is done by pointing using the right hand. With the left hand the user can control the speed of the selection When there are no more recognised words to be displayed, the SpeeG interface pauses to wait for new voice input.

The reason why our components are very decoupled is because of the asynchronous interaction. The user can choose to first speak his complete sentence and later select his intended words, or he can do it in the more optimised case where the selection and the speech is done in parallel.

There is also an option to start and stop the Dasher interface by performing the push gesture.

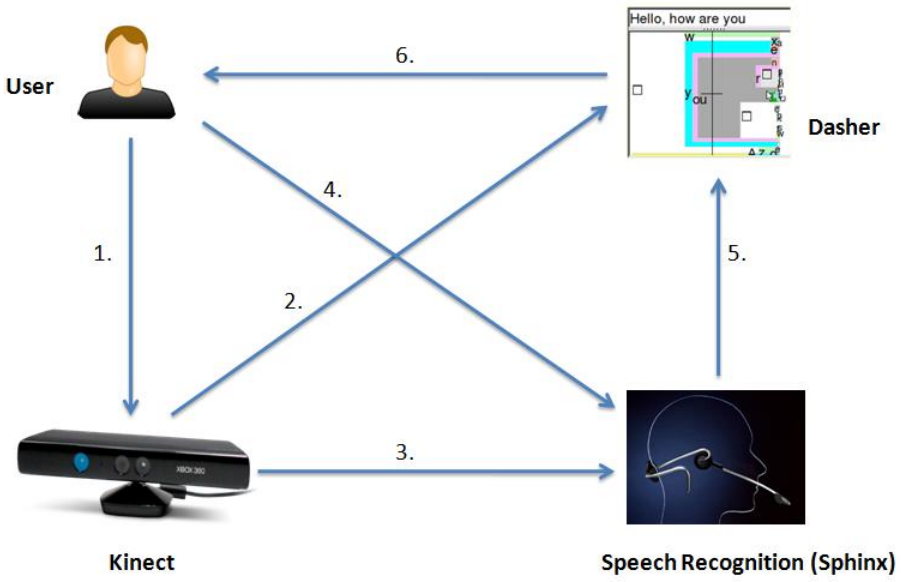



*Figure 25: Implementation Overview*

We will first explain how to communicate is handled between these systems. Further we continue in detail the major components of SpeeG

## E. Socket Link

Sockets are used for the communication between the several systems. As first we have the Kinect which is represent the server. The Kinect capture push operation made by the left. If this operation occurs a click event is send to Dasher and Sphinx. After that action the Kinect send hand coordinates to Dasher. Further the Kinect send also a speed variable to Dasher. In the meantime Sphinx send words that are recognised to Dasher.

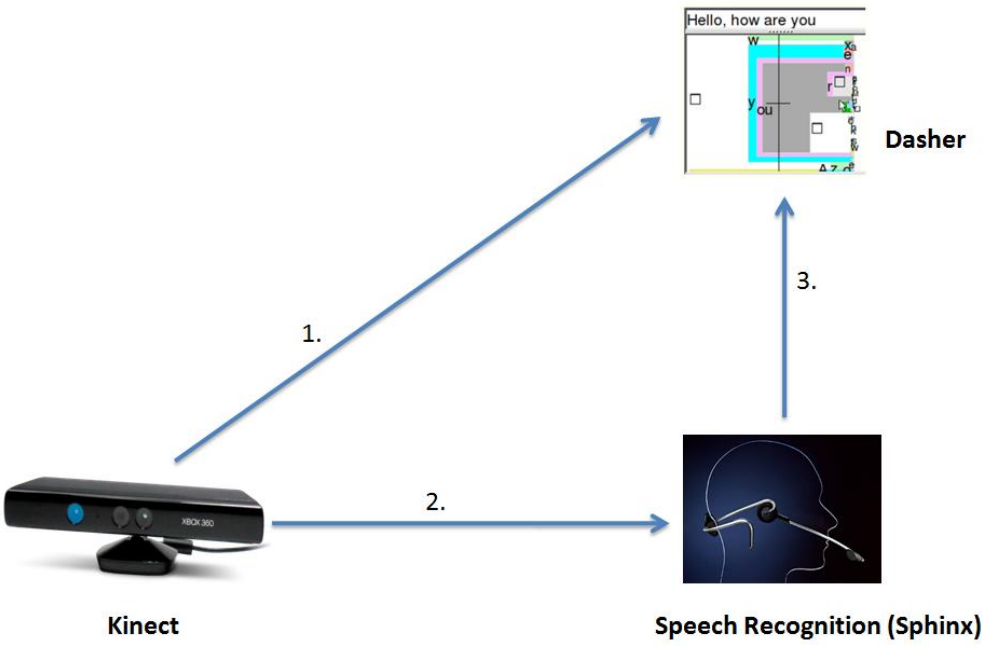



*Figure 26: Socket*

The socket strings for the communication between Kinect and Dasher for sending the coordinates is the following:

- /SPEEG/15/0.56/|
    - The 15 is the joint ID of the right hand
    - The 0.56 is the position of the right hand

The socket strings for the communication Kinect between Dasher for sending a click event is the following:

- /SPEEG/99/|
    - The 99 is the ID the click event

The socket strings for the communication Kinect between Dasher for sending the speed is the following:

- /SPEEG/19/60/|
    - The 19 is the ID for the speed
    - The 60 is the speed

The socket strings for the communication Kinect between Sphinx for sending a click event is the following:

- /SPEEG/99/|
    - The 99 is the ID the click event

The socket strings for the communication Sphinx between Dasher for sending the possible words is the following:

- /SPEEG/a/0.1/air/0.1/as/0.1/|
    - The ‘a’, ‘air’, ‘as’ are the recognized words

## F. Dasher

For Dashers interface is used in SpeeG. There are two implementations available, one in C and the other in JAVA. SpeeG uses the Java Dasher version which is called JDasher. Dasher is based on a text input. Dasher constructs cubes of letters. A cube is called a node Dasher. In figure 8 is an example given of the words 'name' and 'norm'. The reason that the word 'name' is larger than the other word. This is because the probability of the word 'name' is greater than the other word.

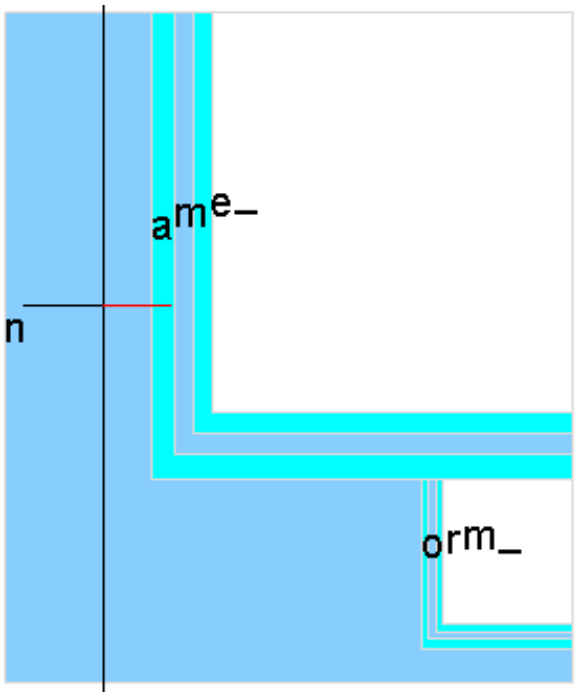



*Figure 27: Dasher Example*

The built of these nodes starts with the first letter of the words, which is 'n'. The letter 'n' is the first letter of the word 'name' and 'norm', so the probability is 100%. Next, the node is split up in two nodes. This is because the letters are not the same. The first node is the 'a' and the second node is 'o'. The nodes are ranged alphabetically. The node 'a' is much larger than node 'o'. This is because the probability that 'a' is the following letter is more than the probability that 'o' is the following letter. The sum of those nodes should be equal to 97%. In case this was 100%, the nodes, the letters, will overlap each other. Further, we continue

with the 3rd node, which contains the letter 'm'. The size of this is 97% of the previous node, so it has the maximum size. The 4th node contains the letter 'r'. The '_' character represents a space.

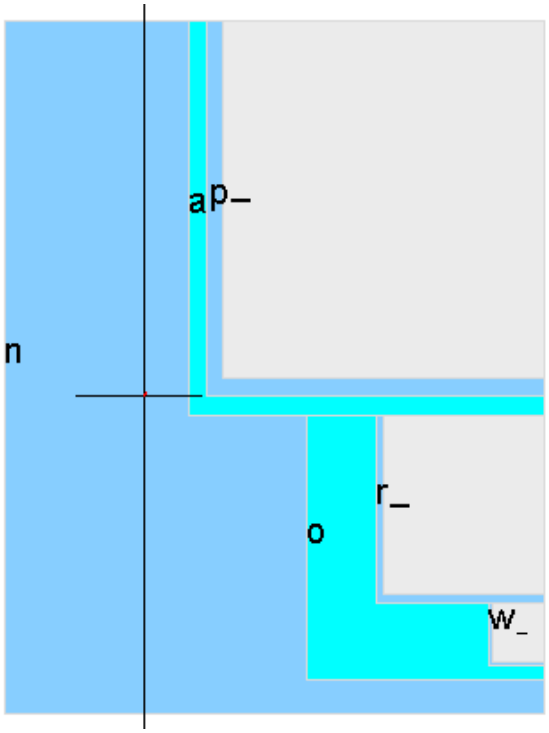



*Figure 28: Dasher Example*

## 1. Node

A Node is a cube which contains a character(s). A Node is not only one character but contains a bunch of children. In the original Dasher this children is the alphabet plus the space character. In the above example 'r' and 'w' are the children of 'o'. So 'n' is the parent of 'r' and 'a'. 'n' does not have parents. The heights of the blocks are described in the children. Every node has the alphabet children plus the space symbol. Most of the children has the height zero. Further a node also has colour and a number. The last one is not implemented in the original Dasher. The utility of the number is described later.

## G. Node != Character

In figure 29 an example is showed of the words ‘nah’, ‘noh’ and ‘zorro’. The first node is showed in figure 30. This are the letter ‘n’ and ‘z’ the rest of the letters of the alphabet is not showed because the size of the cubes is zero. The nodes of the letters ‘n’ and ‘z’ can have children. The children for ‘n’ are ‘a’ and ‘o’. The second node is showed in figure 31. The third node is the children of ‘zorro’ and this ‘o’, figure 32. The fourth node is the children of parent ‘a’ and with the grandparent ‘n’, and this is ‘h’, figure 33. The fifth node is the children of parent ‘o’ and with the grandparent ‘n’, and this is ‘h’, figure 34. The sixth node is the children of parent ‘o’ and with the grandparent ‘z’, figure 35. The seventh node is something special, it is a node but contains no character, figure 42. The last three nodes are also special, figure 39, 40 and 41. These red squares with the ‘_’ character represent the space character. The final result is showed in figure 29.

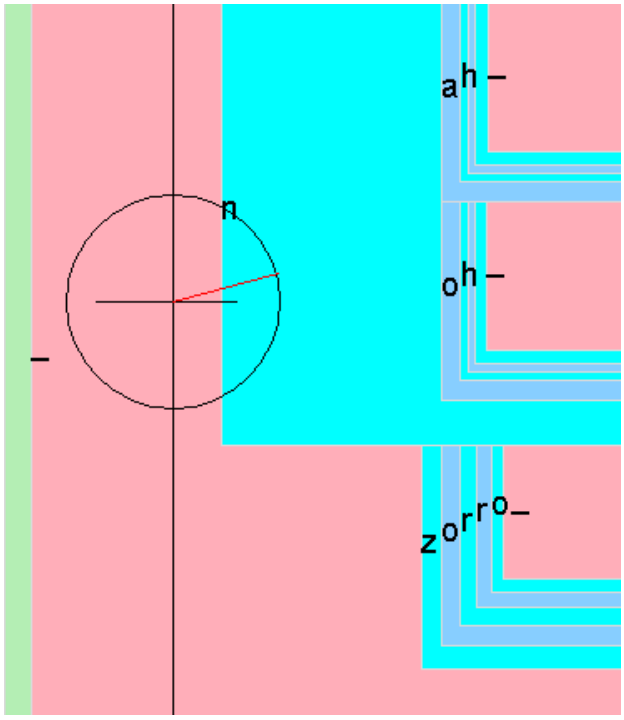



*Figure 29: Dasher 'nah' & 'noh' and 'zorro'*

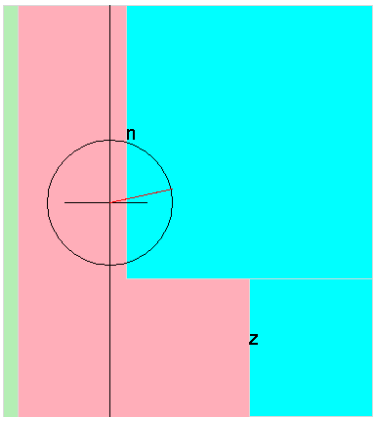



*Figure 30: 1st Node*

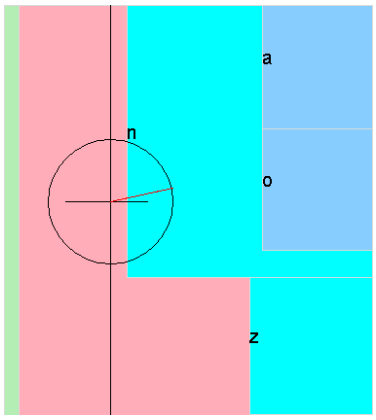



*Figure 31: 2nd Node*

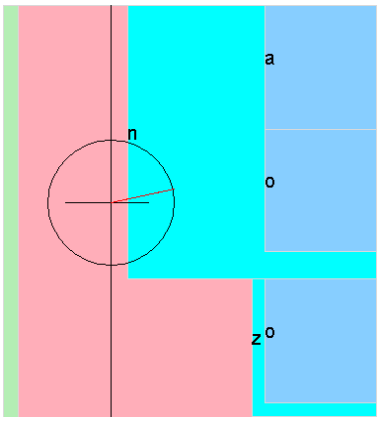



*Figure 32: 3rd Node*

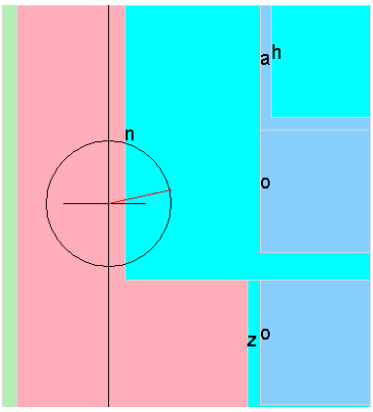



*Figure 33: 4th Node*

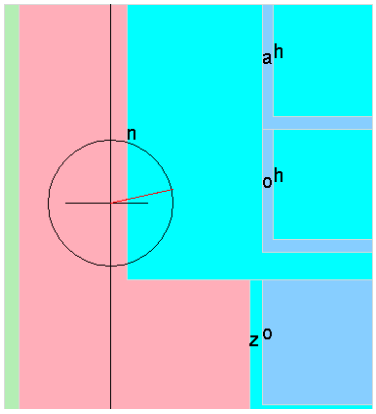



*Figure 34: 5th Node*

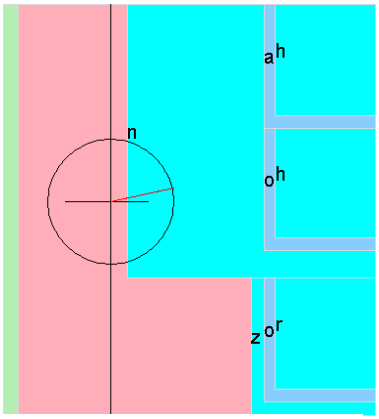



*Figure 35: 6th Node*

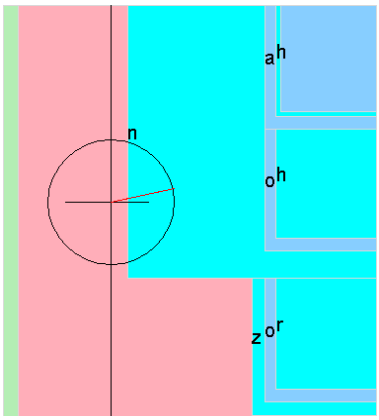



*Figure 36: 7th Node*

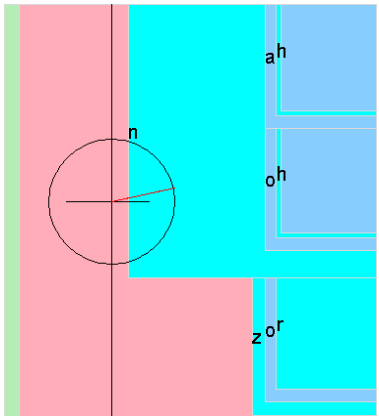



*Figure 37: 8th Node*

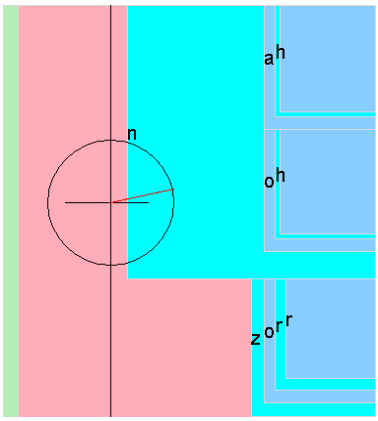



*Figure 38: 9th Node*

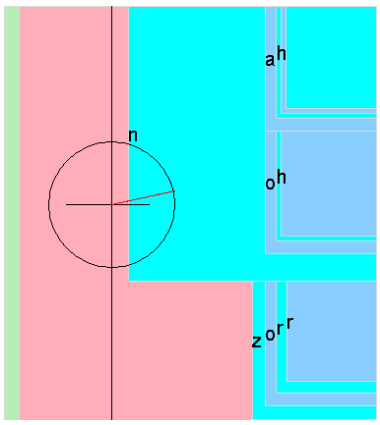



*Figure 39: 10th Node*

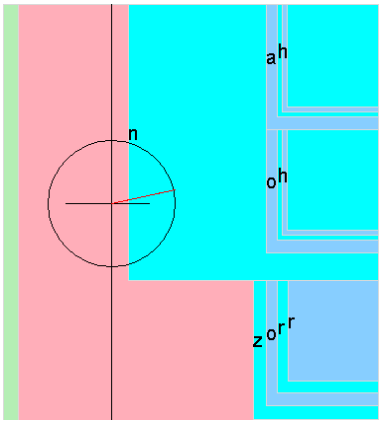



*Figure 40: 11th Node*

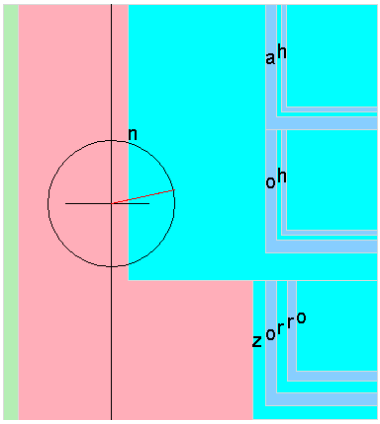



*Figure 41: 12th Node*

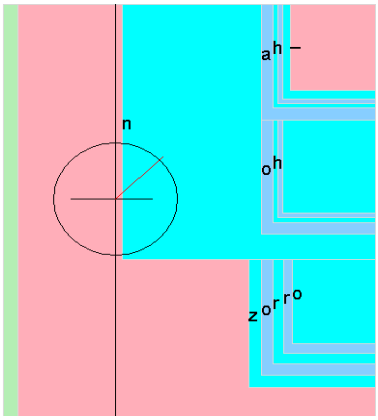



*Figure 42: 13th Node*

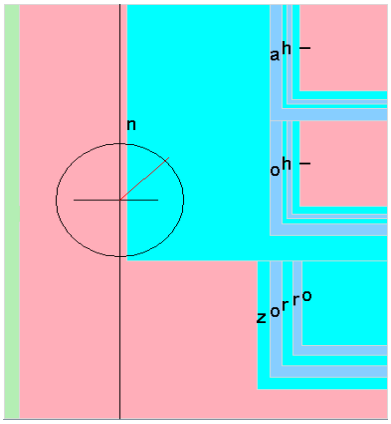



*Figure 43: 14th Node*

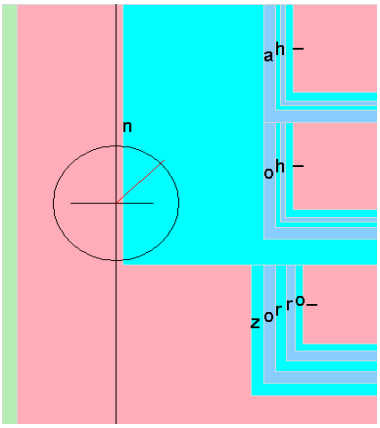



*Figure 44: 15th Node*

## 1. Words to Nodes in SpeeG

From the speech recognize program (Sphinx) we receive a string of words with probabilities. This string has standard layout /SPEEG/name/0.7/norm/0.3/|. String starts with a '/SPEEG/' then the words, again a '/' and finally the probability. If there are more words they will be attached the same way the previous words are done. At the end of the string a '|' is attached.

After the string is send to Dasher, the string is decomposed. After the decomposition, we have to create nodes of the words. The next code is to set that every word is in common which each other. This is done as initialisation. The

```
boolean commonWords[][] = new
boolean[amountOfPossibleWords][amountOfPossibleWords];
//Some words have the same node; this is stored here (when character are the
same of the words)

boolean commonWords2[][] = new
boolean[amountOfPossibleWords][amountOfPossibleWords];
//Same as commonWords but of the previous character
//commonWords ==> Some words have the same node; this is stored here (when
characters are the same of the words)

for(int m = 0; m<amountOfPossibleWords; m++){
      for(int h = 0; h<amountOfPossibleWords; h++){
            commonWords[m][h] = true;
      }
)
```

The next part of the code is to look which words have the same character. We start with the first position of the words. In the above example we have the words 'name' and 'norm'. The first character is 'n' in both words. We compare every words which each other. If they have the same letter in common and they had something in common before we set the variable 'commonWords[i][j]' and 'commonWords[j][i]' on true. If they had nothing in common before 'commonWords[i][i]' was set on true. When they are not the same 'commonWords[i][j]' and 'commonWords[i][j]' is set on false. If the words are treated we put them on a queue ('listTempWord.add(i)'). When the variable 'commonWords2[i][j]' is false and '!commonWords[j][j]' is true, the word has nothing to do with the other words anymore. The words are ordered alphabetically. The words 'hallo' and 'hello' will have the first letter in common. The second letter is not the same, so 'commonWords[j][j]' is set on true, this is done in the end. For the third letter they have the 'l' in common but because they 'commonWords[j][j]' is on true, this will be just queued and the other 'l' from 'hello' is also queued. This is done in the 'listTempWord'. The variable 'wordAlreadyInQueue[i]' is used if the word is not queued.

```
for(int i = 0; i < amountOfPossibleWords; i++){
      for(int j = 0; j<amountOfPossibleWords; j++){
            if(j==i) {

            }
```

```
            else if(Words[i].charAt(positionLetter) ==
                  Words[j].charAt(positionLetter) &&
                  positionLetter<biggestWord_ && commonWords[i][i] &&
                  commonWords[j][j] && commonWords2[i][j])
            {
                  commonWords[i][j] = true;
                  commonWords[j][i] = true;
                  if(!wordAllreadyInQueue[i]){
                      listTempWord.add(i);
                      wordAllreadyInQueue[i] = true;
                  }
            }
            else if(Words[j].charAt(positionLetter) !=
                    Words[i].charAt(positionLetter))
            {
               commonWords[i][j] = false;
               commonWords[j][i] = false;
            }
      }
...
```

At this moment there is a chance that not all the words in the queue. If the 'commonWords[i][i]' are true, the word has nothing in common with the other words, so we add them. When 'commonWords[i][i]' is not true and the rest is false, the 'commonWords[i][i]' is changed to true and we have to add the word in the queue ('listTempWord').

```
if(!wordAllreadyInQueue[i]){
      for(int l = 0; l<amountOfPossibleWords; l++){
            if(commonWords[i][i]){
                  listTempWord.add(i);
                  break;
            }
            else if(commonWords[i][l] && i != l){
                  break;
            }
            else{
                  checkNumber++;
            }
      }
      if(checkNumber==amountOfPossibleWords){
          commonWords[i][i] = true;

          listTempWord.add(i);
          wordAllreadyInQueue[i] = true;
      }
}
```

Next we have to calculate the size of the squares. We remove the words from the queue list from the variable 'listTempWord'. When the character is ' ' empty, we

push the probability where the space symbol is at his maximum. This is done by the function 'pushSpace'.

```
while (!listTempWord.isEmpty()) {
      amountOfWordsInCommon = 0;
      TempWord = listTempWord.remove();
      if(!wordsHandled[TempWord]){
            for(int i = 0; i < amountOfPossibleWords; i++){
                  if(Words[TempWord].charAt(positionLetter) == ' ' &&
!common[TempWord]){
                        pushSpace();
                        WordCounter++;
                        checked = true;
                        break;
                  }
            }
```

The first letter of the word is treated differently. The reason is that the first node represent all the first letters of the words.

Next we fetch the letter from the word and push this probability with the function 'pushProb'. This function is multiply with the probability of a letter and with the maximum size of a node which is 65536.

```
else if(commonWords[TempWord][i] && TempWord == i && FirsTime){
      letterOfTheWord = Words[i].charAt(positionLetter);
      pushProb(letterOfTheWord, i);
      WordCounter++;
      break;
}
```

After the first letter of every word is completly handled. We create a node from the values which are pushed. These values are hold in 'testProb2' array. We loop this array to see how many splits will create in the next node. When the value of that array is not 0, we raise the 'SplitingOfNode' with 1. For example in figure 45, for the first node, which represent the letter 'n' this is one. For the second node this is two, the letters 'a' and 'o'. Eventually 'WhichNode' is raised by one.

```
else if(firsTime && checked){
      addProbs.add(testProb2);
      HowManySplits++;
      int SplitingOfNode = 0;
      for(int o = 0; o<32; o++){
            if(testProb2[o] != 0){
```

```
                    SplitingOfNode++;
              }
        }
        if(SplitingOfNode>1){
              latestSPLIT = positionLetter;
        }
        splitNodesLocation[WhichNode+1] = SplitingOfNode;

}
```

A word which has nothing in common with other words is when the variable 'commonWords[i][i]' is true. The 'common[]' is to point that there is nothing in common in the previous letter. If a word has nothing in common with other words this is set on false.

```
else if(commonWords[TempWord][i] && TempWord == i && !common[TempWord]){
        tempProb2 = Probability[TempWord];
        letterOfTheWord = Words[TempWord].charAt(positionLetter);
        Probability[TempWord] = 0.97f;
        pushProb(letterOfTheWord, TempWord);
        Probability[TempWord] = tempProb2;
        checked = true;
        break;
}
```

For example, the word 'nah' and 'noh', figure 45, the first letter is the same, so 'common [] = {true, true}'. For the second letter the value for 'common' is changed to false and false. For the last letter the previous letter has nothing more in common so the 'common' values are false. 'common' is put on false if they have the previous letter in common but the next letter not anymore. From that moment the words is completely on his own. In figure 45 the 'common' is used. In figure 46 is 'common' removed from the code. The words do not know that they first have to split or not.

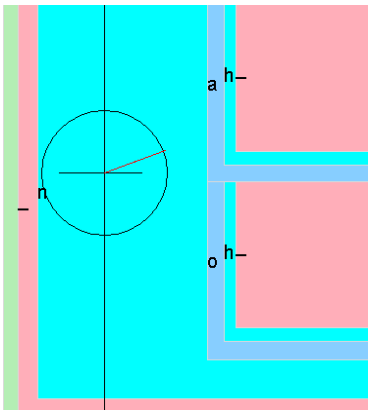



*Figure 33: Dasher 'nah' – 'noh'*

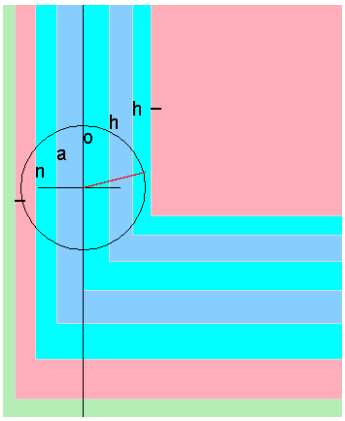



*Figure 34: Dasher 'nah' - 'noh'*

The next code is when a word has nothing in common. Still the 'common[TempWord]' is on true. So the previous position of that word has a letter in common with another word(s). The code changes the variable 'common[TempWord]' value to false and maybe also for the other words which have nothing in common anymore.

```
else if(commonWords[TempWord][i] != commonWords2[TempWord][i] && TempWord ==
i && common[TempWord] ){
      float tempProb3 = 0;
      common[TempWord] = true;
      checked = false;
      int checkCounter = 1;

      pushWords[amountOfWordsInCommon] = TempWord;

      amountOfWordsInCommon++;
      tempProb = tempProb + Probability[TempWord];
      WordCounter++;
      common[TempWord] = false;
      wordsHandled[TempWord] = true;
      for(int j = 0; j < amountOfPossibleWords; j++){
            if(commonWords2[TempWord][j] && !wordsHandled[j] &&
commonWords[TempWord][i] != commonWords2[TempWord][i]){
            pushWords[amountOfWordsInCommon] = j;
            tempProb = tempProb + Probability[j];
            amountOfWordsInCommon++;
            wordsHandled[j] = true;
            WordCounter++;
            if(commonWords[j][j]){
                  common[j] = false;
            }
      }
      else if(!commonWords[TempWord][j] && commonWords2[TempWord][j]){
            checkCounter++;
            tempProb3 = tempProb3 + Probability[j];
```

```
        }
}

for(int k = 0; k < amountOfWordsInCommon; k++){
        checked = true;
        tempProb2 = Probability[pushWords[k]];
        Probability[pushWords[k]] = (float)( ((0.97-tempProb3)/tempProb) *
tempProb2);
        letterOfTheWord = Words[pushWords[k]].charAt(positionLetter);
        pushProb(letterOfTheWord, pushWords[k]);
        Probability[pushWords[k]] = tempProb2;
}
break;
        }
```

The last part is when the words are in common. In figure 47 an example is showed that the words have the same letters of words but they split. On the thirth letter of the words ‘today’, ‘toka’, tokz’, ‘tozaz’ and ‘tozbz’ a split occur. The word ‘today’ has nothing in common after the second letter. For the words ‘toka’ and ‘tokz’ they still have a thirth letter in common. For the words ‘tozaz’ and ‘tozbz’ has also a thirth letter in common. The size of the letter ‘o’ is based on the words ‘today’, ‘toka’, tokz’, ‘tozaz’ and ‘tozbz’. The size of the letter ‘k’ is based on the words ‘toka’ and ‘tokz’. The letter ‘d’, ‘k’ and ‘z’ of the words ‘today’, ‘toka’, ‘tokz’, ‘tozaz’ and ‘tozbz’ are stored in the same node.

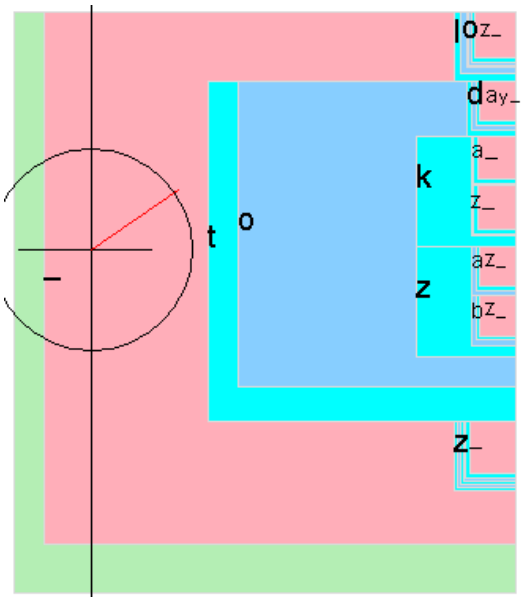



*Figure 35: Dasher 'lol' - 'loz' – ‘today’ – ‘toka’ – ‘tokz’ – ‘tozaz’ – ‘tozbz’ – ‘z’*

```
else if(commonWords[TempWord][i] && TempWord != i  && common[TempWord] &&
!wordsHandled[TempWord]) {
      pushWords2 = new int[amountOfPossibleWords][amountOfPossibleWords];
      sizePushWords = new int[amountOfPossibleWords];
      tempProb_ = new int[amountOfPossibleWords];
      float totalTempProb = 0;
      int totalProb = 1;
      common[TempWord] = true;
      checked = false;
      pushWords2[0][0] = TempWord;
      sizePushWords[0]++;
      tempProb_[0] += Probability[TempWord];

      amountOfWordsInCommon++;
      tempProb = tempProb + Probability[TempWord];
      wordsHandled[TempWord] = true;

      letterOfTheWord = Words[TempWord].charAt(positionLetter);

      int j = 0;
      int NextComp = 1;
      int counte = 1;
      for(; NextComp < amountOfPossibleWords; NextComp++){
            for(; j < amountOfPossibleWords; j++){
                  if(commonWords[TempWord][j] && TempWord!=j){
                        sizePushWords[NextComp-1]++;
                        pushWords2[NextComp-1][counte] = j;
                        tempProb_[sizePushWords[NextComp-1]] +=
Probability[j];
                        totalTempProb += Probability[j];
                        totalProb++;
                        wordsHandled[j] = true;
                        counte++;
                  }
                  else if(!commonWords[TempWord][j] &&
commonWords2[TempWord][j]){
                  TempWord = j;
                  pushWords2[NextComp][counte] = j;
                  tempProb_[NextComp] += Probability[j];
                  totalProb++;
                  totalTempProb += Probability[j];
                  sizePushWords[NextComp+1-1]++;
                  wordsHandled[j] = true;
                  counte++;
                  break;
                  }
            }
            if(j == amountOfPossibleWords){
                  break;
            }
      }
      int z = 0;
      for(int k = 0; k < NextComp; k++){
            for(int p = 0;p < sizePushWords[k]; p++){
                  tempProb2 = Probability[pushWords2[k][z]];
```

```
                    Probability[pushWords2[k][z]] =
(float)(0.97*(float)(sizePushWords[k])/(float)(totalProb)/(float)sizePushWor
ds[k]);
                    letterOfTheWord =
Words[pushWords2[k][z]].charAt(positionLetter);
                    pushProb(letterOfTheWord, pushWords2[k][z]);
                    Probability[pushWords2[k][z]] = tempProb2;
                    z++;
              }
        }
        pushProbQueue = true;
        break;
}
```

After the probabilities of a node are created, we push this on the variable 'testProb2'. We keep track when nodes are split. After this is done we move a letter position and we do the same for the other letters.

```
else if(!firsTime )
{
      if(listTempWord.isEmpty())
      {
            addProbs.add(testProb2);
            int SplitingOfNode = 0;
            for(int o = 0; o<32; o++)
            {
                  if(testProb2[o] != 0)
                  {
                        SplitingOfNode++;
                  }
            }
            splitNodesLocation[WhichNode+1] = SplitingOfNode;
            WhichNode++;
```

## 2. Dasher Works

After the nodes are created Dasher is ready to construct this node visual. We explain on the hand of two strings.

- string 1: '/SpeeG/hallo/0.1/hello/0.1/noo/0.1/|'
- string 2: '/SpeeG/how/0.1/my/0.1/|'

In figure 48, the first string is shown in Dasher. Next when the middle of the circle reaches a node, another string is fetched from the socket. These nodes are attached to the last nodes of the previous words. This is depicted in figure 50. The

words ‘how’ and ‘my’ are showed three times. From the moment that the middle of the circle reach the end of the first words, i.e. the red square of the words ‘hello’, another string is fetched if there is one, otherwise Dasher stops.

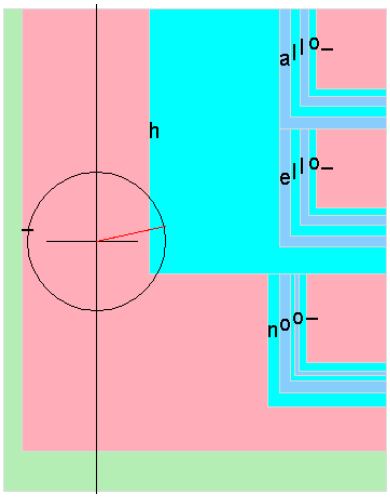


*Figure 48: Dasher ‘hello’ & ‘hallo’ and ‘noo’*

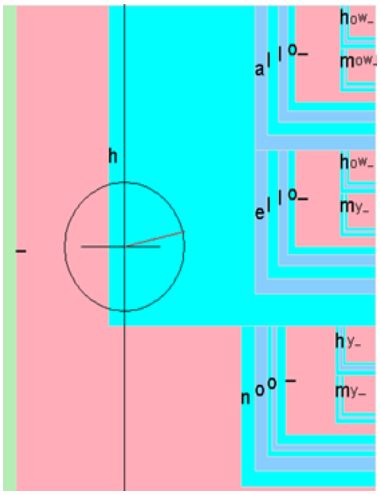


*Figure 49: Broken Nodes*

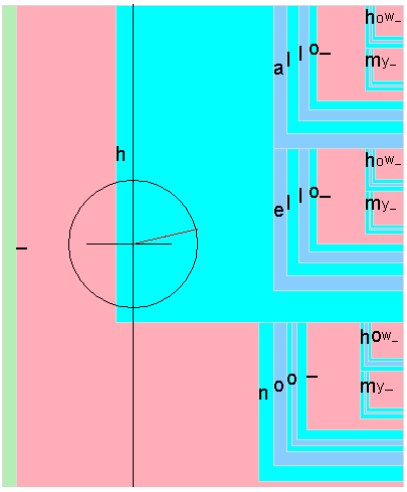


*Figure 50: Dasher ‘how’ & ‘my’*

Nodes are attached to the end of the previous nodes. ‘how’ and ‘my’ are attached to ‘hallo’, ‘hello’ and ‘noo’. First we have to know how many words the first string had, in this example this was three. So the words ‘how’ and ‘my’ are three time attached. The amount of words is put in the variable ‘AmountOfSquare’.

The first node for the words ‘how’ and ‘my’, which is ‘h’ and ‘m’. The next code just multiply this node with the ‘AmountOfSquare’ which is three. The second node represent the letter ‘o’ of how. We are not able to multiply this node with three. This result in broken nodes which is showed in figures 49.

We solve this by first push one node in the queue ready for used in creating the final nodes plus we queue this node in ‘tempQueue’ variable. After this we fetch the next nodes, we push this node, and eventually this node is also queued. This process is repeated of how many splits occur.

After the previous process is done, we fetch the first node which is queued in the ‘tempQueue’. We push this node on the other queue. Next, we fetch next node

which is queued of the 'tempQueue'. The nodes which are fetched from the 'tempQueue' are not removed. This process is repeated as often as there are squares.

```
if (!addProbs.isEmpty() && secondTime) {

tempQueue = new ArrayList<long[]>();

int AmountOfwordsInArray = 0;
long[] Temp4 = new long[32];
long[] TempTest = addProbs.remove();

int secondCounter = 0;
WichProbe++;

for(int p = 0; p<AmountOfSquares ; p++){
 Temp4 = new long[32];
 if(AmountOfwordsInArray == 0){
       System.arraycopy(TempTest, 0, Temp4, 0, iChildCount-1);
       addProbs2.add(Temp4);
       if(flagfirsTime ){
             flagfirsTime = false;
       }
       else if(splitNodesLocation[(WichProbe-1) ] > 1){
             tempQueue.add(Temp4);
             AmountOfwordsInArray++;
             for(int e = 0; e < (splitNodesLocation[(WichProbe-1)] - 1) ;
e++){
                   if(!addProbs.isEmpty()){
                         AmountOfwordsInArray++;
                         Temp4 = new long[32];
                         System.arraycopy(addProbs.remove(), 0, Temp4, 0,
iChildCount-1);
                         addProbs2.add(Temp4);
                         tempQueue.add(Temp4);
                   }
             }
       }
 }
 else{
       int k = 0;
       long[] Temp6 = null;
       for( ; k<(AmountOfSquares-1); k++){
             for(int j = 0; j<AmountOfwordsInArray; j++){
                   Temp6 = new long[32];
                   System.arraycopy(tempQueue.get(j), 0, Temp6, 0,
iChildCount-1);
             addProbs2.add(Temp6);
             }
       }
        p=k;
      }
}
}
```

## 3. Remove Nodes

Nodes are removed for several reasons. First the more nodes we have to create the more CPU power we need. Another issue is that some nodes disappear when they are not visible anymore. The problem is that nodes are attached to other nodes. So when nodes are destroyed we are not sure that nodes are correctly attached. Therefore we remove nodes which are not necessary anymore.

The next function is used to remove nodes. First we see that the node under the mouse is bigger than the variable 'pvH2' (previous highest nodes). 'pvH1' variable is the previous 'pvH2'. The 'pvH2' variable is the highest node number of the previous word. In figure 51 the highest node number belongs to the symbols 'W' and '.' five. After the creation of a new symbol, for example 'a' and '.', 'pvH2' is equal to this number.

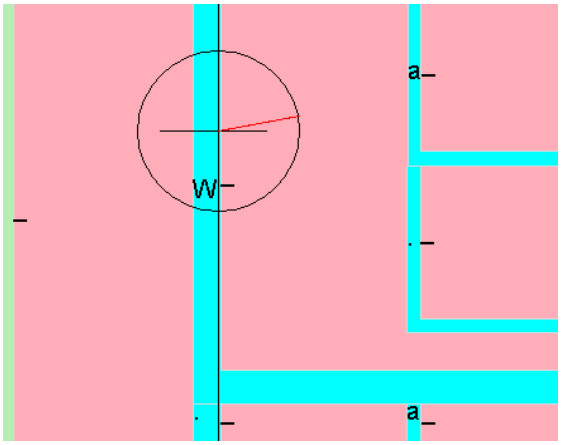



*Figure 36: Dasher 'W'*

The node is destroyed if we enter one of the next nodes 'a' or '.'. The 'pvH2' variable is still five.

Every character has a number; in SpeeG this called a 'nodenumber'. There is a chance that one node carry more than one node number. For example in figure 52, the node first nodes '_' carry the node 'n' and 'z'. These nodes have all different node numbers. This most left '_' character has the node number 1, the 'n' has the node number 2, and 'z' three and so on.

At this moment we have three words and the biggest word of those three words is five plus one for the space symbol.

We expect that the latest node number is three multiply with six (five character plus space symbol). The outcome is 30. Still the latest character of ‘zorro’ is 29. The reason why one node is lost is because ‘n’ of the words ‘nah’ and ‘noh’ is count once and not twice. These lost nodes are needed for removing the correct nodes.

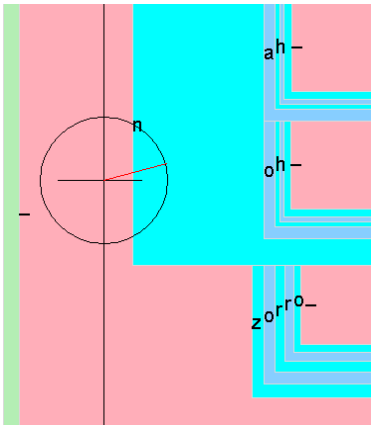



*Figure 37: Dasher ‘nah’ & ‘noh’ and ‘zorro’*

The variable ‘latestSPLIT’ is used to see on which location the nodes are latest split. In the above example this is in the second letter of the words. We do not remove words (nodes), if the words are still need to be split. For example if we want to keep the word ‘noh’ and remove the other words we remove the ‘a’ from ‘nah’ and from ‘zorro’ the ‘o’. This is the same when we want to keep the word ‘zorro’, we remove ‘a’ from the word ‘nah’ and the ‘o’ from the word ‘noh’.

The only output that we receive from this function is which node we want to keep, ‘removeNodeNumberDont’, and the begin of the node ‘removeNodeNumberBegin’ and then of the ‘removeNodeNumberEnd’. The nodes between are removed.

For example when we want to keep track of the word ‘noh’, we fetch the node number of the character ‘o’. This node number is not removed. We know the total amount of nodes this was 29 (‘pvH2’). The formula for calculating is:

*Begin = pvH2 - AmountOfWords___[1] * AmountOfWords___[2] * (BigstOfWords___[2] +1 - (latestSPLITInt2___[2]-1));*

Between the ‘Begin’ variable and the ‘End’ variable nodes are removed. The children of those are destroyed. Destroying represent that the nodes are not drawn anymore in future, the character is set on “” and the color is white.

```
public static void removeNodes() {
lastWord++;
if(FirstTimeRemove){
      CDasherNode nodeUnderMouse = m_Model.Get_node_under_mouse(MX, MY);
      previousDepthW = nodeUnderMouse.DepthW;
      if(nodeUnderMouse.getNodeNummer()<pvH2){
      }
      else{
      lost = pvH2 - AmountOfWords___[2]* ((BiggerstOfWords___[2]+1))-1 ;
            if(nodeUnderMouse.getNodeNummer()>pvH1){
                  if(nodeUnderMouse.m_Parent.m_strDisplayText.equals("W")){
                        nodeUnderMouse = nodeUnderMouse.m_Parent;
                  }
                  if(nodeUnderMouse.m_Parent.m_strDisplayText.equals(".")){
                        nodeUnderMouse = nodeUnderMouse.m_Parent;
                  }
                  boolean temp = true;
                  while(temp){
                  if(!nodeUnderMouse.m_strDisplayText.equals("_") ){
                        nodeUnderMouse = nodeUnderMouse.m_Parent;
                  }
                  else{
                        temp = false;
                  }
                  }
                  int backward = (BiggerstOfWords___[2] -
(latestSPLITInt2___[2]-1));
                  for(int i = 0; i<backward; i++){
                        nodeUnderMouse = nodeUnderMouse.m_Parent;
                  }
            }
            int End = pvH2 - AmountOfWords___[2] * (BiggerstOfWords___[2] -
(latestSPLITInt2___[2]-1));
            removeNodeNumberEND = End+1 ;
            FirstTimeRemove = false;
            removeNodeNumberBEGIN = End - AmountOfWords___[2];
            removeNodeNumberDONT = nodeUnderMouse.getNodeNummer();
            rebuildNodes = true;
            destroyed2 = true;
      }
```

```
}
else{
      CDasherNode nodeUnderMouse = m_Model.Get_node_under_mouse(MX, MY);
      boolean passed = true;
      if(nodeUnderMouse.getNodeNummer()<pvH2) {
      }
      else if(nodeUnderMouse.getNodeNummer()>pvH1 && passed &&
previousDepthW < nodeUnderMouse.DepthW){
            previousDepthW = nodeUnderMouse.DepthW;
            if(nodeUnderMouse.m_Parent.m_strDisplayText.equals("W")){
                  nodeUnderMouse = nodeUnderMouse.m_Parent;
            }
            if(nodeUnderMouse.m_Parent.m_strDisplayText.equals(".")){
                  nodeUnderMouse = nodeUnderMouse.m_Parent;
            }
            boolean temp = true;
            while(temp){
                  if(!nodeUnderMouse.m_strDisplayText.equals("_")){
                        nodeUnderMouse = nodeUnderMouse.m_Parent;
                  }
                  else{
                        temp = false;
                  }
            }
            lost = pvH2 - pvH1 - AmountOfWords___[1]*AmountOfWords___[2]*
((BiggerstOfWords___[2]+1));
            int backward = (BiggerstOfWords___[2] - (latestSPLITInt2___[2]-
1));
            for(int i = 0; i<backward; i++){
                  nodeUnderMouse = nodeUnderMouse.m_Parent;
            }
            int Begin = pvH2  -   AmountOfWords___[1] * AmountOfWords___[2]
* (BiggerstOfWords___[2] +1 - (latestSPLITInt2___[2]-1));
            removeNodeNumberBEGIN = Begin;
            removeNodeNumberEND = Begin +
AmountOfWords___[1]*AmountOfWords___[2]+1;
            FirstTimeRemove = false;
            removeNodeNumberDONT = nodeUnderMouse.getNodeNummer();
            destroyed2 = true;
            AmountOfSquaresSet(AmountOfWords___[3]);
      }
      else{
      }
}
rebuildNodes = true;
}
```

The function removeNodes is called in two functions. The first call appears in the class 'JDasherEdit' when there is new word on the queue and the amount of words, the depth, is more than two. Still there is a chance that nodes are not removed because the node under the mouse is incorrect. Therefore a second call

is made in 'CDasherViewSquare'. This is when the queue which contains the string of words is not empty and there are still nodes need to be deleted. Nodes should first removed and from then new words are allowed to be create.

## 4. New Root

Normally nodes have a root. This is the parent of the parent and so on till the beginning. The reason for setting a node as new root has a lot of advantages. The first one is that these nodes are not used anymore, so we have less CPU consummation. Further when we do not make a new root nodes are getting corrupt. Therefore we create a new root from the nodes which are not removed. This is explained in the previous section. The node which is not removed 'removeNodeNumberDONT' is used as the new root.

## 5. Speed

The speed expresses how fast we travel into the nodes. This can be very slow to very fast. The faster the user travels into the nodes, the more CPU power is acquired. This is necessary because the application has to create new nodes fast. In Dasher this is the distance from the mouse till the middle of Dasher (the cross). The bigger this line the faster Dasher goes. In left figure 53 Dasher moves slowly. In the right figure 54 Dasher moves very fast. The larger the red line, the faster Dasher moves.

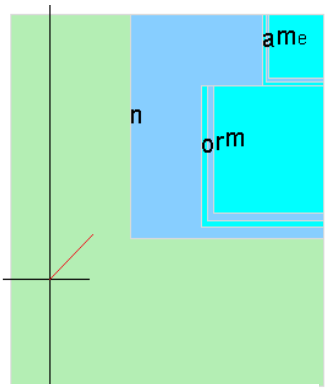



*Figure 38: Dasher Slow*

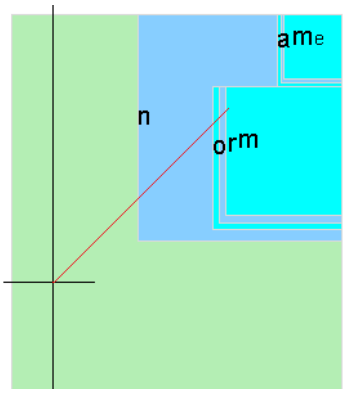



*Figure 39: Dasher Fast*

In SpeeG the application has a constant red line, see figure 55. Further we change ‘m_dMaxbitrate’ to a value that we received from the Kinect.

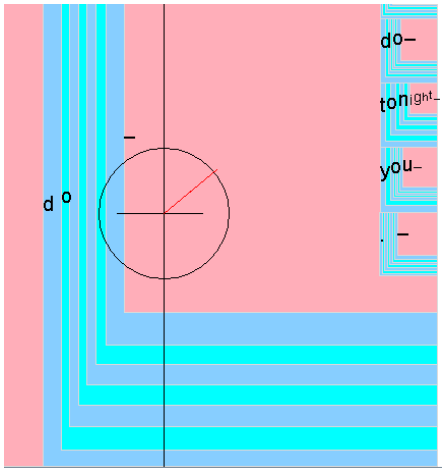


*Figure 40: SpeeG Speed*

The next code calculate the ‘x[1]’ value. The ‘y[0]’ and ‘x[0]’ represent the middle point. The ‘y[1]’ is the coordinate from the user his hand. The formula to calculate the ‘x[1]’ is:

$$(x-a)^2 + (y-b)^2 = r^2$$

The ‘-a’ represent the ‘x[1]’ which we need to calculate. The distance is, ‘r’, is set on 250000.

```
public void DashPolyline2(long[] x, long[] y, int n, int iWidth, int
iColour) {
      CDasherView.Point[] ScreenPoints = new CDasherView.Point[n];

      long y0 = y[0];
      long y1 = y[1];

      long ThisDistance = 250000;
      x[1] = x[0]-(long) (Math.sqrt(ThisDistance -(y1-y0)*(y1-y0)));

      if(y1 > 2566){
            y[1] = 2566;
      }
      if(y1 < 1536){
            y[1] = 1536;
      }
...
```

### 6. Words Of Sphinx

There is a chance that the recognize system detect no words. The reason can be that the recognize system is still finding the word. There is a probability that the system not recognize the words that users say.

If there are no words Dasher push the words 'W' and '.'. The 'W' stands for wait and '.' is a dot. When the user travels to these nodes and no other words are recognize, Dasher is stuck in this node. This is shown in figure 56. Dasher is unstuck if new words are recognized. This is shown in figure 57.

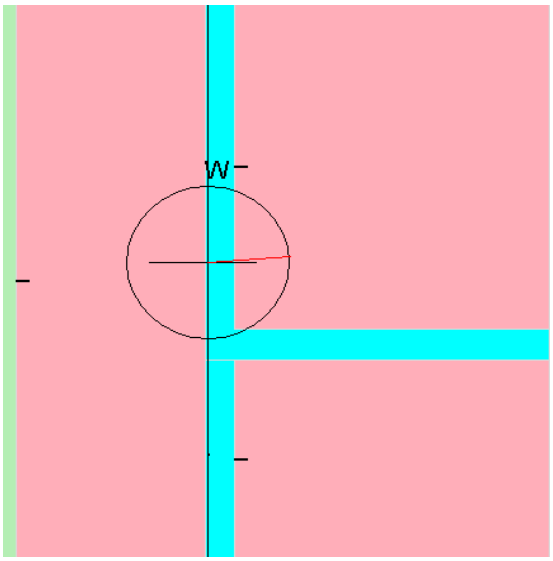



*Figure 56: Dasher Waits*

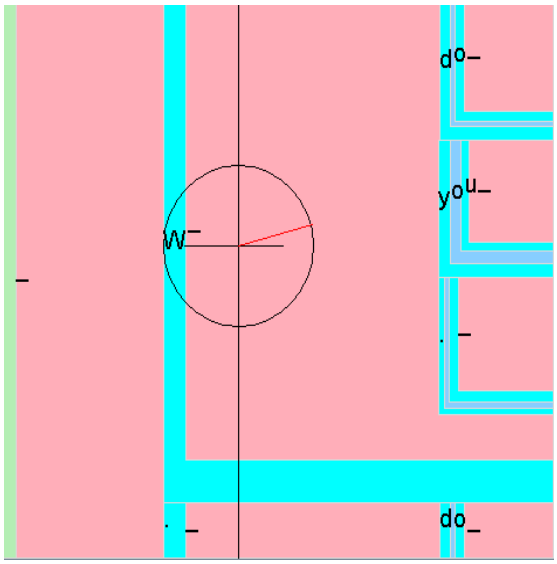



*Figure 57: Dasher Continues*

## H. Skeleton Tracking

The skeleton tracking is done with the Kinect. The SpeeG uses the 3D camera to track the body and to capture gestural push events.

### 1. Body figure

The user segmentation is to identify and track users in the scene. Each user in the scene is given a unique ID. The main output of the user segmentation process is a label map for every user ID. This label map will be used in skeleton tracking algorithm to generate a skeleton.

Some basic assumptions for skeleton tracking are that the ideal distance of the user and Kinect is around 2.5 meter. It is not recommended to wear very loose clothing or to have long hair. Further the upper body must be inside the field of view.

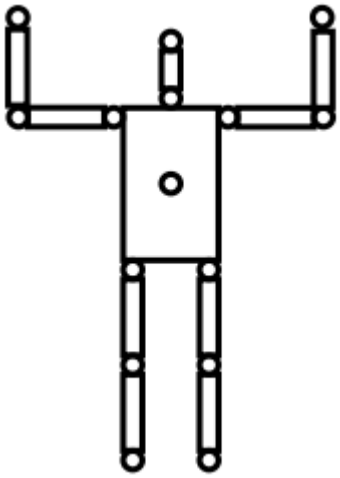


*Figure 41: Body Figure*

In case he skeleton tracking is completed. Calibration will takes place. Calibration is used to adjust the skeleton to the user's body proportion. Pose tracking begins to work after calibration is completed. The calibration step currently takes a few seconds (~<3s). The ideal position of the user is showed in figure 58.

The figure below illustrates the coordinate system and skeleton representation for a user facing the sensor. Every joint position is given in the real world coordinate system. The origin of the system is at the sensor. +X points to the right of the, +Y points up, and +Z points in the direction of increasing depth.

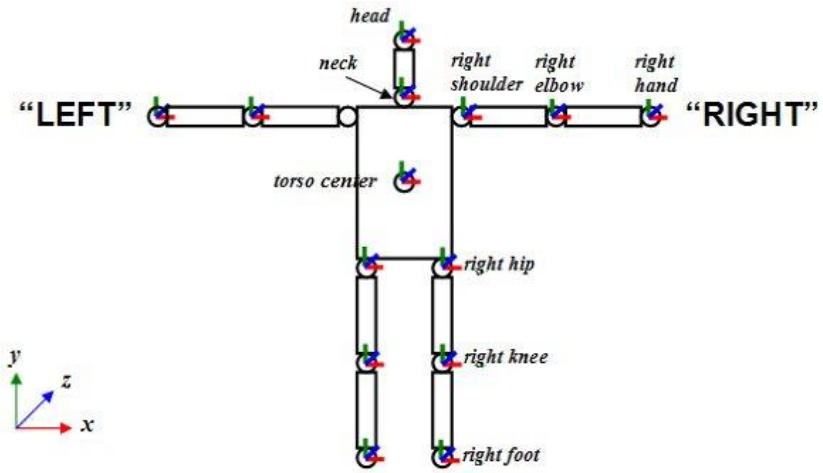



*Figure 42: Joints Body*

There are some known issues. Very fast motions may cause tracking failure. Another issue is that when a skeleton is stuck in a faulty pose, then it is recommended to a simpler pose. Leg tracking is unstable and noisy. The problem is that most of the time legs are too close to each other. Recalibration can resolve most of the problems.

## 2. Push Detection

The PrimeSense has a built control gesture library. They are basic control gesture. For example a wave or click can detect by the Prime Sense library. For starting this application we will use a sort of push/click function. From the moment the user pushes with his left hand the application starts or stops.

The PrimeSense library provides a sort of click detection. From the moment a hand is performing a push operation, an event occurs. The problem that we met after implementing is that we have to do a 'ForceSession'. With the 'ForceSession' we force to look for a hand in a region.

The API gave incorrect information on the push recognition. There was confusion between the left and the right hand. Further, if the hand is near to the body, the push is not be seen. Therefore we developed a simpler but better algorithm.

For the detection of the push we track the distance of our body to our hand in the depth. If the difference is more than 20 cm, a click event occurs. After this event a sleep operation takes place, so that there is one click send and not multiple times. With the push detection our application can be started or can be stopped.

### 3. Hand Coordinates

The hand coordinates are used for several functions. The right hand is used for pointing the way our Dasher has to go. The range between the lowest point and the highest point is between 0 and 1.

```
PersonInfo pi = *personInfo[player];   //The player information
XnPoint3D pt[] = { joint.position };   //3D point of specific joint
//--> example Joint number 6 is the Left shoulder

if (pt[0].Y < pi.y_min) {
          pi.y_min = pt[0].Y - 1;
}
if (pt[0].Y > pi.y_max) {
          pi.y_max = pt[0].Y + 1;
}

double y = (double)(pt[0].Y - pi.y_min) / (pi.y_max - pi.y_min);
```

*Figure 43:Test User*

## 4. Speed

For calculating the speed we use a combination of the left shoulder, the right shoulder and the left hand. At first, we fetch the left shoulder coordinates. Secondly, we fetch the right shoulder coordinates. Then, we calculate the distance between the right and the left shoulder. For this calculation we use the distance formula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The first point is the right shoulder and the second point is the left shoulder. The maximum distance between the left shoulder and the right shoulder is double of this result.

Further we calculate the distance between the left shoulder and the left hand. If this is more than 0, we use this result and we subtract the distance between the

left shoulder and the right shoulder. The outcome is the speed that we send to Dasher.

```
if(i == 6){ //Left Shoulder);
     xLS = pt[0].X; //X Coordinates Left Shoulder
     yLS = pt[0].Y; //Y Coordinates Left Shoulder
}

if(i == 12){ //Right shoulder)
     xLR= pt[0].X;    //Y Coordinates Left Shoulder
     yLR = pt[0].Y;  //Y Coordinates Left Shoulder
     //Distance Right shoulder & Left shoulder
     DistRSandLS = sqrt ((xLS - xLR)*(xLS - xLR) + (yLS - yLR)*(yLS -
yLR));
}

if(i ==9){  //Left hand
     DiffShoulderAndHand = xLS - pt[0].X; //Difference between Shoulder &
hand
     if(DiffShoulderAndHand < 0){
           // Do nothing
     }
     else{
           if((DiffShoulderAndHand - DistRSandLS)>0){
                 SpeedResult = DiffShoulderAndHand - DistRSandLS;
                 //Difference between the distance (Left shoulder & Left
hand)
                 //and distance (Right shoulder & Left Shoulder)
           }
           else{
                 // Do nothing
                 SpeedResult = 0;
           }
     }
}
```

# I. Sphinx

Sphinx is the recognize system that is implemented in SpeeG. Sphinx tries to recognize words the users said. Sphinx presents a list with possible words. Normally a speech recognition system only returns the best recognized word. For SpeeG we want to know all the words, also the words which have a probability of almost zero.

After speaking a word we received a bunch of results. Words are put in 'Nodes'. These nodes contain other nodes. The nodes are the words which Sphinx recognizes. In the output below the beginning and the end represent the frame during which the words are recognized. Every frame is 10ms long. For example the first 'you' is recognized from the 0 frame till the 43th frame and it has a confidence of 0.328. Further the second 'you' has a different end frame and the confidence is not the same as the previous one. Other words like 'a', starts at the 6th frame. The other 'a', starts at the 37th frame and so on.

```
-Node----- => 4618641
---------- => you          Begin:0   End:43 Confidence: 0.328)
-Node----- => 32619928
---------- => you          Begin:0   End:50 Confidence: 0.28)
-Node----- => 19935173
---------- => you          Begin:0   End:38 Confidence: 0.39)
-Node----- => 1340650
---------- => <sil>          Begin:0   End:6 Confidence: 0.28)
-Node----- => 24325842
---------- => a          Begin:6   End:28 Confidence: 0.025)
-Node----- => 13129484
---------- => a          Begin:37   End:50 Confidence: 0.0025)
-Node----- => 26174809
---------- => you          Begin:0   End:37 Confidence: 0.0001)
-Node----- => 14513572
---------- => a          Begin:29   End:50 Confidence: 0.089)
-Node----- => 1173553
---------- => a          Begin:43   End:50 Confidence: 0.035)
-Node----- => 26030331
---------- => a          Begin:6   End:29 Confidence: 0.070)
-Node----- => 749304
---------- => </s>          Begin:50   End:-1 Confidence: 0.281)
-Node----- => 13101223
---------- => do          Begin:28   End:50 Confidence: 0.203)
-Node----- => 14098944
---------- => <sil>          Begin:0   End:5 Confidence: 0.258)
-Node----- => 15272259
---------- => a          Begin:6   End:50 Confidence: 0.089)
```

## 1. SpeeG - Sphinx

We modified the code of Sphinx to fetch all the recognized words. SpeeG uses the following class of Sphinx:

- ✓ Lattice
- ✓ Confidence
- ✓ SausageMaker

The following code handles all the nodes and puts the first recognized words in a queue. When the first node contains values that are not interesting, for example the silence, we track the second word.

```
protected List<String> allPathsFrom(String path, Node n,
                String NodeNummer) {
boolean CheckForNonWords = false;
String p = "";
if(n != null){
     String tempString = n.getWord().toString();
     boolean FirstWord = true;
     if (SausageMaker.LocationCounter != 0) {
           if (path.equals("")) {
                 CounterTemp = -1;
           }
     }
if (!tempString.equals("</s>") && !tempString.equals("<sil>")
           && !tempString.equals("<s>")) {
TempN = n;
}
if (!n.getWord().toString().equals("<s>")
     && !n.getWord().toString().equals("</s>")
     && !n.getWord().toString().equals("<sil>")) {
           if (!n.getWord().equals("</s>") || !n.getWord().equals("<sil>")
                           || !n.getWord().equals("<s>")) {
                 p = n.getWord().toString() + "ID" + n.getId();
                 if (!NodeNummer.equals(tempID)) {
                       if (CounterTemp >= SausageMaker.LocationCounter) {

     if(!SocketWords.contains(n.getWord().toString())) {
           SocketWords.add(n.getWord().toString());
           tempID = n.getId();
           temp33 = false;
           }
     else {
           CounterTemp++;
     }                 }
           }
```

```
    }
}
} else {
    CheckForNonWords = true;
    p = path;
}
}
    List<String> l = new LinkedList<String>();
    if (n == terminalNode) {
        l.add(p);
    } else {
        for (Edge e : n.getLeavingEdges()) {
            l.addAll(allPathsFrom(p, e.getToNode(),
e.fromNode.getId()));
        }
    }
    return l;
}
```

## 2. Sphinx Results

To send the result to Dasher we create a thread. When the recognition system doesn't recognize words, Sphinx sends the following string, /SPEEG/ /|. If there are words recognized the following string is send to Dasher: /SpeeG/a/0.1/do/0.1/you/0.1/|

```
public void run() {
boolean SendAlready = true;
int counter = 0;
while (true) {
    if(counter == 0){
        try {
            mylink.Connect();
        } catch (IOException e) {
        // TODO Auto-generated catch block
            e.printStackTrace();
        }
        try {
            String temp = "";
            if(!QueueListWithWords.isEmpty() && !SendAlready) {
                temp = QueueListWithWords.remove();
            }
            else{
                SendAlready = false;
                temp = "/SPEEG//|";
            }
            mylink.SendString(temp);
        } catch (IOException e) {
```

```
            // TODO Auto-generated catch block
                  e.printStackTrace();
            }
            try {
                  mylink.Close();
            } catch (IOException e) {
            // TODO Auto-generated catch block
                  e.printStackTrace();
            }
      }
      else{
            if(!QueueListWithWords.isEmpty()){
                  counter = 0;
            try {
                  mylink.Close();
            } catch (IOException e) {
            // TODO Auto-generated catch block
                  e.printStackTrace();
            }
      }
      }
}
```

# Chapter 4: Evaluation & Results

To evaluate the SpeeG prototype, we performed a qualitative study with 7 users. The qualitative study blueprints the reactions and feelings of an independent group of users towards our hand gesture-assisted speech recognition solution. Our main goal was to test the feasibility of our approach and to investigate the current usability of the prototype. Additionally, it is important to explore the potential of improved versions in the future in order to find out if further development is valuable. After all, to measure is to know.

## A. Method

The study included 11 mandatory checkbox questions using the five-point Likert scale and 5 open questions.

The experiment was conducted with men and women between 18 and 35 years of age. All the subjects had the profile of expert computer users, and all obtained at least a Bachelor's degree.

The survey evaluated whether interaction with the SpeeG application was enjoyable, whether the quality of the implementation is sufficient and whether the prospects of the technology are positively oriented. The following questions and answers provide those insights.

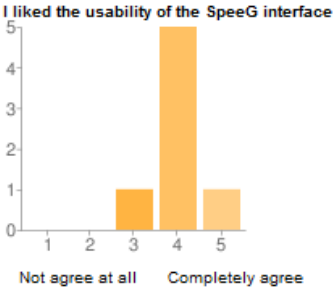
## 1. Statements that tests if the users were open for new technologies and whether they enjoyed using the application

*I'm actively seeking for new kinds of text-input interfaces.*


| | | | |
|---|---|---|---|
| 1 - | Not agree at all | 1 | 14% |
| 2 | | 2 | 29% |
| 3 | | 1 | 14% |
| 4 | | 1 | 14% |
| 5 - | Completely agree | 2 | 29% |

A significant share of the group indicated to be searching for a new method to input text. Although, we can argue that our audience also contained users with a negative prejudgement. It is important to stress that the group consisted of academics and therefore these figures should not be generalised.

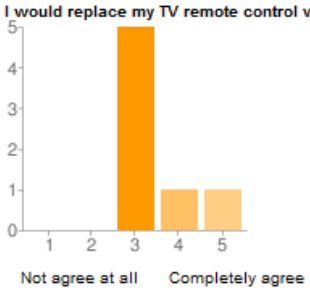
*I liked the usability of the SpeeG interface.*




| | | | |
|---|---|---|---|
| 1 - | Not agree at all | 0 | 0% |
| 2 | | 0 | 0% |
| 3 | | 1 | 14% |
| 4 | | 5 | 71% |
| 5 - | Completely agree | 1 | 14% |

The graph above clearly shows that experimental subjects experienced SpeeG as a practical tool to insert text. 85% of the test group was satisfied of the usability.

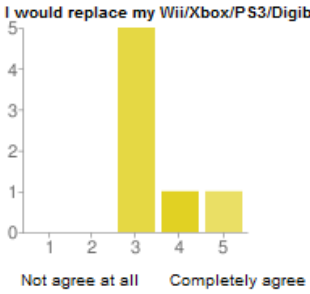
*I would replace my TV remote control with this interface today.*




| | | | |
|---|---|---|---|
| 1 - | Not agree at all | 0 | 0% |
| 2 | | 0 | 0% |
| 3 | | 5 | 71% |
| 4 | | 1 | 14% |
| 5 - | Completely agree | 1 | 14% |

At the moment, users did not find the prototype ready to replace the TV remote controller.

*I would replace my Wii/Xbox/PS3/Digibox on-screen keyboard with this interface today.*

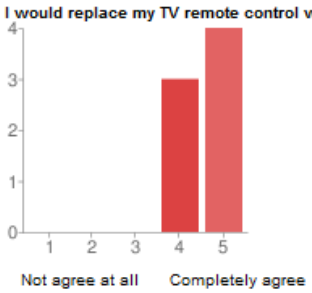



| | | | |
|---|---|---|---|
| 1 - | Not agree at all | 0 | 0% |
| 2 | | 0 | 0% |
| 3 | | 5 | 71% |
| 4 | | 1 | 14% |
| 5 - | Completely agree | 1 | 14% |

The results are the same as for the previous question. Again, 70% of potential users are doubtful. One possible reason for this is that the users only received some basic training, which is hard to compare to an every-day tool such as the remote control or game controllers. However, since 70% of the users are already neutral, we can already conclude that the current version is at least status quo with a remote control and other existing controllers. Additionally, we will show that in the next set of questions, all users experienced the tool in a positive, joyful way and believe in its potential as a future text-input mechanism.

## 2. The statements that analyse future significance of the application:

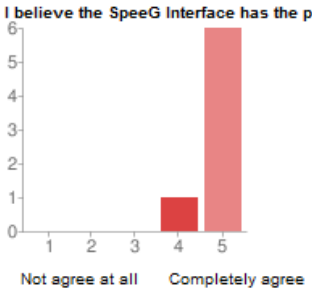
*I would replace my TV remote control with an improved version of this interface.*




| | | | |
|---|---|---|---|
| 1 - | Not agree at all | 0 | 0% |
| 2 | | 0 | 0% |
| 3 | | 0 | 0% |
| 4 | | 3 | 43% |
| 5 - | Completely agree | 4 | 57% |

A positive shift in comparison with previous questions is detected. Fine-tuning would definitely push users towards application of the technology in the home environment. No score lower than 4 has been recorded.

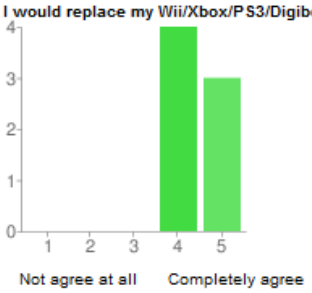
*I believe the SpeeG Interface has the potential to be more usable in future versions.*




| | | | |
|---|---|---|---|
| 1 - | Not agree at all | 0 | 0% |
| 2 | | 0 | 0% |
| 3 | | 0 | 0% |
| 4 | | 1 | 14% |
| 5 - | Completely agree | 6 | 86% |

As much as 86% of the users are convinced that future versions of the SpeeG Interface can provide an improved interface.

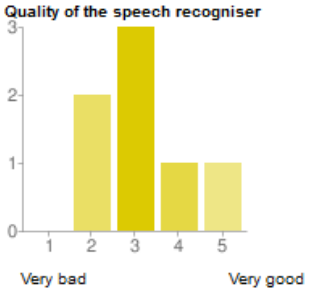
*I would replace my Wii/Xbox/PS3/Digibox on-screen keyboard with an improved version of this interface.*




| | | | |
|---|---|---|---|
| 1 - | Not agree at all | 0 | 0% |
| 2 | | 0 | 0% |
| 3 | | 0 | 0% |
| 4 | | 4 | 57% |
| 5 - | Completely agree | 3 | 43% |

## 3. The statements that question the quality of the implementation

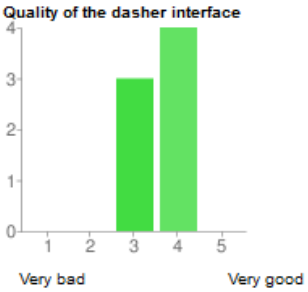
*Quality of the speech recogniser.*




| | | | |
|---|---|---|---|
| 1 - | Very bad | 0 | 0% |
| 2 | | 2 | 29% |
| 3 | | 3 | 43% |
| 4 | | 1 | 14% |
| 5 - | Very good | 1 | 14% |

The quality of the speech recogniser is only mediocre. This result was expected. Multiple influences have an effect on the accuracy of the speech recognition. There was a background noise from the projector during the test, not the best available microphone was used and some speakers had a non-US accent while the voice recogniser is based on the US accent. All these elements can disturb correct recognition. Additionally, we did not train the voice recogniser for the specific user, which is a prerequisite with all major speech recognisers out there today. With SpeeG, this 'training-step' can easily be skipped.

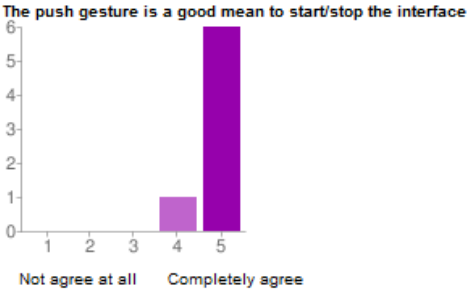
*Quality of the dasher interface.*




| | | | |
|---|---|---|---|
| 1 - | Very bad | 0 | 0% |
| 2 | | 0 | 0% |
| 3 | | 3 | 43% |
| 4 | | 4 | 57% |
| 5 - | Very good | 0 | 0% |

The dasher interface has an overall good score. The learning curve of the interface is quite steep. Once some effort from the user has been invested, efficiency gains can be observed.

*The push gesture is a good mean to start/stop the interface.*




| | | | |
|---|---|---|---|
| 1 - | Not agree at all | 0 | 0% |
| 2 | | 0 | 0% |
| 3 | | 0 | 0% |
| 4 | | 1 | 14% |
| 5 - | Completely agree | 6 | 86% |

There were no problems recorded with the push gesture.

*The pointing gesture is a good mean to control the dasher input.*


| | | | |
|---|---|---|---|
| 1 - | Not agree at all | 0 | 0% |
| 2 | | 0 | 0% |
| 3 | | 0 | 0% |
| 4 | | 1 | 14% |
| 5 - | Completely agree | 6 | 86% |

Users quickly mastered their pointing skills. 86% of the users completely agree that the pointing gesture is a good mean to control the input.

Subsequently, the open questions are listed and in response direct quotations of the experimental group. We selected a few interesting responses. An exhaustive list of all answers can be found in Appendix X.1.

*Why would you (not) like to replace your current TV remote control with this interface?*

> "With practice this interface would help you type sentences or words you wan't to search on your T.V. without the annoying selection process they work with today"

*Why would you (not) like to replace your current Wii/Xbox/PS3/Digibox on-screen keyboard with this interface?*

> "The way in which word insertion is done in these devices is extremely annoying and time consuming."

*Why do you think this is (not) an improvement over existing speech technology?*

> (+) Corrections! -> Speech recogniser which _works_ ?
> (+) Shows that speech recognition from multiple results for one word (people are not always aware of that)
> (-) Heavy workload for the brain
> (-) Hard to have really fluid input (at least in the beginning)"

*Why did you (not) like it?*

> "I think it would be handier if both the speed and the direction could be done by only the right hand. I stretch the arm I am pointing with when I want to gain speed. But maybe this is just a matter of habit."
>
> "This interface makes the writing of words fun by itself. It could even become a game."
>
> "I liked the fluent motion of the interface. It took some getting used to, but I can imagine it becoming a second nature."

*Do you have any proposals to improve the usability? (controls, interface, performance,…)*

> "The control of the speed can be a bit tiring after a while, maybe it would be cool to put it to some place where your hand doesn't need to be raised all the time."

# Chapter 5: Conclusion

The foundations are built and the first brick is there. SpeeG has grown from a concept to an application.

Speech recognition problems can now be corrected in a fast and pleasant manner. We believe that the fine tuning of SpeeG is a crucial next step.

Our user evaluation confirms that SpeeG is a promising new text input interface. The reactions of users were mainly positive and showed major interest in the idea of a novel speech- and gesture-based input method. They also argue that SpeeG can be a viable replacement for existing techniques. Further, the fact that the controller has become an artefact of the past is no longer science fiction. Standing still is moving backwards.

We would recommend further development of the SpeeG application, so that the use in a home environment can be accelerated. We also believe that there are applications of the SpeeG technology that we have not thought about yet.

The performed experiments allows us to conclude that SpeeG is an enjoyable useful tool and future improvements would definitely be valuable.

# Chapter 6: Future Work

The work done has certainly made an impact. However, there are a handful of issues that could still benefit from optimalisation and fine tuning.

Before the SpeeG application can be used, three different programs need to be activated. It is obvious that this is not favourable. It would be desirable to add a start/stop button that allows all three programs to start at once.

The next step is to calibrate the Kinect camera. This phase takes a few seconds during which the user has to lift both arms before calibration takes place. Users may find this setup annoying and time consuming, and we would not disagree. The time of calibration could be reduced by using the implementation of the Kinect SDK, instead of OpenNI that we currently use.

Another element that we have noticed to be suboptimal is the 'W' that appears when a complete word has been selected. The 'W' symbol induces a moment to 'wait'. It would be better to skip this step. A fluent movement from one word to another is more advisable.

Further, there have been some problems with the speech recognition software. We were obliged to stick to the limited number of 250 words to be recognised by Sphinx. In a future version of SpeeG, this issue has to be reviewed and hopefully the identifiable vocabulary can be expanded. This is very important since SpeeG would be used to replace existing text input methods. There is already the possibility to correct but an optimisation of the system at the basis, there were the words are recognised, is the best way to start off.

The user evaluation has been very efficient in pointing out the flaws in the system. Multiple times it was the case that users where convinced the system did not pick up any speech input. This lead to users speaking the same word twice or three times. This implied that the user travelled through the path and the words kept (re)appearing on the right-hand side. We would suggest to add an indication mechanism to denote the number of words already recognised. For instance the

developer could employ colors to make a distinction. Another possibility is that a symbol appears at the moment the user may pronounce a word.

The following issue handles about the Dasher navigator. In the current version of SpeeG it is not possible to move backwards. The user can indicate the next word they would like to type, but if they change their mind about the sentence they would like to write, they cannot delete the last written words. It would be possible to introduce this option if the previous words would be stacked in a queue.

Further, it is important to emphasise the fact that in the future eye tracking by the Kinect or any other camera could become reality. For the moment, this is not possible because the resolution of the camera is too low. But better resolutions could open new doors. SpeeG could be adapted to be controlled by eye movement instead of hand gestures. People who are physically disabled would be capable to input text in this novel way, which would increase the quality of their lives drastically. We have to stress that this is future talk. At the moment, we are not capable of such an implementation.

However the SpeeG application could use fine-tuning, we have been able to demonstrate the value of SpeeG in the search for new text input methods and the usability of the current prototype. The thesis is a steady foundation for further development.

# Chapter 7: Bibliography

[1] R. Thangarajan and A.M. Natarajan. A Robust Front-End Processor combining Mel Frequency Cepstral Coefficient and Sub-band Spectral Centroid Histogram methods for Automatic Speech Recognition. *International Journal of Signal Processing, Image Processing and Pattern Recognition,* 2(2), pages 67–97, June 2009.

[2] Daniel Jurafsky and James H. Martin. Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition, and Computational Linguistics (2nd Ed.), Prentice Hall, 2009.

[3] Per-ola Kristensson. SHARK: A Large Vocabulary Shorthand Writing System for Pen-Based Computers. *In Proceedings of UIST 2004, 17th Annual ACM Symposium on User Interface Software and Technology*, pages 43–52, Santa Fee, USA, October 2004.

[4] Michael A. Grasso, Structured Speech Input for Clinical Data Collection, Ph.D., Segue Biomedical Computing, Laurel, Maryland and University of Medicine and Dentistry of New Jersey, 2002.

[5] *Keith Vertanen.* Speech and Speech Recognition during Dictation Corrections. In Proceedings of *ICSLP 2006, 9th International Conference on Spoken Language Processing,* pages 1890–1893, Pittsburgh, Pennsylvania, September 2006.

[6] Timo Sowa, Fiorenza Arisio and Luca Cristoforetti. DICIT: Evaluation of a Distant-talking Speech Interface for Television. *In Proceedings of LREC 2010, 7th International Conference on Language Resources and Evaluation*, pages 2161–2168, Valetta, Matla, May 2010.

[7] Joanna Lumsden, Scott Durling and Irina Kondratova. A Comparison of Microphone and Speech Recognition Engine Efficacy for Mobile Data Entry. *In Proceedings of MONET 2008, 3rd International Workshop on MObile and NEtworking Technologies for Social Applications*, pages 519–527, Monterrey, Mexico, November 2008.

[8] Tomáš Beran, Vladimír Bergl, Radek Hampl, Pavel Krbec, Jan Šedivý, Bořivoj Tydlitát and Josef Vopička. Embedded Viavoice, *In Proceedings of TSD 2004, 7th International Conference on Text, Speech, Dialogue*, pages 269–274, Brno, Czech Republic, September 2004.

[9] Peter Boothe, Using Cell Phone Keyboards is (NP) Hard, *In Proceedings of FUN 2010, 5th International Conference on Fun with Algorithms*, Ischia, Italy, June 2010.

[10] David J. Ward, and Alan F. Blackwel. Dasher - A Data Entry Interface Using Continuous Gestures and Language Models. *In Proceeding of UIST 2000, 13th International Conference on User Interface Software and Technology*, pages 129–137, San Diego, USA, November 2000.

[11] Tom Bellman and I. Scott MacKenzie. A Probabilistic Character Layout Strategy for Mobile Text Entry. *In Proceedings of GI 1998, 3rd International Conference on Graphics Interface*, pages 168–176, Vancouver, Canada, June 1998.

[12] Thomas Schlömer, Benjamin Poppinga, Niels Henze and Susanne Boll. *Gesture Recognition with a Wii Controller. In Proceedings of TEI 2008, 2nd International Conference on Tangible and Embedded Interaction*, pages 11–14, Bonn, Germany, February 2008.

[13] Keith Vertanen and David J.C. MacKay. Speech Dasher: Fast Writing Using Speech and Gaze. *In Proceedings of CHI, 28th International Conference on Human Factors in Computing Systems,* Atlanta, Georgia, USA, April 2010.

[14] David Huggins-Daines, Mohit Kumar, Arthur Chan, Alan W Black, Mosur Ravishankar and Alex I. Rudnicky. A Free, Real-time Continuous Speech Recognition System for Hand-held Devices. *In Proceedings of ICASSP, 31st International Conference on Acoustics, Speech, and Signal Processing*, pages 185–188, Toulouse, France, March 2006.

[15] Shumin Zhai, Michael Hunter, and Barton A. Smith Performance Optimization of Virtual Keyboards. *Human-Computer Interaction, 17(2): 89–129,* 2002.

[16] Hedy Kober, Eugene Skepner, Terry Jones, Howard Gutowitz, and Scott MacKenzie. Linguistically Optimized Text Entry on a Mobile Phone. *In Proceedings of CHI 2001, 19th International* Conference on *Human Factors in Computing Systems*, Seattle, Washington, USA, March 2001.

[17] Miika Silfverberg, I. Scott MacKenzie and Panu Korhonen. Predicting Text Entry Speed on Mobile Phones. *In Proceedings of CHI 2000, 18th International Conference on Human Factors in Computing Systems*, pages 9–16, Hague, The Netherlands, April 2000.

[18] Elena Sánchez-Nielsen, Luis Antón-Canalís and Mario Hernández-Tejera. Hand Gesture Recognition for Human-Machine Interaction. Journal of WSCG, 12(1–3): 91–96, 2004.

[19] Vincent Spruyt, Allesandro Ledda, and Stig Geerts. Real-time Multi-Colourspace Hand Segmentation*. In Proceedings of ICIP 2010, 17th International Conference on Image Processing*, pages 3117–3120, Hong Kong, Hong Kong, September 2010.

[20] Cristina Manresa, Javier Varona, Ramon Mas and Francisco J. Perales. Hand Tracking and Gesture Recognition for Human-Computer Interaction. *Electronic Letters on Computer Vision and Image Analysis,* 5(3):96–104, 2005.

[21] Lode Hoste. Aligning Programming Paradigms with the Multi-Touch Revolution. Master's thesis, Vrije Universiteit Brussel, May 2010.

[22] Luke Churc. Introducing #Dasher, A continuous Gesture IDE, A Work in Progress. *In Proceedings of PPIG 2005, 17th* International Conference *of the Psychology of Programming Interest Group*, pages 227–241, Brighton, United Kingdom, June 2005.

# Appendix

| Timestamp | I'm actively seeking for newer kinds of hard-input interfaces | I liked the usability of the SpeeG interface | I believe the SpeeG interface has the potential to be more usable in future versions | Quality of the speech recogniser | Quality of the dasher interface | The push gesture is a good mean to start/stop the interface | The pointing gesture is a good mean to control the dasher input | I would replace my TV remote control with this interface today | I would replace my TV remote control with an improved version of this interface | I would replace my Wii/Xbox/PS3/Digib oard with this interface today | I would replace my Wii/Xbox/PS3/Digib oard with an improved version of this interface | Why would you (not) like to replace your current TV remote control with this interface? | Why would you (not) like to replace your current Wii/Xbox/PS3/Digib oard with this interface? | Why do you think this is (not) an improvement over existing speech technology | Why did you (not) like it? | Do you have any proposals to improve the usability? (controls, interface, performance, ...) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8/25/2011 11:23:14 | 5 | 4 | 5 | 3 | 3 | 5 | 5 | 4 | 5 | 3 | 5 | I think this technology has great potential in the future, but for the moment it is not optimised. It seems as for vector recognition s and a user expects everything to go real. But I do see myself using an upgraded version of this when I sit in my sofa watching television. The fact that you don't need a remote control will be a major advantage for future users (in a home environment). | Same answer as above | I really like the idea that you can pick the word you meant. I know that existing speech recognition softwhare is not 100% accurate and therefore this is not an issue. As You can see all the possible matches to the word you said and it is the user how decide which is the correct choice without any auto correct or predefined suggestions | I think it would be more handy if both the speed and the direction could be done by only the gesture, the aim I am pointing with when I want to gain speed. But maybe this is just a matter of habit. | - Controlling the uncertainty of the speech recogniser<br>- Possibility to have user-dependent training to improve recognition rates making the pointing more and more optionals. |
| 8/25/2011 11:28:35 | 4 | 4 | 5 | 3 | 3 | 5 | 5 | 5 | 5 | 4 | 5 | With practice this interface would help you type sentences or words you want to search on your T-V, without the annoying selection process they work with today | The way in which word selection is done in these devices is extremely annoying and time consuming. | This interface makes the writing of words fun by itself. It could even become a game. | No vector recognition of the body (if possible?)<br>Use of other speech program (dragon?) |
| 8/25/2011 11:32:58 | 1 | 5 | 4 | 5 | 4 | 5 | 5 | 5 | 5 | 5 | 5 | (+) With experience, should be fast and efficient<br>(+) Funny + killer demos :-)<br>(+) Easy controls<br>(-) Still in unstable state<br>(-) Possibly steep learning curve before being really efficient<br>(-) It takes more time to setup | (Same as TV remote) (wondering for the context of an in-game Speed= :-) ) | (+) Correctionsl -> Speech recognisers which works_?<br>(+) Zero recognition from multiple results for one word (people are not always aware of that)<br>(-) Heavy workload for the brain<br>(-) Hard to have really fluid input (at least in the beginning) | (+) Gamification of speech recognition<br>(+) Easy control<br>(-) Heavy workload for the brain<br>(-) I'm worried about the workload in long sessions of use.<br>I liked the fluent motion of the interface.<br>It took some getting used to, but soon it would become a second nature. | The control of the speed can be a bit unstable after a while, maybe it would be cool to put it to some place where your hand doesn't need to be raised all the time. |
| 8/25/2011 11:43:14 | 2 | 3 | 5 | 2 | 4 | 4 | 4 | 3 | 4 | 3 | 4 | | | | | Better components, buy licenses :-) |
| 8/25/2011 11:42:27 | 5 | 4 | 5 | 2 | 4 | 5 | 5 | 3 | 4 | 3 | 4 | | | | | - Possibility to pause (and even get back to a previous word)<br>- Better recognition of the state of spinns (I have X proposals)<br>- The "W" to end words is counter-intuitive<br>- Needs easy setup to be really setup |
| 8/25/2011 11:44:51 | 3 | 4 | 5 | 4 | 3 | 5 | 5 | 3 | 4 | 3 | 4 | | | | | Maybe add a gesture to mute the mic and stop the system. In case you need to speak to someone else. |
| 8/26/2011 | 2 | 4 | 5 | 3 | 4 | 5 | 5 | 3 | 5 | 3 | 4 | | | | | |