Multi Modal User Interfaces

Put What Where

Report

Mathias Øgaard, Tobias Verheijen & Xipeng Wang

31 May, 2024

Contents

1 Introduction

The main focus of our project is to create a simplified modern rendition of the *Put That There* projecct[1].

Put That There's objective was to create a Multi Modal Interface which accepted speech and gesture recognition as inputs and had a visual display. The user could draw, move, change colors of, and remove objects from a screen using a synergistic combination of speech commands and pointing gestures.

Our project uses this as a baseline and attempts to bring it to the modern era of computer science using improved speech recognition capabilities and eye tracking as command inputs [2][3]. We implement this multi modal interface using the python programming language and other python modules. Our approach maintains the synergistic quality of *Put That There* and combines modalities in an admittedly user-unfriendly environment.

Put What Where is a fairly technical project, and may not have an immediately obvious real-life application. However, we believe that fusion between eye gaze and speech could significantly benefit users with hand-related disabilities, i.e. users for whom it might be difficult to interact with a keyboard, or to make hand gestures as in *Put That There*.

Our report is outlined as follows: We first provide a description of the project concept and software, then we outline the human-computer interaction interface. We follow this with a brief summary of the model's position in the *Case* and *Care* framework, and we will close with results from the accuracy of our software and comparisons to alternative modalities.

2 Project Concept

As mentioned above, our project seeks to extend the *Put That There* project and provide a similar environment with eye tracking replacing gesture recognition. We provide the user with some key commands to have some behind-the-scenes short cuts as well. The project uses a combination of seven `ACTION` words:

- Select
  - *select* the quadrant which the user is looking at
- Go Back
  - *deselect* the most recently selected screen quadrant
- Collect
  - *pick up* an object in the specified location

- Pick Up
  - *pick up* an object in the specified location (identical to *collect*)
- Release
  - *drop* an object in the specified location
- New Object
  - place a *new* object in the specified location
- Delete
  - *delete* an object in the specified location

The project makes use of a PyGame interface as the main running body and GUI [4]. More information about the structure of our project can be found in section 2.1. The user is able to interact with the display and embedded objects such as circles and squares.

2.1 Software Architecture

The main architecture of our model is visualizable in figure 1. We implemented

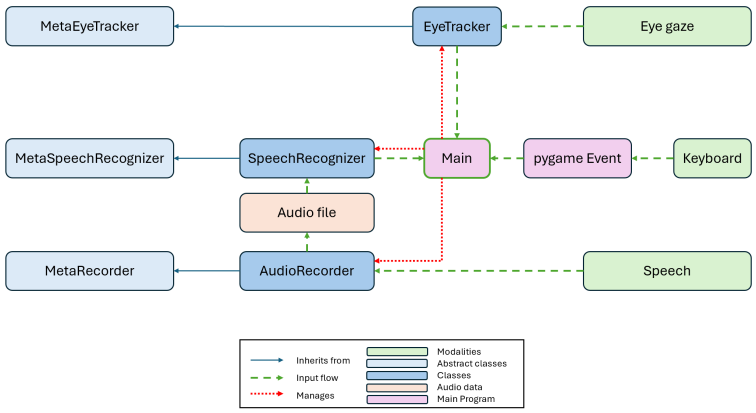

Figure 1: Software Architecture Diagram

a few `MetaClasses`, which are Object Oriented programming inspired with a few core methods that the implemented versions must also contain. This is a key detail in the ability to change between different modules depending on the hardware you have available. The idea was to keep the actual processing of multimodal inputs encapsulated, so that others could easily implement their own classes for eye tracking, recording and speech recognizing without affecting the main program.

To explain figure 1 in more detail, we implement custom instances of our meta classes. We use the standard *pygame event loop* to handle events triggered by the keyboard. To keep the program responsive to commands and speed up processing, we require the user to press the space bar to start the audio recognition and eye tracking actions. Upon pressing the space bar, the program sends a signal to the `EyeTracker` and the `AudioRecorder`. The `EyeTracker` will track the user's gaze position while the `AudioRecorder` will write to an audio file and send a flag back to the main event loop upon completion. Main will then tell the `SpeechRecognizer` to start recognizing the audio file to check for compatibility with any of the available user commands. In the following sections (2.2 & 2.3), we will describe these three classes in more detail.

2.2 Eye Tracking

Our initial plan was to be able to do live eye gaze tracking. Using a laptop webcam, this turned out to be more difficult than expected, so we changed our strategy. Our new solution was to split the screen into four quadrants. Our `EyeTracker` would then be used to detect which quadrant the user is looking at in a given moment.
Once a quadrant is selected, this is again split into four subquadrants, giving us 16 subquadrants in total (see figure 2). These make the set of valid positions in our program, in which objects can be put.

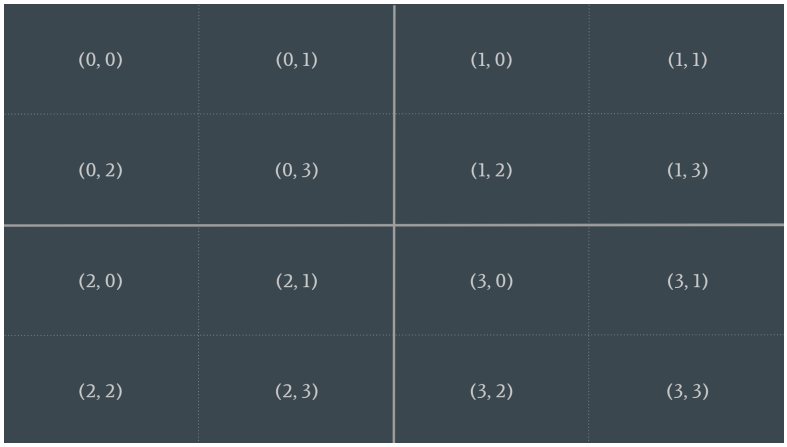

Figure 2: Grid of valid positions for objects

For the implementation of our `EyeTracker`, we used code from a repository called GazeTracking [3]. GazeTracking provides a way to take a picture of a face, analyze it, and ultimately tell where the eyes are looking along two axes: up/down and left/right. The methods where initially assuming the cen-

ter/origon to be when a pupil was placed in the center of the visible eye. However, since we are interested in where the user is looking at the screen, we tweaked some values to align the central position with the middle of the screen. That gives us two important methods: `gaze.is_up()` and `gaze.is_right()`. See figure 3 for how these are used to detect what quadrant the user is looking at.

```
#Check where user is looking
self.gaze.set_right()
self.gaze.set_up()
#Update chosen quadrant
up = self.gaze.is_up()
down = not up
right = self.gaze.is_right()
left = not right

if up and left:
    self.quadrant = 0
elif up and right:
    self.quadrant = 1
elif down and left:
    self.quadrant = 2
elif down and right:
    self.quadrant = 3
```

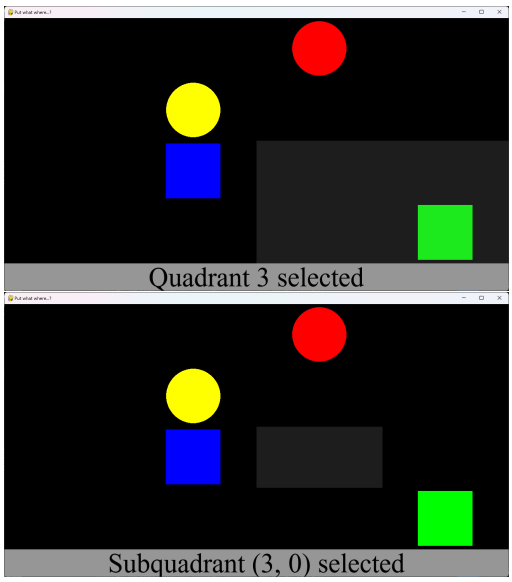

Figure 3: Methods from GazeTracking are used to select a position on the screen

2.3 Audio Processing

Initial versions of our program would have the `SpeechRecognizer` class handle both audio recording and speech recognition. This was mostly functional, however turning this into a parallel process was rather challenging as audio information would quickly become corrupted and recognition times took much longer. This inspired us to create an `AudioRecorder` class which handles the audio recording as its own process.

Separating audio recording was key to improve recognition speeds. It also ensures that the program will not try to start a second recording process as we do not allow for secondary audio file generation while recording is taking place. The audio recorder writes to a `.wav` file while the user holds the space bar down. Once the user is content with their recording, they can release the space bar which calls the `AudioRecorder.finish_recording()`. This signals for the back end of the `AudioRecorder` to stop recording and joins the thread into the main thread to wait for the process to finish.

After the `AudioRecorder.finish_recording()` function is called, the main loop knows that the desired audio file exists and initializes the `SpeechRecognizer` with the file as its audio source. Upon completion of the `SpeechRecognizer` task, the message is retrieved and displayed to the user. Before running through the pygame event loop again, we check if the retrieved message is within our verified list of `ACTIONS` defined in section 2. If the action is recognized it will be performed immediately afterwards. If the action is not recognized, no harm is done and the event loop begins anew.

For our `SpeechRecognizer`, we used the python library `speech_recognition`, which contains multiple models for recognizing speech. However, only of them did not require an API key - Google's free online model - which became the base technology of our `SpeechRecognizer`.
Our hypothesis is that a more up-to-date model, preferably with the option to give recognition biases towards certain keywords, would improve the performance significantly compared to our current implementation.

3 Human Interaction and CASE/CARE

Human interaction is vital to the functionality of our application. The user is able to control the environment with a synergistic combination of speech recognition and eye tracking. As described in section 2.2, the eye tracking is unfortunately segmented into quadrants and sub-quadrants on the screen. This means that the truly synergistic command style from the original *Put That There* project where gestures controlled the position of the mouse is no longer a viable control mechanism. However, we are still able to fuse the speech recognition and eye tracking modalities just in a one-step-separated way. Through indicators supplied by the user's key presses we are able to have the program start the correct processes and inform the user of what the program is doing.

In addition to the responsive text display, the highlighting of the current selected quadrant always keeps the user in the *driving seat*, with the option to correct any potential misunderstandings from the program (with the voice command *"go back"*).

We will now categorize our project within the CASE-CARE framework, which is a useful way to define an application's use of different modalities, and their interrelations.

3.1 CASE

As mentioned in previous sections, our *Put What Where* application fits into the *synergistic* framework, as we are able to use eye tracking and speech recognition at the same time. While the program is recording audio, it is also tracking

the user's eye position. The commands also combine the information of both modalities. When the `'select'` command is spoken, the program takes the user's eye position as the positional input for the next phase of the application. This means that the user is providing both inputs simultaneously and the information from both modalities complement/rely on each other.

3.2 CARE

The CARE portion consists of four descriptors for the program's action options: Complementary, Assignment, Redundancy, Equivalent.

In the complementary lens, speech recognition and eye tracking complement each other to provide for the ability to select quadrants of the screen and move objects. We assign what the user is selecting to the eye tracking modality, whereas the command inputs are assigned to the speech recognition module. In order to provide the user with some faster options, we have also included a redundant set of commands tied to different key presses from the user. This means that they are able to do similar actions. In fact, the keyboard inputs are able to trigger eye tracking and all other speech recognition actions. Thus the keyboard and speech recognition are almost equivalent modalities, and the keyboard can be used in place of speech recognition as a modal option.

4 Results

To test the quality of our program we thought it was best to test the accuracy of the individual modalities. The speech recognition module required a couple of rounds of testing to include the use of some functionalities included in the package.

The recognizer is able to correct for ambient noise to fine-tune the performance of the recognition. We evaluated a number of paired t-tests on recognition vectors for each of our key words. Our alternative hypothesis for these tests were that the ambient noise correction would make the accuracy of the speech recognition model improve. We tested the accuracy of the speech recognizer by repeating the action commands 15 times and writing a 0 if the recognizer missed, and a 1 if the recognizer hit. This was first done without the ambient noise correction, and then repeated with the ambient noise correction. The results can be seen in figure 4.

As we can see, there is a statistically significant increase in accuracy after the ambient noise correction as indicated by the * above the *average* column of the histogram. We have also included a boxplot demonstrating how long recognizing the audio file takes once processing has begun as seen in figure 5.

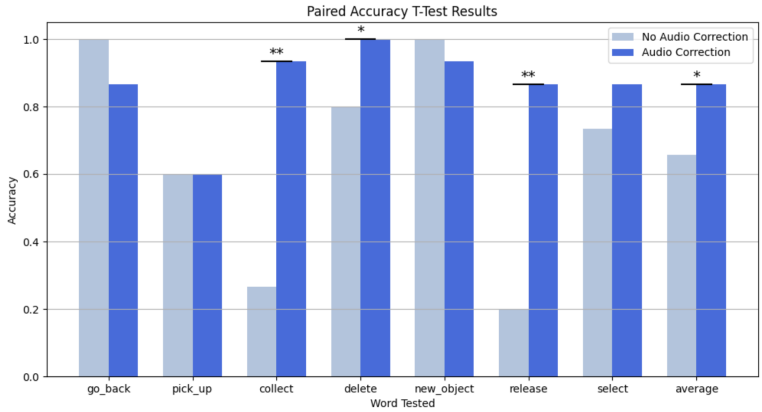

Figure 4: Word Recognition Histogram

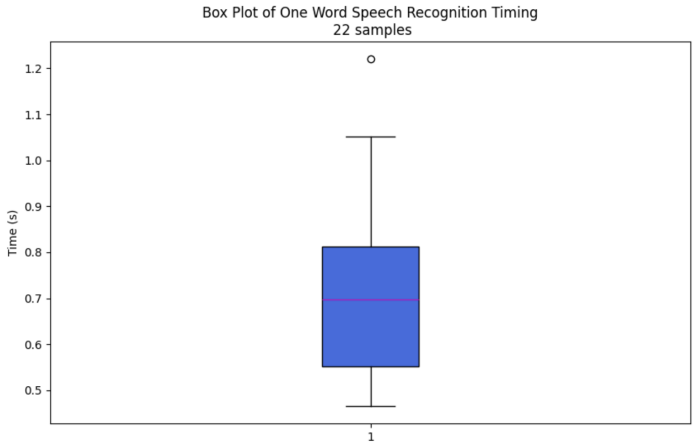

Figure 5: Speech Recognition Timing

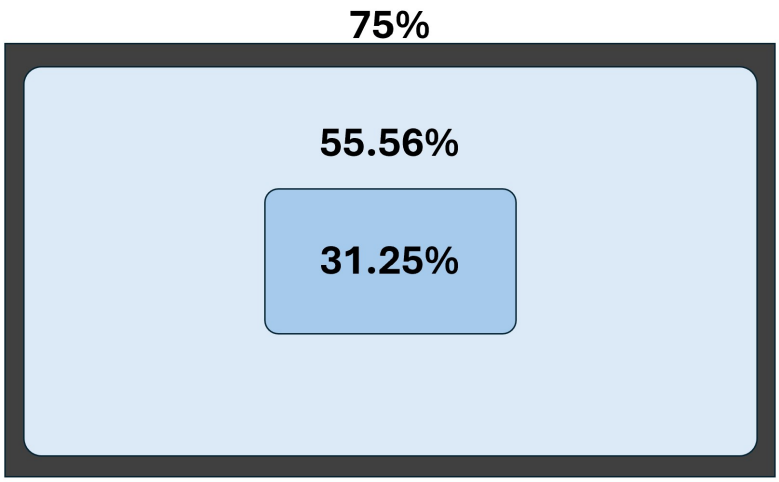

Figure 6: Eye Tracker Accuracy by Screen Region

It was also important for us to test the accuracy of the simplified eye-gazing approach we adopted. To achieve this, we developed a modified version of the program that displayed circles at various positions within our quadrant grid. The user was instructed to look at the displayed circle and trigger the `EyeTracker`, while the test script recorded the number of hits and misses.

In figure 6, we observe the accuracy distribution based on what screen region the user is looking at when selecting a quadrant. When closer to the center, the model is wrong quite frequently, so we urge the user to look at the corners of their computer screen when selecting quadrants for a better experience.

5 Conclusion

In the end, we did achieve a synergistic fusion of eye gaze and speech, which was the main goal of our project.

While testing the program, we found that with some training, users can quite quickly learn to use their eye gaze as an effective form of input. As mentioned in section 4, this works best if the user is looking at the outer edge of the screen, rather than the middle.

The program's speech recognition side, however, is quite slow and inaccurate. As mentioned in section 2.3, we think this could be improved by implementing a `SpeechRecognizer` with a more up-to-date technology.

Regarding the human interaction aspect, the program performs well in educating the user on how he/she should interact with it. The highlighting of the current selected quadrant, together with a responsive and informative text display, helps the user understand the ongoing processes of the program. The user is always in control, and can either correct ("go back") or verify the program's understanding of the modal input.

References

[1] R. A. Bolt, "'Put-that-there': Voice and gesture at the graphics interface," in *In Proceedings of the 7th annual conference on Computer graphics and interactive techniques (SIGGRAPH '80).*, ACM, New York, NY, USA, 262-270, 1980.

[2] A. Zhang, "Uberi/speech_recognition," *GitHub*, [Online]. Available: `https://github.com/Uberi/speech_recognition#readme`.

[3] A. Lamé, "antoinelame/GazeTracking," *GitHub*, Antoine Lamé under the terms of the MIT Open Source License. [Online]. Available: `https://github.com/antoinelame/GazeTracking`.

[4] P. D. Team, *PyGame*, `http://pygame.org/`, Open Source project distributed under GNU LGPL version 2.1, 2024.