Building a Multimodal Text Editor

Léandre Dubey, Hannah Portmann, Diogo Rocha

Introduction

Our idea for a multimodal interface was to build a text editor that only uses gaze and speech as input modalities. This interface could particularly be helpful for people with tetraplegia or other motor disabilities who cannot type text on a keyboard, as the hands would not be needed for input.

Our interface utilizes eye-tracking and speech recognition to write and format text. The idea was for users to select a text segment by looking at it and then issue commands via speech to perform actions such as bolding or deleting. The application's output includes written text and auditory feedback, enhancing the user experience. The multimodal human-computer interaction is illustrated in Figure 1.

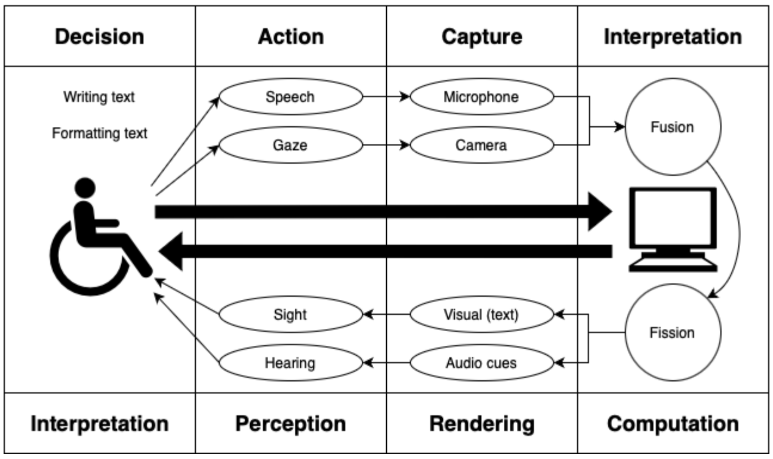

Figure 1. *Multimodal interaction with our application. (inspired by [1])*

Our Application - Methods and Modalities

Multiple resources were used to build the different parts to develop the application before fusing them all together. The text editor is based on a text editor found on codepen.io, adapted for our purposes [2].

Speech Recognition

Speech is one of the main modalities of our interface. The principle of speech recognition is that a model takes speech sounds as input and converts them into text. That transcription can then be used to interpret commands and obtain content that can be outputted to the text editor.

When looking into functional libraries for speech recognition, the one that seemed most promising was the speech recognition library from Python. It allows intuitive and automatic loading of a multitude of pre-trained models and gets them to recognize speech directly from the computers' microphone as a source.

A typical limitation of speech recognition is accuracy. Indeed, one of the most popular locally runnable models is pocket-sphinx, which, although lightweight, suffers from low accuracy. However, due to recent advancements in deep learning and new architectures such as self-attention transformers [3], we could access a recent model from OpenAI called "Whisper" [4], an encoder-decoder transformer that translates log-Mel spectrograms into text. This model provides much higher accuracy than pocket-sphinx while having a similar runtime.

The speech recognition part of the project currently works in two steps:

1. Recognize speech with Whisper: This is a straightforward step where we just need to initialize the microphone and recognizer, feed the sound from it to Whisper through *recognize_whisper*() with the language set as English, and get the text as an output.
2. Command identification: We go through all words inside the recognized text, strip them of special characters, and compare them to our list of keywords. When a word is recognized, it is appended to the command list, and everything after it is set as the content. If the recognized word is "type," then the rest of the text is set as content and not checked for commands anymore. This allows for multiple commands to be recognized, which greatly enhances the usability.

Eye Tracking

The second modality of our interfaces uses face recognition, eye tracking, and prediction models to map the gaze position to coordinates on the screen. To be widely accessible, we decided to use WebGazer [5], which allows us to easily add eye tracking to any web application. WebGazer works by capturing live video feeds from the computer camera. A computer vision algorithm is then applied to the video feed to recognize and track the face and the different facial parts. Additionally, it creates a face mesh (3D representation) by mapping different points/landmarks on the face. WebGazer extracts features of the eye region (pupil and eye position) using the face mesh and uses these features to train a gaze estimation model that maps the 3D eye features to 2D coordinates on the screen. This estimation model can be improved with calibration using error minimization. Even with calibration, the accuracy of the eye tracking is our project's most significant limitation. This is likely due to moving head position or low camera resolution.

The eye-tracking part of the project currently works in two steps:

1. Calibrate the eye tracking by looking at points on the screen and clicking on them. This considerably improves the accuracy of the prediction model.
2. Eye tracking computes online mapping of the gaze estimation to the coordinates on the computer screen. Thus, the coordinates are constantly updated to match where the user is looking on the screen and made available to use with commands.

Fusion

The speech recognition model runs through Python using the 'SpeechRecognition (3.10.4, https://pypi.org/project/SpeechRecognition/) library, while WebGazer (3.3.0, https://webgazer.cs.brown.edu/) and the rest of the HTML logic operate through JavaScript. These components are integrated and served through a Flask (v3.0.3, https://pypi.org/project/Flask/) app in Python, which manages the server-side operations. The diagram in Figure 2 illustrates the architecture of the application, showcasing how the various components interact with each other.

The 'main.js' file is the central coordinator of the app, integrating various functionalities such as speech recognition, gaze tracking, and user input handling. It initializes the WebGazer library, setting up the gaze predictions and event listeners for user interactions. It continuously monitors the user's gaze coordinates and triggers the HTTP request for speech recognition when the gaze prediction coordinates hover over the recording button for three seconds.

Once recognized, the speech is processed and sent to 'main.js' as JSON format, containing keys 'commands' and 'content', or an error message if the recognition fails. This output from the API is managed in conjunction with the gaze prediction coordinates. If the users's gaze is focused on a paragraph, a *for loop* iterates through all recognized commands, mapping and executing the corresponding functions on that paragraph.

With this architecture in mind, we can conclude that our fusion is done at the decision level since we combine the output of speech recognition and eye tracking.

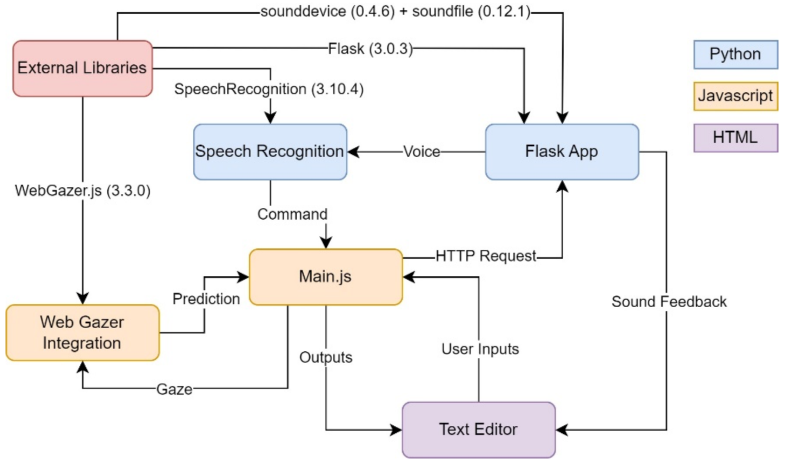

Figure 2. *The software architecture of our application.*

Fission and Error Handling

The application produces visual and auditory feedback. Once command handling terminates (if successful), the user can observe its effects through changes on the displayed text of the selected paragraph (either input of new text or modifications on existing text such as size, bold, italic, or underline). The auditory feedback comprises three different sounds played with different conditions: 1) A cue_in sound is played when speech recognition is ready to record user speech. 2) A cue_out sound plays at the termination of speech recognition. It indicates to the user that speech input is not being recorded anymore. 3) A dedicated error sound feedback is played when no command is recognized from user input. The audio is managed through flask with the sounddevice (0.4.6, https://pypi.org/project/sounddevice/) and soundfile (0.12.1, https://pypi.org/project/soundfile/) libraries.

Users commonly encounter two main errors while using our application: unrecognized commands and gaze prediction missing the intended paragraph. If no command is recognized, the speech recognition stops, and an error sound is played to alert the user. For out-of-bounds gaze predictions, the recording restarts automatically, and a cue_in sound is played to signal the user to try again. This process allows users to reissue commands without having to refocus on the recording button.

Functioning of Our Final Application

Figure 3 shows the final interface of our application, and Figure 4 shows schematically how the application works. To start speech recognition, the user has to look at the recording button for 3 seconds until the cue_in sound is played. Then, the commands can be spoken while the user is looking at the paragraph where they want the commands to be performed until the speech is recognized and cue_out is played.

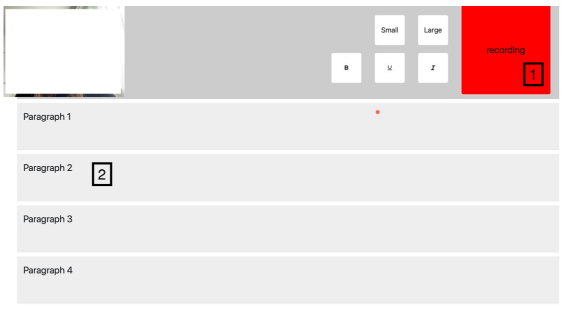

Figure 3. *User interface of our application. The numbers show important gaze positions: (1) Recording button, (2) Paragraph box.*

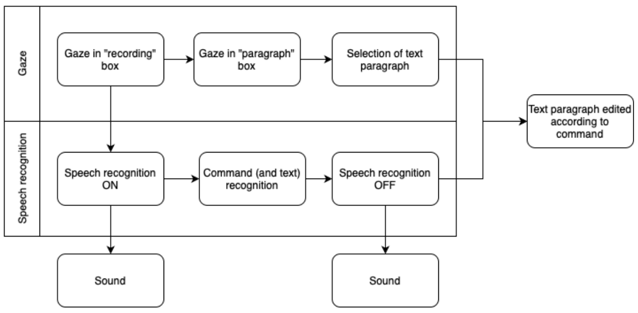

Figure 4. *Schematic explanation of how our application functions, including both input modalities and the feedback/output.*

Table 1 summarizes the commands currently implemented. The commands can either be used by themselves ("strong") or be used in a sentence ("Put this paragraph in strong"). Commands can also be used together ("Put this in big and underline"). Type should always be used as the last command as the text after it is typed. Formatting should only be applied to the final text, as it will be removed if something new is typed.

Table 1. *Commands to use in our application.*

COMMAND	PERFORMS
TYPE (+TEXT)	*Write* the text in the paragraph
DELETE	*Delete the whole* paragraph
STRONG	Put in *bold*
ITALIC	Put in *italic*
UNDERLINE	*Underline* the text
SMALL	Change size to *small*
BIG	Change size to *big*
STRIP	*Remove all formatting*

CARE/CASE Models

There are two models to describe the multimodality of an application. The CASE model specifies the multimodal communication types on the machine side. The use of modalities in our application is sequential as well as parallel. It is sequential, as we first need to look at the "recording box" to start speech recognition. After this first step, the modalities are used in parallel, as a command needs to be given by speech, and at the same time, the paragraph to which the command should be applied needs to be looked at. The fusion of our modalities is combined, as fusion is needed to apply a command given by speech to a paragraph that was selected by gaze. Our application implements this by having a function that takes the command given as input, which then selects a further function that takes the paragraph that was looked at as input. The levels of abstraction used in the application are quite processed already. For speech recognition, we directly use the recognized words. We use the predicted x- and y-coordinates on the screen to recognize the gaze. Considering all this, our application would be synergistic according to the CASE model. To perform a task such as writing a sentence in a specific paragraph box, both modalities are used in parallel and coreference each other.

The CARE model specifies the multimodal system usability properties on the human side. Our application has two of those properties: Complementarity and Assignment. To activate speech recognition, only gaze can be used. Therefore, there is an absence of choice, which indicates the Assignment property. Then, to write or edit text in a specific paragraph box, both speech and gaze are needed within a temporal window. None of the modalities by itself is sufficient to do this, which indicates the Complementarity property.

User Evaluation

Idea and Hypothesis

In this user evaluation, we compared using our multimodal interface for text editing to dictating text changes to another person, who would then execute them in a standard text editor. This evaluation approach was chosen with the primary user group in mind - paralyzed individuals who cannot type independently. The two conditions were compared using a within-group design. Our hypothesis was that our application would be at least as fast as dictation while offering higher user satisfaction.

Methods

Participants were first instructed on the context, imagining themselves as tetraplegic individuals needing these systems. They were given detailed written instructions for the functioning of both methods before testing them. Condition 1 involved using our application, while Condition 2 involved dictating to another person. Seven participants took part in the evaluation, each testing both conditions. Three participants started with Condition 1 and four with Condition 2.

In both conditions, participants were tasked with editing the same provided text containing four paragraphs as follows:

- P1: Changing text to bold, underlined, and large (“strong, underline, big”)
- P2: Removing bold, italic, and underline formatting (“strip”)
- P3: Deleting existing text and typing a given new text (“delete, type + text”)
- P4: Changing text to small and italic (“small, italic”)

The time taken to complete each paragraph was recorded.

After each condition, participants completed a questionnaire with five questions, which they had to answer with a number from 1 (not so much) to 5 (very much). The questions were the following:

- Q1: How independent did you feel?
- Q2: How much fun did you have?
- Q3: How likely are you to use it again in the future? (imagining they could not type themselves)
- Q4: How accomplished did you feel?
- Q5: How easy was it to navigate and use the application?

One participant with incomplete tetraplegia provided additional detailed feedback. Although she can use her hands, she experiences pain and difficulty typing long texts and currently uses Microsoft Word with dictation to write and edit text.

Data were processed using R Studio, calculating each user's total time and average questionnaire responses. Statistical tests included Shapiro-Wilk tests for normality and either paired t-tests or Wilcoxon tests, depending on whether the data were normally distributed.

Results

Figure 5 shows the total time means for both conditions in seconds. There is a significant difference between Condition 1 ($M = 263.57$, $SD = 63.43$) and Condition 2 ($M = 122.00$, $SD = 37.07$), with Condition 1 taking significantly longer; $t(6) = 7.74$, $p < 0.001$

Figure 6 illustrates the means of the average questionnaire answers, representing overall user satisfaction, for both conditions. The difference between Condition 1 ($M = 3.49$, $SD = 0.65$) and Condition 2 ($M = 2.86$, $SD = 0.71$) is significant, with higher satisfaction reported in Condition 1; $t(6) = 4.96$, $p < 0.01$

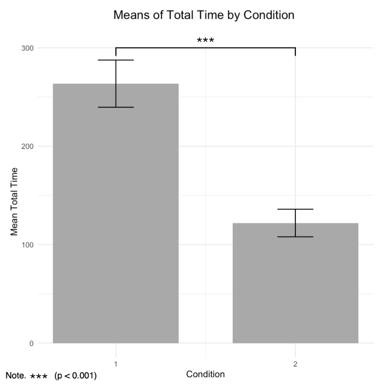

Figure 5. Mean total time to edit the text by condition.

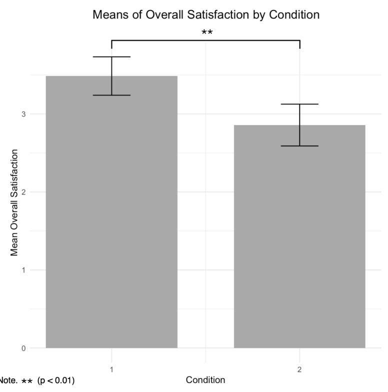

Figure 6. Mean overall user satisfaction score by condition.

As these differences were significant, we also tested the time for each paragraph and the answer to each question individually to detect where exactly the most prominent differences lie.

Figure 7 shows the average times for paragraphs 1 to 4 for both conditions. It can be observed that the average times for all four paragraphs were shorter in Condition 2. Significant differences were found in:

- Paragraph 1: Condition 1 (M = 79.71, SD = 56.43) vs. Condition 2 (M = 17.43, SD = 5.94), t(6) = 2.80, p < 0.05
- Paragraph 4: Condition 1 (M = 38.71, SD = 25.77) vs. Condition 2 (M = 11.86, SD = 2.27), t(6) = 2.64, p < 0.05

However, no significant differences were found in:

- Paragraph 2: Condition 1 (M = 32.71, SD = 28.03) vs. Condition 2 (M = 24.43, SD = 18.37), V = 17, p = 0.207
- Paragraph 3: Condition 1 (M = 112.43, SD = 78.07) vs. Condition 2 (M = 68.29, SD = 24.97), t(6) = 1.86, p = 0.112

Figure 8 shows the mean scores for each question in the questionnaire for both conditions. The largest and only significant difference between Condition 1 (M = 3.86, SD = 0.90) and Condition 2 (M = 1.29, SD = 0.49) was found in question 1; V = 28, p < 0.05

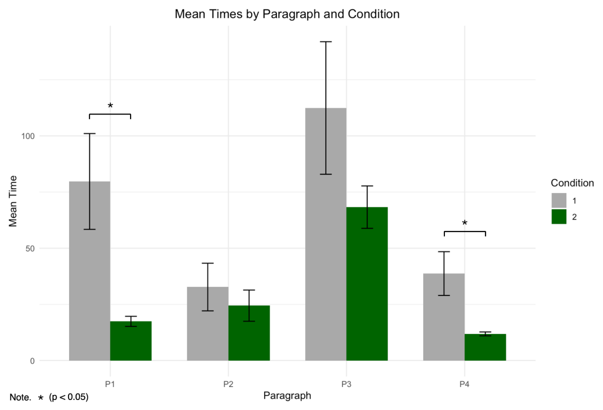

Figure 7. Mean times per paragraph by condition.

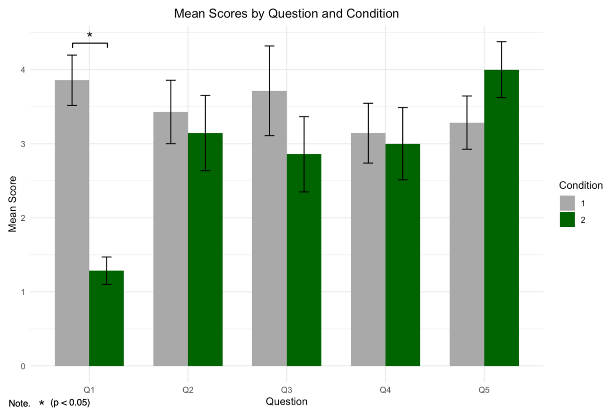

Figure 8. Mean scores per question by condition.

No significant differences were found for the other four questions:

- Question 2: Condition 1 (M = 3.43, SD = 1.13) vs. Condition 2 (M = 3.14, SD = 1.35), V = 9.5 , p = 0.679
- Question 3: Condition 1 (M = 3.71, SD = 1.60) vs. Condition 2 (M = 2.86, SD = 1.35), t(6) = 1.87, p = 0.111
- Question 4: Condition 1 (M = 3.14, SD = 1.07) vs. Condition 2 (M = 3.00, SD = 1.29), V = 7.5, p = 1
- Question 5: Condition 1 (M = 3.29, SD = 0.95) vs. Condition 2 (M = 4.00, SD = 1.00), V = 5, p = 0.284

The participant with incomplete tetraplegia reported that our application would be very useful to people with complete tetraplegia who do not have any motor abilities in the hands and also to her if it were improved. She would also use it if it were publicly available and the accuracy of the eye tracking was improved. However, it would need to be possible to select individual words to correct errors that might occur.

Discussion of Results

Our hypothesis was not supported by the results. Although user satisfaction was higher with our application, the time taken to edit text was significantly longer than dictating changes to another person.

The considerable time difference in the first paragraph was likely due to participants' unfamiliarity with the interface, as they had never used it before the evaluation. This probably impacted the times of the other three paragraphs as well, but the effect is most prominent in the first paragraph. To check this assumption, we compared the mean total time of the three developers, who are very familiar with the interface (117.00 s) with the participants' dictation condition time (122.00 s) and found them to be more similar than the participants' mean time using our interface (263.57 s). This suggests that familiarity with the interface significantly reduces editing time.

As for the significant mean difference in paragraph 4, we observed that recognition of the paragraph by gaze tracking was less effective than in other paragraphs. Unfortunately, we currently do not know the cause of this.

The questionnaire results indicated that participants felt more independent using our interface, which was to be expected. The only higher mean for Condition 2 was question 5, describing ease of use, which is understandable since telling another person what to do can be much easier than using a new interface where specific keywords must be memorized, and gaze has to be precisely controlled.

The main issues encountered during the evaluation were that people did not realize when the speech recognition was on vs. off, forgot the correct commands, or that commands were not correctly recognized, as none of the participants were native English speakers, and most had a minor accent. Furthermore, eye-tracking was often not accurate enough.

Limitations and Potential Improvements

The main limitation is the relatively poor eye-tracking accuracy. Indeed, WebGazer is quite limited even with calibration. Since the commands are useless without the prediction given by eye-tracking, the app could greatly benefit from migrating to GazeCloudAPI or using dedicated equipment like eye-tracking glasses.

The speech recognition for keywords is imprecise, and keywords are often not understood/misunderstood. We could try to add more similar words that are recognized to activate the same function.

Speech Recognition works correctly in general, but performance can be somewhat unstable at some points for unknown reasons. One reason could be unusual pronunciation from non-native English speakers. To improve speech recognition, we could explore other recognizers, such as Google Cloud Speech API, and add error handling solutions, such as a lightweight LLM that autocorrects transcribed speech.

Users can only select entire paragraphs to delete, format, and add text. Selecting individual words is not yet possible in our application and would be impossible with the poor accuracy of eye-tracking.

The performance of our application clearly rises for people familiar with it, indicating that some learning time is needed to benefit from the application fully. To make using the application a bit easier and more intuitive, we could add more keywords that could be used to describe the same thing, such as “type” and “write”.

Conclusion

Our project aimed to create a multimodal user interface that could actually have a practical application. For that purpose, we developed a text editor that would allow users to input text and format it by interacting only with your voice and eye gaze so that it could be used autonomously, for example, by people with tetraplegia. We successfully fused a voice recognition model (Whisper) with an eye-tracking library (WebGazer) to provide commands to a web-based text editor. We then asked participants to test our application against simple dictation in an experimental setting. This allowed us to shed light on some of the strengths and limitations of our application. Mainly, speed limitations arose from errors due to poor accuracy of the tools (mainly the eye-tracking) used before fusion and from the time needed to get familiar with the environment. The strengths lay in the quality of the speech recognition for long text inputs, as well as in the pleasure felt during the use of our application. In conclusion, the application was of significant interest and seemed very promising. With an extended development time, it could strongly benefit from more accurate input tools and added features.

https://github.com/DiogoRoch/SR_Gaze_TextEditor

References

[1] B. Dumas, D. Lalanne, and S. Oviatt, “Multimodal Interfaces: A Survey of Principles, Models and Frameworks,” in *Human Machine Interaction: Research Results of the MMI Program*, D. Lalanne and J. Kohlas, Eds., Berlin, Heidelberg: Springer, 2009, pp. 3–26. doi: 10.1007/978-3-642-00437-7_1.

[2] F. Nur Wahid, “Text Editor.” 2021. Accessed: Mar. 16, 2024. [Online]. Available: https://codepen.io/fajarnurwahid/pen/NWvxeXj

[3] A. Vaswani *et al.*, “Attention Is All You Need.” arXiv, Aug. 01, 2023. doi: 10.48550/arXiv.1706.03762.

[4] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, “Robust Speech Recognition via Large-Scale Weak Supervision”.

[5] A. Papoutsaki, P. Sangkloy, J. Laskey, N. Daskalova, J. Huang, and J. Hays, “Webgazer: scalable webcam eye tracking using user interactions,” in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, in IJCAI’16. New York, New York, USA: AAAI Press, Jul. 2016, pp. 3839–3845.