BCI user interface

Marlon van Zijl & Chelsea Apawti

Delft University of Technology

TU Delft

BCI user interface

by

Marlon van Zijl & Chelsea Apawti

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Monday June 27, 2023 at 11:00 AM.

Project duration:	April 24, 2023 – June 15, 2023	
Thesis committee:	Prof. dr. B. Hunyadi,	TU Delft, supervisor
	Dr. G. Joseph,	TU Delft

Cover: https://www.allerin.com/wp-blog/wpcontent/uploads/2019/08/risks-of-brain-computer-interfaceapplications-of-brain-computer-interfaces.jpg
Style: TU Delft Report Style, with modifications by Daan Zwaneveld

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

This document presents the development of a user interface for an EEG motor imagery based Brain-Computer Interface (BCI) as the interface subgroup. The aim of this subgroup in the project was to design and implement a graphical user interface (GUI) incorporating visual neurofeedback [1] to enhance the accuracy of the decode algorithm, developed by the other subgroup, during the calibration/training stage for the user such that the user will have better motor imagery control using the GUI. The GUI features two interactive games, namely pong and breakout, and incorporates topographic maps displaying the user's brain activity. The primary goal of these maps was to improve the training process of the algorithm for each individual user. Additionally, the thesis explores the feasibility of incorporating steady-state visually evoked potential (SSVEP) elements in the games or for creating a new game that combines motor imagery and SSVEP elements [2] allowing for more complex games to be played thus positively affecting the user experience.

Preface

This Bachelor thesis serves as a final graduation project in which the members of the group were tasked to develop an electroencephalogram (EEG) based brain computer interface (BCI). Our subgroup specifically was tasked with exploring the possibilities of user interfaces utilizing neurofeedback to potentially improve the performance of the BCI system as well as create a fun and engaging environment for the user. In this thesis, we will describe the design and implementation of a graphical user interface that includes the ability to play two games providing neural feedback in the form of visual stimuli. Additionally, topographic mapping displaying the user's brain activity is explored as well as research into implementing SSVEP elements for future further development of the BAP project in order to positively affect user experience.

We would like to thank our main supervisor prof. dr. B. Hunyadi and also prof.dr.ir. L. Abelmann for giving his insight and advise during the project. Additionally we would like to thank the PhD students who explained certain concepts as well as the staff of the Tellegen Hall, with a special emphasis on ing. M. Schumacher who played the role of test-subject for numerous tests throughout the project.

We hope that this project will be picked up and further developed after completion of this thesis, since it is a very interesting subject.

Contents

1 Introduction

A Brain-Computer Interface (BCI) is a system that connects the brain with a computer and allows for direct communication between the two [3], [4]. In Figure 1.1, a BCI is represented as a schematic. The BCI works a follows: the user sets a mental intention, which creates a brain wave. The brain wave is detected and recorded by a measurement device with electrodes and is then sent to a computer. The acquired signal is then digitized and ready to be processed. In the signal processing, the signal is decoded and translated into a command signal that is then sent from the computer to the interface (effector). The interface executes the command it has received.

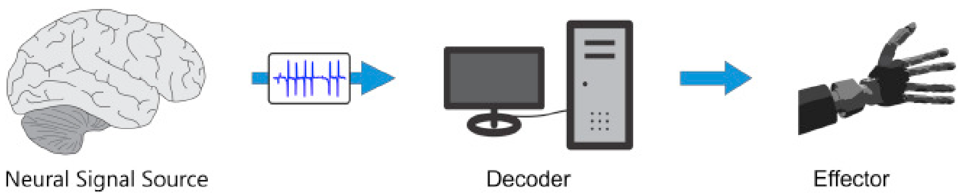


Figure 1.1: Schematic of a BCI [5]

Due to the setup of this system, it can be used for many different purposes. A lot of research has been done already which uses some type of BCI, especially within the field of neurology and with the use of electroencephalogram (EEG)-based BCI [6]. The research has resulted in many different applications for which BCI has been proven useful. The following applications have been discovered:

1. **Restoring neural functioning.** An example would be a stroke, which causes not only motor impairments but can also affect certain cognitive brain functions. [7] discusses stroke rehabilitation methods including a section on using BCIs to stimulate affected brain regions into self-recovery.
2. **Diagnosing neurological disorders.** Several neurological disorders can be diagnosed with the use of BCI such as epilepsy [8] and Parkinson's disease. However, other disorders and diseases such as brain tumors, ADHD and schizophrenia [9] can also be diagnosed using BCIs.
3. **Physical rehabilitation.** [10] discusses a BCI's ability to stimulate physical rehabilitation for patient with sensory-motor defects. They propose a method of combining exercise with video games by using the kinect sensor (movement tracking).
4. **Brain research**. Brains are largely undiscovered mysteries due their vast complexity. BCIs can be used to study brain activity and how certain processes are linked to which brain regions etc. The researchers from [11] for example, studied learning behaviour in monkeys through BCIs.
5. **Neuroprosthetics.** [12] showed the feasibility of controlling an upper-limb prosthesis through a BCI.

Even outside of medical research, BCI has shown to be very practical. While many applications have been come up with, BCI has especially shown to have a lot of potential as a control system. So far, BCI has been used to control robotic systems, vehicles, and video games [6]. However, many of these applications have been developed by using clinical-grade measurement equipment, which are quite different from the consumer-grade measurement equipment that is available. Until this day, the development of practical BCI applications using consumer-grade measurement equipment has been not been as extensive as with the use of clinical-grade measurement equipment, and therefore, there is still a lot of opportunity in that area of BCI development.

1.1. Problem definition

As mentioned previously, most BCI applications have been developed by using clinical-grade measurement equipment. This type of equipment is very impractical when it comes to using it outside of clinical settings. It requires very extensive and time-consuming setup, use and cleanup and it is expensive [13]. Furthermore, the equipment has many uncomfortable electrodes that need conducting gel and the system is stationary. Consumer-grade equipment have become more commercially available in the recent years and have the advantages of being lighter weight, wireless, cheaper and generally easier in use than its clinical-grade equivalent [14]. Whilst the consumer-grade equipment deals with issues such as high susceptibility to noise, research has already shown that consumer-grade equipment can still obtain ”fairly good quality EEG data” [15].

Today, the few consumer-grade measurement equipment on the market are either sold as merely a measurement tool or come as a part of a simple BCI. These BCIs are often for entertainment purposes. With this project, we hope to show that it is possible to create a BCI with consumer-grade measurement equipment that has high practical value. We want to do this by showing that the BCI works well with the interface that we create and consequently, the BCI system should work for applications such as rehabilitation and online mobility for the motor-impaired. This thesis will describe and justify the design process for the interface of the BCI.

1.2. Thesis outline

The thesis is structured as follows. In Chapter 2, the Program of Requirements is laid out. Chapter 3 explains the choice of the interface and the design plan created based on the requirements. Chapter 4 discusses the development of the games specifically and how the code written for the games is structured. Chapter 5 goes into the literature study and tests conducted in order to make well-informed decisions concerning the neurofeedback. In Chapter 6, some the steps taken towards the integration of the entire BCI are described. Chapter 7 provides the conclusion and the future work that has to be done. In the Appendices A-C, the Python scripts for the interface, topographic maps and the conducted tests can be found. Appendix D contains the research done into the possibility of including SSVEP elements in the games, with additional figures from the research located in Appendix E. Appendix F contains a timetable with the tasks performed during the course of the project.

2

Program of Requirements

The general requirement from the BAP proposal goes as follows: create an EEG based BCI that will allow a subject to control something. The general requirement for the interface subgroup has some degree of freedom to it: create a user interface that can be interacted with based on the decoded control signals from the decode subgroup. Although the proposal itself hinted at the use of a graphical interface, it was still of importance to consider other forms of interfaces such as controlling an RC-car. This consideration will be discussed in Chapter 3.

Based on the aforementioned general requirements, the interface group decided on the requirements below with the primary goal of creating an enjoyable experience for the user.

- **Must** operate with a control input delay of less than 500 ms between the mental intention and the command execution.
- **Must** receive commands from the decode group with an accuracy (with respect to the user's actual brainwaves) of at least 70 percent.
- **Must** be completed without needing any additional budget.
- **Must** create an initial user interface that works through manual control(such as a keyboard) and that can easily be updated to work through decoded signal control.
- **Should/could** create a user interface for the user to control using control inputs from the decode group.
- **Should/could** add a live topographic map that displays the brain activity and/or a live time-frequency plot.
- **Should/could** add a calibration hub to obtain training data such that the decode subgroup can use it as a baseline.

The 500 ms delay between the mental intention and the command execution is based on the idea that having more than that amount of delay would be rather frustrating even when controlling simple things such as a cursor. Similarly, a brain wave decoding accuracy of less then 70 percent will very likely result in a less enjoyable experience for the user. Both the delay and accuracy requirements are a necessity for the interface but is something that only the decode subgroup can work towards; therefore, this requirement is also included by the decode group. It should however be said that the interface group can have a small influence on the delay between receiving the command and the execution of the command on the interface depending on the type of interface implemented; think of the time needed to execute code or sending a command over Bluetooth to an actuator. The should/could requirements include a topographic map of the scalp that would show the user which parts of their brain are the most active, providing feedback. This also provides useful data and performance metrics for further development and improvement of the EEG-BCI and for brainwave activity research purposes. The other should/could requirement is a calibration hub such that the decode model can be trained for every user separately since each individual's brain works differently.

3 Design of the interface

3.1. Interface choice

The choice for the type of interface affects the appearance and goal of the product as a whole and is thus an important choice in the project. The following concepts were considered:

- Translation of covert speech to the selection of a specific word
- Control of a wheelchair/drone/RC-car
- Identification of emotions
- Control of a cursor
- Control of a game

Each of these designs was also proposed with the impact that the product could have on potential users in mind. For most designs, the product is aimed at people with a certain impairment since they would have a necessary use for the product. The budget requirement and time constraints have the highest priority within the project and thus weigh greatly for the choice of the end product/interface. The ”translation of covert speech to the selection of a specific word” was taken out of the running mostly because it would not be feasible within the given time range. Additionally, the knowledge needed to accomplish this is not in line with that possessed by bachelor students. Controlling a wheelchair, drone or vehicle with brain waves would not only violate the budgetary requirement because of the hardware needed, there would also be a huge safety concern which would require too much time to offset. The identification of emotions was deemed too complicated since there is a lot of discrepancy between individuals in the way a specific emotion shows up in electroencephalographic (EEG) data [16].

This leaves only the option for the cursor and game. Ultimately, it was chosen to try to implement both of these with the game as a main component because it satisfies the budgetary requirement and was estimated to be feasible time-wise. Furthermore, this option was also feasible in terms of required background knowledge with choosing motor imagery as the control paradigm. The control of a cursor is also planned to be developed as soon as the decode subgroup would be able to classify five different control commands since the additional implementation of a cursor would be relatively straightforward. Another crucial factor for choosing a game for the interface is the fact that according to literature, visual feedback can help the user better control the game with their brain waves. Overall, using a graphical interface in the form of a game has a lot of potential for further development after the Bachelor Graduation Project ends.

3.2. Design features

Since the main task for this subgroup is to create a graphical user interface in the form of a game for the user to control, it is important to design the interface as user-friendly as possible. The interface consists out of several features that ought to optimize user experience. The user can choose to pick one of the features at the main menu. If the cursor hoovers over one of the menu options, that options takes on another color until the cursor is moved away from the menu option again. This phenomenon can be seen in Figure 3.1. If the user wants to select the option that the cursor is hoovering over, the user sends the correct command for that. If the cursor control does not end up being implemented as mentioned in the previous section, then it is the plan to have the user be able to scroll through the menu options and select one since this needs less control commands than a cursor that has to move freely over a large area.

Two of the features that can be selected in the menu are the games pong and breakout. For both games, the user controls the paddle in two directions, left and right for breakout and up and down for pong. It was decided to create both these games because of their simplicity and similarities but also with the idea in mind to analyze whether there are any difference in motor control ability between left/right and up/down after the BCI system if fully integrated. Because of the nature of the games, the user has incentive to move the paddle such that the ball is being reflected and does not travel past the paddle. The design of the pong and breakout games is further discussed in Chapter 4.

The third feature that can be accessed through the main menu is the calibration screen. The purpose of the calibration screen is for the user to check whether the electrodes are making properly contact with the user's scalp. The screen displays a layout of the user's head with the electrodes positioned on them and the electrodes are colored according to the sufficiency of the electrodes' contact with the scalp; red if the level of contact is insufficient and blue if the level of contact is sufficient. The calibration screen also displays the impedance of each electrode. The layout of the calibration screen can be seen in Figure 3.2. The calibration hub is to be completed with the display of a topographic map. The topographic map is supposed to display the live-recorded EEG data from the user, and this format is a clear and user-friendly way of doing this. It is intended for the topographic map to be shown on the interface during calibration such that they can increase their awareness of the required level of focus and intention to successfully control the games when playing them. The topographic map is also displayed during the games to provide the user with extra feedback. There is more information on topographic maps in Chapter 5.

Figure 3.1: Main menu with cursor hoovering over ”Calibration” option

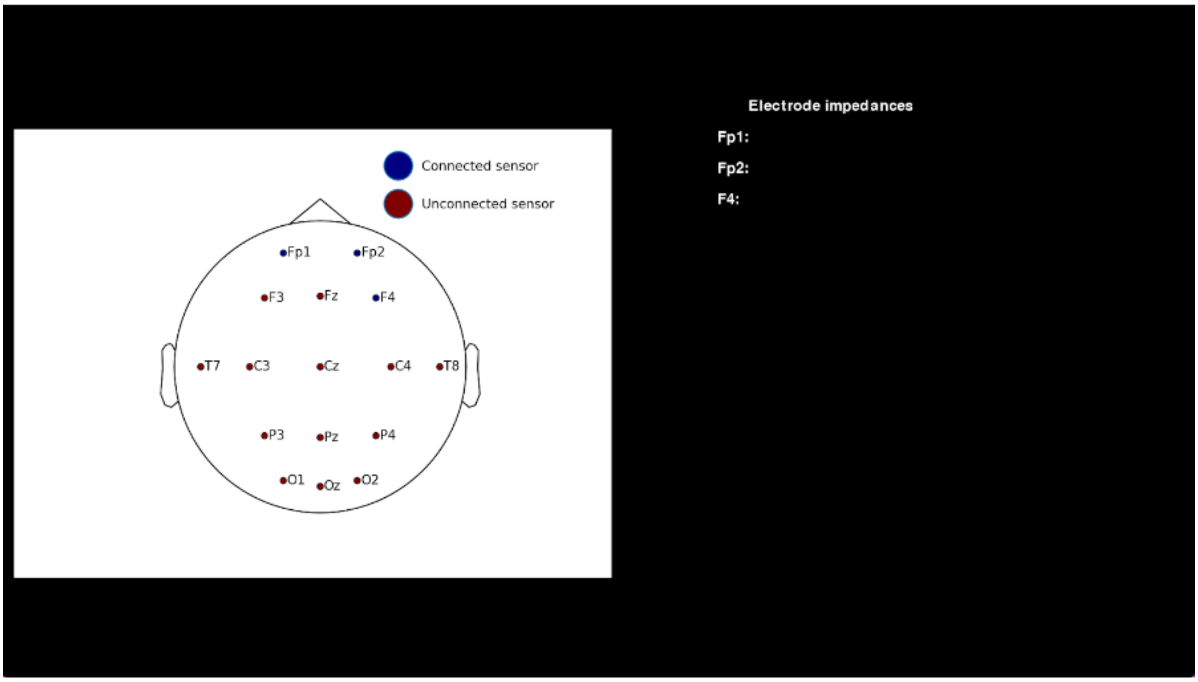


Figure 3.2: Calibration screen

3.3. Conclusion

As was described at the start of this chapter, a number of ideas for the interface were considered for the product with the end user, the BAP team's ability, time constraint and budget in mind. The final choice fell on a graphical user interface in the form of a game and optionally a movable cursor as well, since this fitted the best within all the requirements mentioned and due to the benefit of visual feedback according to literature. The features that are implemented in the graphical user interface are a start screen and two games: pong and breakout, which were chosen for their simplicity and the possibility of analyzing any motor imagery control performance difference between the games once the BCI is fully integrated. The details of the two games and their implementation will be discussed in the next chapter.

It should be said that at the time of writing this thesis, the subgroups' individual parts that form the full BCI system are not integrated yet. The decode group focused on being able to decode 2-3 different commands from the brain waves and since the cursor control requires 5 different commands, the cursor has not been implemented as of yet. The start screen can still be navigated with a regular mouse pad though. Also, the calibration hub was created but displaying the sensor electrodes does not work since a way to send over the live impedances of the electrodes through the OpenVibe software was not found. Openvibe will be explained in Chapter 6. If time permits it, the plan is to also figure out a way together with the measurement group to still receive the live electrode impedances. This can also be seen in the timetable in Appendix F.

4 Pong and Breakout

The two games that have been implemented are the games pong and breakout. Both games require the user to control a paddle in two directions such that a ball is reflected from the paddle instead of the ball passing the line that the paddle moves across. The games are programmed in Python using a specific library called Pygame. Pygame is designed for the development of video games in python and contains several modules and features that are useful for this. This chapter will describe the process of designing and implementing the games and will explain some of the principles behind certain techniques that are used to give insight in some specific design choices. The last section of this chapter will also describe the basic outline of the Pygame code for the games and the start screen.

4.1. Pong

Pong is a game that is played one on one. The user controls a paddle on one end of the field while the other paddle on the other side of the field is controlled by another player or is AI-controlled. The goal is to try to score a point by getting the ball past the paddle of the opponent and to prevent the opponent from scoring a point by hitting the ball back to the opponent's side of the field before the ball passes one's own paddle. The game ends when either the player or the opponent first reaches a score of five points.

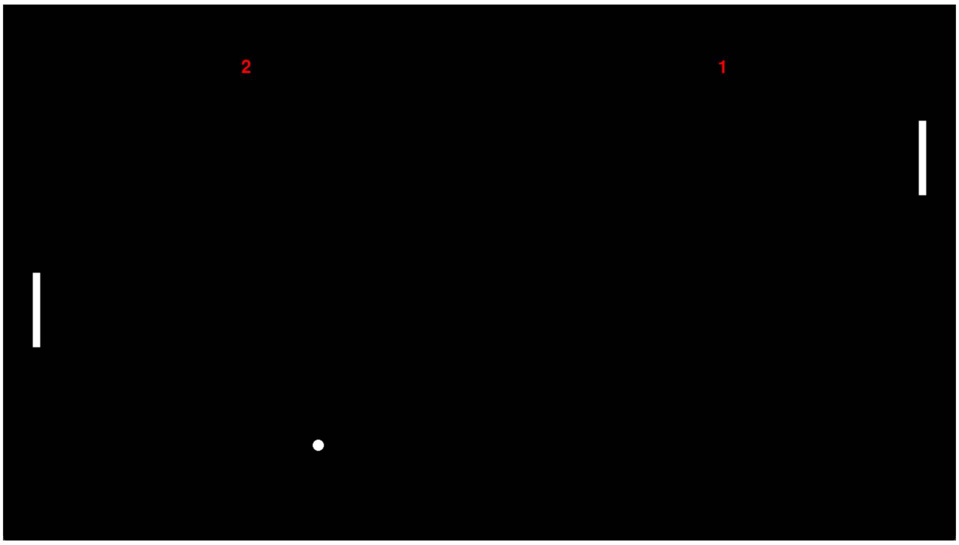


Figure 4.1: Pong

4.2. Breakout

Breakout is a single-player game where the user tries to eliminate all the bricks on the top of the screen by hitting them with a ball. At the same time, the player must use a paddle to keep the ball from passing through the bottom of the field and to reflect the ball back to the top of the field where the bricks are located. If the ball gets past the paddle at the bottom of the field, the ball is put back in the game and the player loses a life. The games ends when either all the bricks have been eliminated or the player has no lives left, whichever happens first.

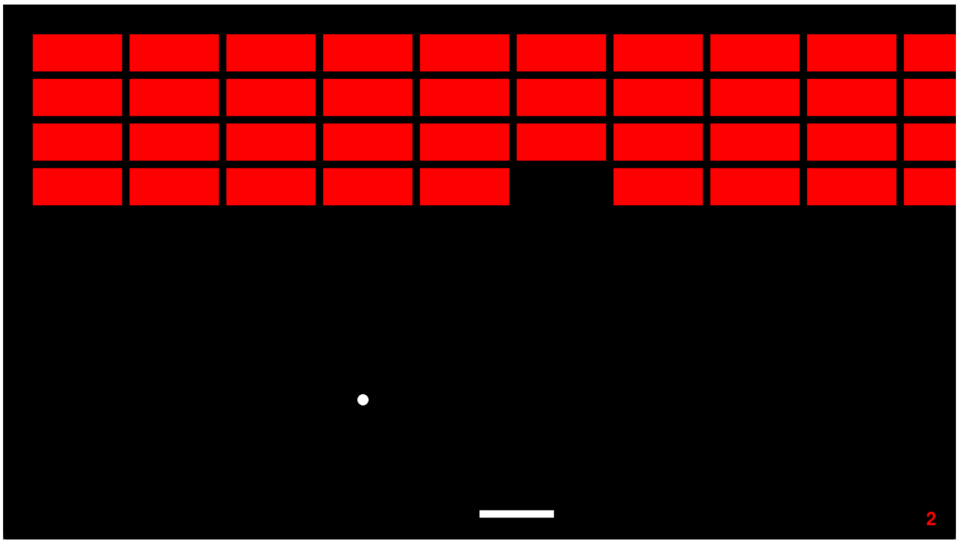


Figure 4.2: Breakout

4.3. Coding process and outline

Figure 4.3 below shows a rough overview of the full Pygame code which includes the start screen, calibration screen and the two games. In general, Pygame works by creating an infinite repeating game loop according to a certain number of frames per second and within each iteration certain game elements can be drawn on top of each other after which the screen is updated.

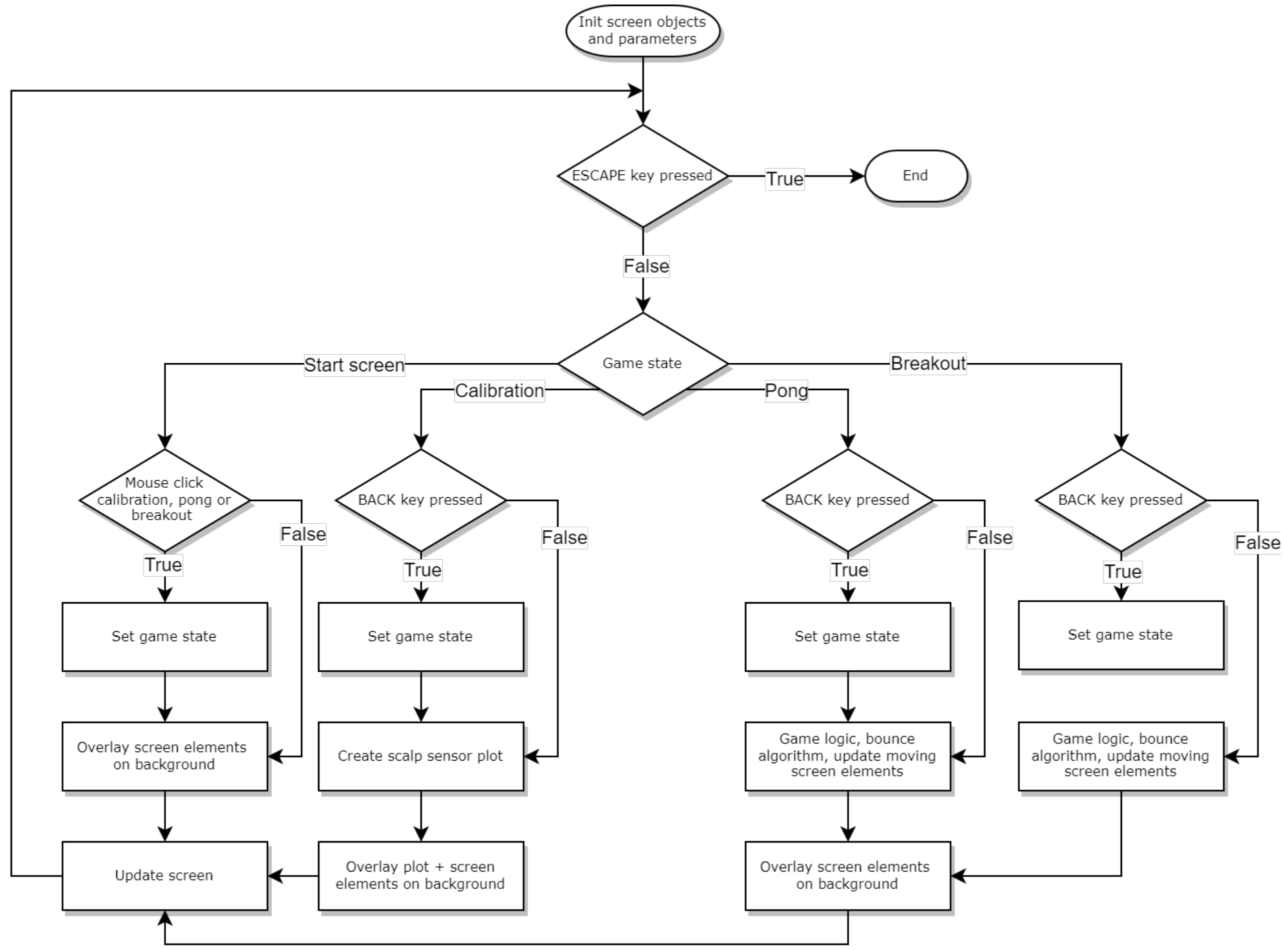


Figure 4.3: Code overview

The games are coded in a way such that they work on keyboard control, since the decode group must first be finished with their decode algorithm in order to know how to integrate the decoded outputs with the game. In the code, sprite classes are utilized for the paddles and the ball in which the drawing of the object and behavioural functions are defined. A sprite class is defined by Pygame itself and contains useful properties for moving objects. For instance, given that all sprite classes in one's game have a draw function defined, it is possible to draw them all at once on the screen with one line of code. The small piece of code below shows such a draw function.

```
    self.image = pygame.Surface([width, height])
    self.image.fill(black)
    self.image.set_colorkey(black)

    #For the ball:
    pygame.draw.ellipse(self.image, white, [0, 0, width, height])

    #For the paddle:
    pygame.draw.rect(self.image, white, [0, 0, width, height])
```

The way to create games in Pygame is by creating surfaces with shapes on them and then overlaying them on top of each other in the desired manner each iteration of the game loop. The small piece of code above creates a surface of the same color as the background of the field (black), on which the desired shape such as a circle is then drawn in white. This manner of creating visual objects is typical for Pygame. The paddle class further contains functions that are called upon when their respective key is pressed. As can be seen in lines 448-449 of Appendix A.1, if the key "up" gets pressed, the function move_player_up() is called, in which the vertical position of the paddle is adjusted upwards if the topside of the paddle is still below the upper boundary of the screen. A similar mechanism is used for the paddle to move down, left or right. The ball class contains update functions and a reset function. In the update functions, the code is written such that the ball continues its course but gets reflected if it hits a paddle, a brick in breakout, the upper or lower boundary of the field in pong, or the upper, left side or right side boundary of the field in breakout. It also updates the scores of the players if the ball hits the left side or right side boundary of the field in pong and updates the number of lives of the player if the ball hits the lower boundary of the field in breakout. The reset function is then called, which positions the ball back into the field and decides randomly in which direction the ball is launched. The paddle move functions and the position update functions are part of the 'game logic' in Figure 4.3. The ball reflections are part of the 'bounce algorithm' shown in Figure 4.3. Figure 4.4 below shows a diagram of the ball bounce code in the case of collision between ball and brick in the breakout game. The principle is the same for all other reflection such as the ones in the pong game. The initial script for the interface can be found in Appendix A.1.

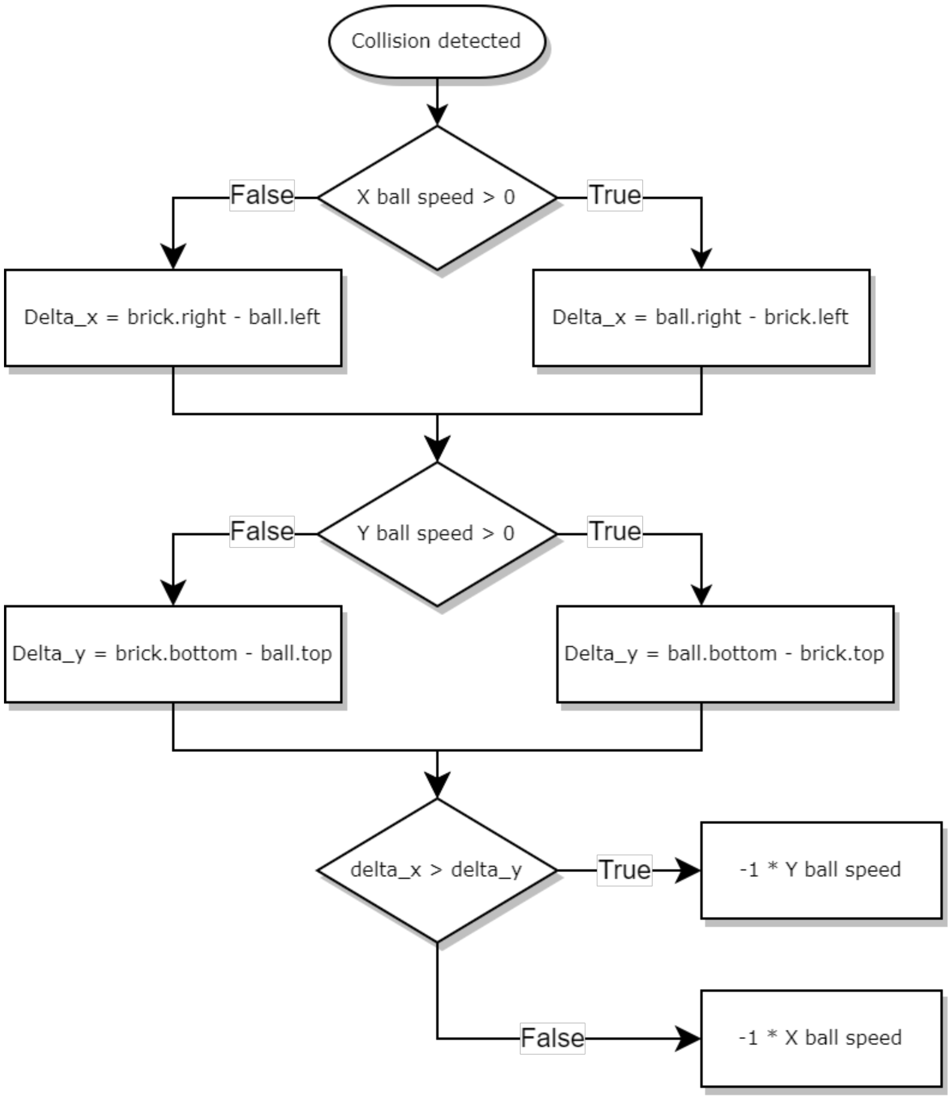


Figure 4.4: Ball bounce algorithm

4.4. Conclusion

The games pong and breakout are fully coded, implemented and tested with manual keyboard control. The scripts are written with the user in mind; for example, the AI in pong is made to be imperfect and the measurements of objects are defined such that the game can be played on screens with different dimensions without looking distorted.

As briefly mentioned in the conclusion of the previous chapter (Section 3.3), at this time, the BCI system is not fully integrated yet. In order to integrate the interface with the rest of the BCI, the code must be slightly adjusted for the new input. This should not prove difficult since this could for example be done with a single integer variable that is received from the decode group. This variable will replace the keyboard inputs up/down or left/right. However, whether the games will still function accordingly with the control commands that are decoded from the measured EEG data, can only be truly known after testing the fully assembled BCI. Subsequently, some fine tuning of the ball speed, paddle speed or AI behaviour might be needed to keep the games enjoyable to play. If time permits, the plan is to integrate the BCI system and thus change the games control from keyboard to commands from the decode group in the weeks before the thesis defence. This can also be seen in the timetable in Appendix F.

5
Visual feedback

5.1. Neurofeedback choice and improvement

As mentioned in the introduction, neurofeedback has proven to have a positive impact on the performance of one's motor imagery control [17]. This is especially significant since it allows for the BCI system to have a higher overall performance without having to improve anything internally. Whilst neurofeedback has shown to be beneficial for the most common modalities of sensory feedback across the board, there are (sometimes subtle) differences between their efficacy on specific motor-imagery-controlled mechanisms. For example, a 2023 study compared how tactile, visual and tactile-visual feedback influenced the performance of participants executing several different motor controlled tasks with a myoelectric prosthetic hand, and it concluded that each of the three types of feedback had the most impact on a different type of motor controlled task [18]. Although [18] mentions multiple studies with different conclusions on the comparison between the effectiveness of tactile and visual feedback, it seems that some sort of kinesthetic feedback is generally more beneficial than visual feedback for the improvement of motor imagery control [19]. Based on this knowledge, choosing a form of kinesthetic feedback as the sole form of feedback or as part of the feedback system would optimize the overall BCI system the most. However, Chapter 3 discussed the reasoning (such as time constraints and budget etc) for choosing a graphical user interface and therefore visual feedback as neurofeedback, which might not be the best option but still a fitting one in the context of this project. It is planned to test the effect of the visual feedback after the entire BCI is integrated by having one test group doing motor imagery through playing the games and having another test group simply doing motor imagery with their eyes closed. The results are expected to show that the first test group has a higher accuracy in the translation of the mental intention to the control command than the second test group. This would indicate that the visual feedback in fact improves the performance of the entire BCI system.

5.2. Topographic maps

Another visual feedback related topic that was also included as a should/could element in the program of requirements, is topographic maps ([20] discusses an algorithm for topographic brain activity plotting). The idea behind using topographic maps, potentially during the calibration of the decode algorithm for a user, is to help the user learn what works best and how hard they should focus to best control the game by showing their brain activity. Furthermore, the topographic map can be displayed during the game to provide the user with additional feedback.

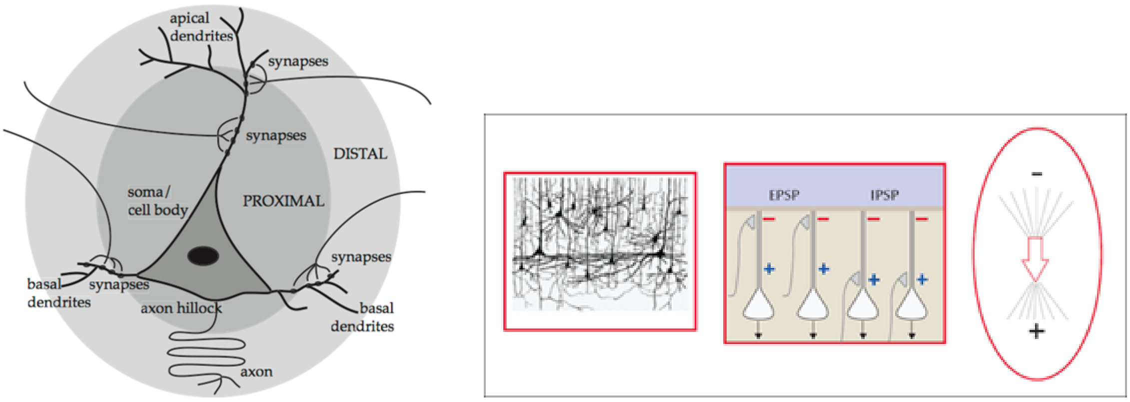
5.2.1. Background

Understanding the information shown by topographic mapping of brain activity requires some knowledge on what actually happens in the brain and how the EEG measurements are then used to create the map.

In order for a certain brain activity to occur, neurotransmitters are used to signal neurons in the brain.

Pyramidal neurons are neurons that are located near the cortical surface and are positioned perpendicular to it. The apical dendrites of the pyramidal neurons lay parallel to each other and run from the cell body to the cortical surface [21]. The arrangement of the pyramidal neurons is depicted in Figure 5.1b. When the pyramidal neurons receive neurotransmitters, channels in the neuron membrane are opened and allow for a flow of ions into or out of the neuron. As a result, an electrical potential is created in the local extracellular space. For excitatory synapses, the local extracellular space becomes negative while for inhibitory synapses, the local extracellular space becomes positive. A dipole exists as the relative charge in the non-local extracellular space is opposite to the charge in the local extracellular space. This generates a current going in a specific direction [22]. Since the pyramidal neurons are arranged parallel to each other, the currents can sum up to a large enough current that it can be detected through the scalp [21]. In general, an area of about 10 cm^2 should have simultaneously activated neurons for the brain activity to be recorded over the scalp [21].
Synapses can exist close to the cell body of the pyramidal neuron or further away on one of the dendrites. If a synapse that is located on an apical dendrite close to the cortical surface causes an Excitatory Postsynaptic Potential (EPSP), or a synapse far from the cortical surface causes an Inhibitory Postsynaptic Potential (IPSP), the negative pole of the dipole is near the cortical surface. If a synapse that is located on an apical dendrite close to the cortical surface causes an Inhibitory Postsynaptic Potential (IPSP), or a synapse far from the cortical surface causes an Excitatory Postsynaptic Potential (EPSP), the positive pole of the dipole is near the cortical surface. Figure 5.1b shows a visual representation of this phenomenon.
Because the cortical surface does not run parallel to the skull everywhere, one does not simply observe a positive or negative potential right above an area of neural activity. The orientation of the dipole along the neurons relative to the scalp determines how the positive and negative potentials are distributed on a topographic map. Figure 5.2 depicts how this occurs.

The electrodes measure the potentials on the scalp only on the point where they make contact. For the areas between the electrodes, interpolation is used to calculate the potentials there [22]. The potentials can then be mapped as has been done in Figure 5.3, where the blue color represents negative potentials and the red color represents positive potentials. There is however also a different way of plotting topographic maps compared to the 'classic' topographic maps described before. Appendix E.3 shows an example of such a map which is called a PSD topographic map. Rather than potential, these maps show the spectral power density averaged over a specified range of frequencies or at a specific frequency. So to clarify, the units of the classic and PSD topographic maps are μV and $\mu V^2/Hz$ respectively. Note that the PSD topographic maps display only red colours since power spectral density is always positive. It should also be noted that there are different ways of creating classic topographic maps in temporal terms. As described before, PSD topographic maps are created by averaging the PSD of a data segment over a certain range of frequency or at a certain frequency. For classic topograhpic maps however, it is possible to plot the activity at one time instance as well as the average activity over a certain time window.



(a) Pyramidal neuron **(b)** EPSP and IPSP [21]

Figure 5.1: Schematics of pyramidal neurons

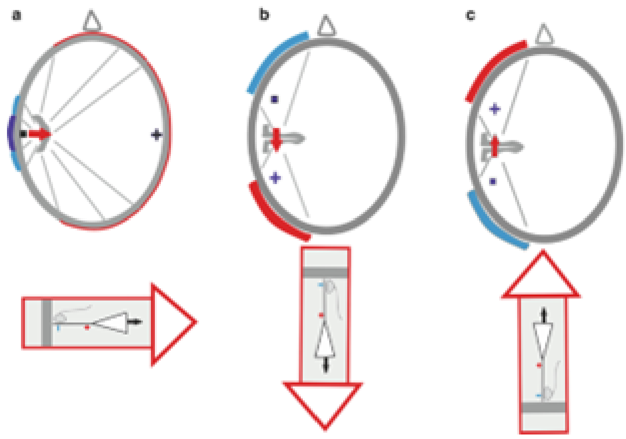


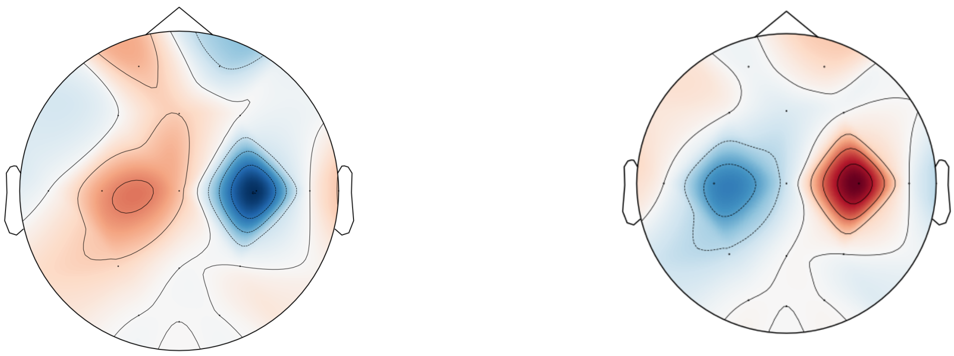
Figure 5.2: Orientation of dipole relative to the scalp [22]

5.2.2. Coding

Initially, a script for topographic mapping was written such that a recording of EEG data could be plotted. The data that was used for this initial script was a motor control recording made by the measurement subgroup. During the recording of the data, the subject had their left hand open hand, clenched a fist and then released it again. The code for generating the topographic mapping is written in Python and makes use of the library MNE. MNE is specifically made for working with neurophysiological data, which includes EEG. In the code, a montage of the helmet with the sensors' positions is created. The recorded data is extracted from its file and set with the sensor locations; with this, topographic maps are created for each timestamp. Using the function FuncAnimate(), an animation is created from the individual topographic maps. The animation is similar to how the mapping should look like on live-streamed data. The script of this code can be found in Appendix C.1. The code here creates a topographic map for each timestamp, but the code can be adjusted such that the topographic map is made for the average of multiple timestamps, as has been done for the code in Appendix C.2.

5.2.3. Interpretation

The topographic maps are relatively easy to interpret. In Figure 5.3, the topographic maps shown are snapshots from the animation that was mentioned in the previous subsection. As mentioned in Section 5.2.1, the blue color represents negative measured potentials and the red color represents positive measured potentials. The darker the color is, the higher the absolute amplitude is. The measured potentials are scaled in μV. The topographic map is quite intuitive to read in the sense that the area on the head plot with darker red or blue corresponds to the part of the brain where relatively more activity is taking place. This phenomenon can also be seen in Figure 5.3.

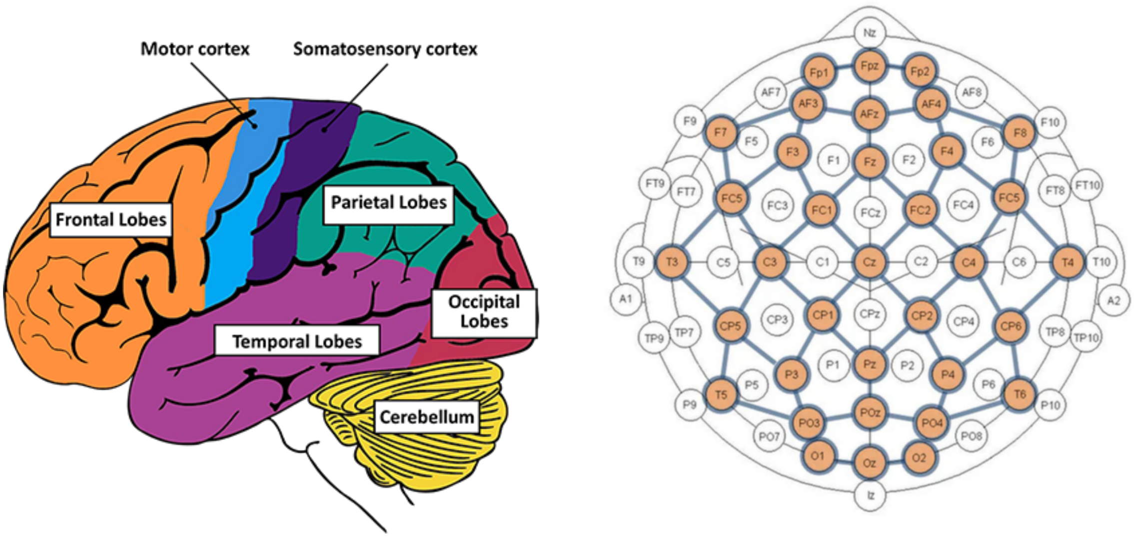


(a) t = 0 - 0.128 s

(b) t = 0.256 - 0.384 s

Figure 5.3: Left hand clench topographic maps

As briefly mentioned before, the test setup from which the above topographic maps resulted, included left hand clenches. The electrode placements on the scalp followed the 10-20 system. The 10-20 system, as shown in Figure D.2, is especially designed to capture all the different interesting brain regions with specific electrode placements on the scalp. The electrodes names are a reference to certain brain regions, where Figure D.2 also shows these main human brain regions and their names. For example, in the O1 and O2 electrodes, the letter ’O’ stands for the occipital region and the number 1 indicates that it is the most left position; an increase in number means that the electrode is more to the right.



(a) Brain regions) **(b)** 10-20 EEG electrode system

Figure 5.4: Brain regions and electrode placement

For this test, where the subject is clenching their right hand the regions of interest lie in the motor cortex as indicated in the figure below. The motor cortex is mainly involved in planning, control and execution of voluntary movements. The closest electrodes on the OpenBCI headset to this region are C3 and C4. These are represented by two black dots in the previously shown topographic map, where C3 is the middle left dot (near the slightly darker orange/red color) and C4 is the middle right dot (right on top of the dark blue are). It is expected that from a left hand clench, there should first be a relatively high negative potential response from C4 and a smaller positive potential at the opposite C3. Then afterwards the opposite should happen where there will be a relatively high positive potential response from C4 and a smaller negative potential response from C3. This phenomenon should show virtually only in the alpha waves (7-12 Hz) [23]. Though, since during the test, the subject had their eyes closed and sat as still as possible, these expected alpha wave potentials should be dominant such that they will be clearly visible in the potential topographic map that contains all frequencies.

Looking back once again at the two topographic maps from the left hand clench in Figure 5.3, we indeed see the high negative peak as the dark blue color on top of C4 in the left hand figure, alongside the high positive potential in C4 in the right hand figure. A longer sequence of topographic maps from this left hand clench are shown in Appendix E.1. The topographic maps are each created from the average potentials from 32 samples of data (approximately equal to 130 ms). The maps start from t = 0, which simply indicates the starting point that was chosen where the left hand clench approximately started and not the very start of the recording itself.

5.3. Conclusion

So now to go back to the initial goals of this chapter in order to discuss the conclusion for them: 1) discuss the choice of neurofeedback chosen and 2) create and implement topographic maps to potentially help with user calibration.

As described at the start of this chapter, it will only be possible to fully test the influence of visual feedback and justify our choice once the entire BCI has been integrated using the work of all three subgroups. So the conclusion is that tests should be done after integration in case of continuation of the BAP project after the deadline. The proposal for the future is to create a test setup with two separate groups where one group plays the games using the BCI system while the other group has to simply close their eyes and attempt to think of left and right while comparing decode algorithm accuracy results.

As far as the topographic maps go, it is possible to plot them for pre-recorded data. The previously shown topographic maps from a left hand clenching data segment, showed results that were very similar to what was expected. Though looking at the longer sequence of topographic maps in Appendix E.1, it becomes evident that the topographic maps do not show perfect results. The second figure (t = 0.128-0.256) for example, shows an odd topographic map in-between the very nice looking maps. It should also be said that not all left hand clenches appeared in the way shown in Figure 5.3. Sometimes they showed very messy topographic maps indicating a lot of activity in other channels. Unfortunately, the brain is very complex and recorded brain waves are very sensitive to all sorts of factors such as test setup, focus, noise etc.

But nonetheless, the topographic maps could be a helpful tool for extra feedback by making the user aware of the focus required to create a correct command from the decode algorithm. However, as will be discussed in Chapter 6, the integration with the software that will connect all the subgroups together and will provide the topographic maps with live data to plot, came with some issues. At this moment in time, this issue was not solved. It was collectively decided that all the subgroups should focus on their individual parts and to leave the live data integration for the weeks before the thesis defence, which again can be seen in the timetable in Appendix F. Once the BCI is assembled, the effect of the topographic maps can also be tested in a similar test setup as proposed for the testing of the visual feedback that the game provides; have one test group train the decode algorithm with motor imagery tasks while looking at the topographic map of their own brain activity while another does the same but without the feedback from the topographic maps. A third and final test proposal is for testing the influence of showing the live topographic map of the user's brain activity while playing the games. Similar to the other tests, one group should play the game without the topographic map and the other group with the topographic map.

6 Integration elements

6.1. Concurrency

The possibility of showing topographic maps of brain activity during the game was explored after the creation of the intial script described in Section 5.2.2. Recall that this was a should/could point in the determined requirements in Chapter 2 as well. Pygame already handles 'events' such as keyboard inputs in a concurrent way, however the future addition of live topographic map plotting might not work so well in synchronous code. The infinite game loop iterates a certain number of times per second and it could for example happen that the topographic map plot takes too much time in order for the game loop to upkeep its framerate. The framerate is tied to the speed of certain game elements and thus it could cause unwanted behaviour. In order to prepare for the need for parallel execution of both the interface with the games and the topographic map plotting for the complete integration of the BCI system, the possibilities of asynchronous/concurrent programming was researched and implemented using an animation plotting function from the package matplotlib in Python to simulate the topographic map plotting. There are three packages available for Python that allow for asynchronous, concurrent and/or parallel execution of code: multiprocessing, threading and asynchio. Each of these has their own advantages and disadvantages which will be discussed below in order to pick the most suited option.

Multiprocessing runs multiple processes simultaneously and has a separate memory space for each of the processes. It uses multiple CPU cores and is therefore able to truly run the processes parallel. Being able to truly execute tasks parallel increases the performance of the program since the workload is distributed across the CPU cores and their resources are used efficiently. True parallelism also comes with the benefit that the failure of one process does not affect the other processes which makes for a more robust program. Because of the use of multiple CPU core, multiprocessing is great for performing CPU-bound tasks such as simulations and extensive calculations. However, there are also negative consequences attached to the use of multiprocessing. Even though the individual memory allocation of each process has benefits, it also results in a higher memory usage, which can be a very large downside depending on the nature of the program multiprocessing is used on. Another disadvantage of multiprocessing is the time consumption for the startup and teardown that comes with the creation and management of processes, and for the additional communication overhead needed to share data between processes. If these downsides have too large of a negative impact, the use of threading or asyncio should be considered.

Threading allows for running multiple threads with a single process. Threading has the major advantage that it is easily applicable; if code has not explicitly been written to run asynchronous, threads can still be used with this. However, this also comes with a foundational limitation to threading. Threading is not true parallelism. In Python, only one thread is allowed to execute code at a time. Threading is still more efficient than subsequent programming but it is not improving the performance of a program on the same scale as for example multiprocessing does. That is not to say that there are not other characteristics that make threading a viable option for certain scenarios. Threading comes with the benefit that threads are lighter-weight than processes since the threads within a process share one memory space. The shared space of memory also allows for simplified communication between the threads within a process. Moreover, the creation and management of threads is generally faster and requires less overhead than processes. Even though threads have great properties, there are also a few issues to take into account with them. Multiple threads simultaneously accessing shared resources can lead to race conditions and damage to data. Thread safety must therefore be managed but this can be challenging. Furthermore, an increasing number of threads can exhaust resources for their creation and the increasing overhead to manage the threads can have a negative impact on the performance.

Asyncio uses a single-threaded event loop to switch between coroutines, which results in efficient handling of I/O operations without the execution being blocked. More specifically, the I/O-bound tasks are efficiently handled because other coroutines are allowed to continue their execution while waiting for I/O operations to complete. The keywords ”async” and ”await” are used to create the coroutines. The coroutines can be scheduled and executed concurrently. The single-threaded operation of the event loop provides a level of simplicity in reasoning the concurrency since there are not multiple processes or threads to manage. It also makes asyncio more efficiently handle many concurrent connections, which is a desirable trait for certain applications. Unfortunately, the single-threaded nature of asyncio also has negative consequences; for once, it does not allow for true parallelism which makes CPU-related tasks perform low with. Using asyncio can also be quite complex to incorporate if one does not have a good enough understanding of its operating procedure.

When considering the concurrent execution of the plotting animation and interface games, the code is not I/O bound. I/O bound refers to long waiting times on inputs, such as with servers processing requests. The reason for implementing concurrency was the possible CPU bound within each game loop iteration. According to the advantages and disadvantages of the three concurrency packages explained above, multiprocessing is the best option for a CPU bound problem. On the other hand, this is not a classic CPU bound problem but a relative CPU bound caused by the framerate of Pygame (number of game loop iterations per second). As a result, threading could also be suited for this particular case, since in case of running things concurrently the potential issue of the topomap plotting taking too long for the game framerate is no longer. Both multiprocessing and threading have been tested without any noticeable difference between the two. Due to the nature of the code it is hard to truly benchmark the efficiency of both concurrency methods but since multiprocessing requires more overhead, it was ultimately decided to go with threading. A number of things in the initial interface code had to be adapted to work with threading. The revised code can be viewed in Appendix A.2.

6.2. Implementation

At the end of the project, it is the goal to integrate the parts from each subgroup into a fully functioning BCI. Collectively, it was decided to first attempt to do the integration in OpenVibe [24]. OpenVibe is the software that the measurement subgroup used to acquiring the data from the OpenBCI headset, which streams the recorded raw data to a USB stick that can be plugged into a computer. OpenVibe can be used to read out this USB stick and display the live incoming data. It is also possible in OpenVibe to create boxes in which subsystems are defined and then to assemble the full system by connecting the boxes with lines. The lines are drawn so that signals can be transported between different parts of the system. For a Python script to be executed in OpenVibe, it must be put into a Python Scripting box and the code itself should be structured in a specific manner. The code should contain a class called MyOVBox which should be made up out of a constructor, a process function and a deconstructor. In the process function, new chunks of data that have been stored in an input buffer by OpenVibe can be accessed. The data is then to be used for whatever the script has been written for and possible output data can be stored in an output buffer. Therefore, the Python code that has been written for the interface has to be adjusted such that it conforms to this format. This manner of integration is using built-in functions of OpenVibe, but it would also be possible to have the data from the OpenBCI headset be streamed to an IP address and to access that data whilst executing the code in Spyder, PyCharm or another environment that supports Python. Both are possible but using the built-in functions of OpenVibe would streamline the total integration of the BCI, so executing the code in OpenVibe is the initial approach.

A first attempt towards integration was done by adjusting the code written for the topographic map for it to fit the format that OpenVibe requires for Python scripts. If it ended up being possible to display the topographic map in OpenVibe and to make the data it plots be live streamed, then this would indicate that integrating the entire BCI system this way is very likely to be successful. OpenVibe essentially calls on the process function every clock tick (with a maximum frequency of 128 Hz) to update the input buffer with new chunks of data. As a result the way in which the animation is created in the original script would not work with this architecture of data. Since the function that takes care of the animation (FuncAnimation from the matplot library) in fact behaves as a while loop that runs until all frames of the animation have been displayed, the execution of the script in OpenVibe would cause the entire software to crash. The script was then rewritten such that the individual topographic map for the average of a selection of timestamps would be plotted in a window and then the window would be cleared; for the next iteration of the looping process function, the topographic map for the average of the next selection of timestamps would be plotted in the same window and then the window would be cleared again. The window should thus never close. However, running this script in OpenVibe did not give the desired results. The topographic map of the average of a selection of timestamps would be plotted but the window had to be manually closed before a new window would pop up with the topographic map for the next selection of timestamps. The script can be found in Appendix C. It is most likely that there is something internally with OpenVibe that causes this since the code does work in Spyder when the OpenVibe-specific format is removed and the code is put in a while loop. The alternative way of streaming the measurement data to an IP address and executing the code externally would bypass the issues with OpenVibe; however, it does affect how the entire BCI is integrated. In the coming weeks, this option that can be discussed and explored with the other subgroups.

6.3. Conclusion

The interface has also been coded in a revised way that allows for an animation to run concurrently with the interface, in preparation for live topographic map plotting while running the interface as soon as the BCI system is integrated. The method of concurrency was chosen to be the Threading method since multiprocessing and threading showed no difference and threading is known to require less overhead. Attempts at using OpenVibe to make the first steps towards the integration of the complete BCI have been unsuccessful. More time must be done to either get this to work or an alternative method must be explored.

7 Conclusion

7.1. Conclusion

The goal of this part of the project was to create an interface as part of a BCI system. It was decided that the interface were to consist out of two games that can be controlled using the command signals that were decoded from the EEG data. Besides the two games that were developed, a complete graphical user interface was created with additional features to optimize the user experience.

The interface had to comply to the requirements stated in Chapter 2. The requirements cannot be said to have all been met so far; this is mainly due to the fact that we have not been able to receive input from other subsystems to test requirements or be able to consider requirements fulfilled without that ability:

- The requirement to create a user interface that works with manual control has been met.
- For the requirement to create a user interface for the user to control using control input from the decode group, the user interface has in fact been created as mentioned in the previous point but it is still not being controlled with decoded control commands.
- For the requirement to operate with a control input delay of less than 500 ms between the mental intention and the command execution and the requirement to receive commands from the decode group with an accuracy of at least 70 percent, these cannot be achieved (solely) by us. These requirements are very much depended on the performance of the decode subsystem. We have also not been able to test whether these requirements are met since that would require the full integration of the BCI.
- The requirement for the subsystem to be completed without the use any additional budget has been met.
- For the requirement to possibly add a live topographic map or live time-frequency plot, the topographic map has been implemented but has not been shown to work on live streamed data as of yet.
- For the requirement to possibly add a calibration hub, many elements of this feature have been written but as will be discussed in the next section, this is not completely finished.

Furthermore, research has been done into the possibility of including SSVEP elements in the games in the future. This could be done by making the ball constantly flicker or even creating new games that combine motor imagery tasks and SSVEP elements. The research is described in Appendix E.2. This is placed in an appendix because the conclusions following from the research do not relate to the requirements described in Chapter 2.

7.2. Future work

There are several aspects of the interface that can be improved in the future such that the interface can be fully functional and work within the BCI as intended.

As of now, the interface still works through keyboard and mouse control. A substantial amount of time has already been spent adjusting pieces of code to attempt using it in OpenVibe. OpenVibe is the environment which was initially intended for the project to integrate the entire system on but due to the problems that were encountered trying to do this, as described in Section 6.2, this might not end up being used. Either a lot of time must be spent resolving the problems or an alternative to using OpenVibe must be searched for. Actually changing the keyboard input to a command input coming from the decode subsystem should be quite straightforward.

As described in Chapter 5, the impact of the visual feedback still needs to be tested. Only after the assembly of the complete BCI system, the tests can be conducted. Three tests have been proposed to conduct after the completion of the BCI, which should indicate whether the visual feedback has an effect on the user's performance. The first test consists of two test groups where one group plays the games using the BCI system while the other group is the control group. The second test requires the first group to train the decode algorithm with motor imagery tasks with feedback from the topographic map of their own brain activity, whilst the control group does the same but without the feedback from the topographic maps. For the third test, the first test group plays the games with a live topographic map as additional feedback and the control group plays the games without the topographic map being shown. It is expected that in each test, the group with the (additional) visual feedback has better results than the control group and that it can be concluded that the feedback improves the performance of the entire BCI. However, this can only be done after the tests have actually taken place.

Another point of improvement would be the completion of the calibration hub. So far, several features have been created for the calibration, such as the topographic map and the calibration screen with the display of the electrode impedances and with the map that shows whether an electrode is making contact with the user's scalp. Above all, the calibration hub must be tested with live input from the headset and decoded control commands, and again, this can only be done after the BCI is fully integrated.
For the calibration hub, there is also the issue with the ability to display the topographic map in the same window as the calibration. This is mostly a stylistic issue as the topographic map could also be shown in a separate window and still fulfill its purpose. Therefore, this issue has been put on the back burner but this is still something we would like to resolve. The same is true for the topographic map being displayed during the game.

As mentioned, research into the feasibility of including SSVEPs or combining them with motor imagery to increase the user experience was done since it would allow for more complicated games to be played. The research showed that we can see SSVEP responses from the measurement with the OpenBCI headset and data acquisition methods employed by the measurement subgroup. Though, the SSVEP responses were not present in the expected occipital channels. Since most SSVEP detection algorithms would then probably fail to actually detect them, it should be concluded that at this time, working with SSVEPs will prove to be difficult. Nonetheless, SSVEPs are a very interesting topic and thus we propose that in case of further development of the BAP project, more test should be conducted to try and get better results such that combining SSVEPs with motor imagery and implementing an SSVEP detection algorithm would be feasible in the future.

Bibliography

[1] M. Alimardani, S. Nishio, and H. Ishiguro, "The importance of visual feedback design in BCIs from embodiment to motor imagery learning," *PLOS ONE*, vol. 11, no. 9, D. Hu, Ed., e0161945, Sep. 2016. DOI: `10.1371/journal.pone.0161945`. [Online]. Available: `https://doi.org/10.1371/journal.pone.0161945`.

[2] Z. Wang, Y. Yu, M. Xu, Y. Liu, E. Yin, and Z. Zhou, "Towards a hybrid BCI gaming paradigm based on motor imagery and SSVEP," *International Journal of Human–Computer Interaction*, vol. 35, no. 3, pp. 197–205, Mar. 2018. DOI: `10.1080/10447318.2018.1445068`. [Online]. Available: `https://doi.org/10.1080/10447318.2018.1445068`.

[3] H. Cho, M. Ahn, M. Kwon, and S. Jun, "A step-by-step tutorial for a motor imagery–based bci," in Jan. 2018, pp. 445–460.

[4] N. Padfield, J. Zabalza, H. Zhao, V. Masero, and J. Ren, "EEG-based brain-computer interfaces using motor-imagery: Techniques and challenges," *Sensors*, vol. 19, no. 6, p. 1423, Mar. 2019. DOI: `10.3390/s19061423`. [Online]. Available: `https://doi.org/10.3390/s19061423`.

[5] C. Klaes, "Invasive brain-computer interfaces and neural recordings from humans," in *Handbook of Behavioral Neuroscience*, Elsevier, 2018, pp. 527–539. DOI: `10.1016/b978-0-12-812028-6.00028-8`.

[6] M. F. Mridha, S. C. Das, M. M. Kabir, A. A. Lima, M. R. Islam, and Y. Watanobe, "Brain-computer interface: Advancement and challenges," *Sensors*, vol. 21, no. 17, 2021. DOI: `10.3390/s21175746`.

[7] F. Su and W. Xu, "Enhancing brain plasticity to promote stroke recovery," *Frontiers in Neurology*, vol. 11, Oct. 2020. DOI: `10.3389/fneur.2020.554089`. [Online]. Available: `https://doi.org/10.3389/fneur.2020.554089`.

[8] Z. Song, T. Fang, J. Ma, *et al.*, "Evaluation and diagnosis of brain diseases based on non-invasive BCI," in *2021 9th International Winter Conference on Brain-Computer Interface (BCI)*, IEEE, Feb. 2021. DOI: `10.1109/bci51272.2021.9385291`.

[9] S. Siuly, S. K. Khare, V. Bajaj, H. Wang, and Y. Zhang, "A computerized method for automatic detection of schizophrenia using EEG signals," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 28, no. 11, pp. 2390–2400, Nov. 2020. DOI: `10.1109/tnsre.2020.3022715`.

[10] J. E. Muñoz, R. Chavarriaga, J. F. Villada, and D. SebastianLopez, "Bci and motion capture technologies for rehabilitation based on videogames," in *IEEE Global Humanitarian Technology Conference (GHTC 2014)*, 2014, pp. 396–401. DOI: `10.1109/GHTC.2014.6970312`.

[11] J. A. Hennig, E. R. Oby, M. D. Golub, *et al.*, "Learning is shaped by abrupt changes in neural engagement," *Nature Neuroscience*, vol. 24, no. 5, pp. 727–736, Mar. 2021. DOI: `10.1038/s41593-021-00822-8`. [Online]. Available: `https://doi.org/10.1038/s41593-021-00822-8`.

[12] C. Vidaurre, C. Klauer, T. Schauer, A. Ramos-Murguialday, and K.-R. Müller, "EEG-based BCI for the linear control of an upper-limb neuroprosthesis," *Medical Engineering &amp Physics*, vol. 38, no. 11, pp. 1195–1204, Nov. 2016. DOI: `10.1016/j.medengphy.2016.06.010`. [Online]. Available: `https://doi.org/10.1016/j.medengphy.2016.06.010`.

[13] P. Peining, G. Tan, A. Aung, and A. a. Phyo wai, "Evaluation of consumer-grade eeg headsets for bci drone control," Aug. 2017.

[14] K. H. Khng and R. Mane, "Beyond bci—validating a wireless, consumer-grade eeg headset against a medical-grade system for evaluating eeg effects of a test anxiety intervention in school," *Advanced Engineering Informatics*, vol. 45, no. 101106, 2020. DOI: `10.1016/j.aei.2020.101106`.

[15] E. Ratti, S. Waninger, C. Berka, G. Ruffini, and A. Verma1, "Comparison of medical and consumer wireless eeg systems for use in clinical trials," *Frontiers in human neuroscience*, vol. 11, no. 398, 2017. DOI: 10.3389/fnhum.2017.00398.

[16] S. Gannouni, A. Aledaily, K. Belwafi, and H. Aboalsamh, "Emotion detection using electroencephalography signals and a zero-time windowing-based epoch estimation and relevant electrode identification," *Scientific Reports*, vol. 11, no. 7071, 2021. DOI: 10.1038/s41598-021-86345-5.

[17] D. Zapała, P. Francuz, E. Zapała, *et al.*, "The impact of different visual feedbacks in user training on motor imagery control in bci," *Applied Psychophysiology and Biofeedback*, vol. 43, pp. 23–35, 2017. DOI: 10.1007/s10484-017-9383-z.

[18] S. Su, G. Chai, W. Xu, *et al.*, "Neural evidence for functional roles of tactile and visual feedback in the application of myoelectric prosthesis," *Journal of Neural Engineering*, vol. 20, no. 1, 2023. DOI: 10.1088/1741-2552/acab32.

[19] M. Vukelić and A. Gharabaghi, "Oscillatory entrainment of the motor cortical network during motor imagery is modulated by the feedback modality," *NeuroImage*, vol. 111, pp. 1–11, 2015. DOI: 10.1016/j.neuroimage.2015.01.058.

[20] L. S. Hooi, H. Nisar, and Y. V. Voon, "Tracking of eeg activity using topographic maps," in *2015 IEEE International Conference on Signal and Image Processing Applications (ICSIPA)*, 2015, pp. 287–291. DOI: 10.1109/ICSIPA.2015.7412206.

[21] S. Beniczky and P. Sharma, "Clinical electroencephalography," in Springer, 2019, ch. Electromagnetic Source Imaging, High-Density EEG and MEG. DOI: 10.1007/978-3-030-04573-9_20.

[22] M. T. Foged, M. Scherg, M. Fabricius, and S. Beniczky, "Learn to interpret voltage maps: An atlas of topographies," *Epileptic Disorders*, vol. 24, no. 2, pp. 229–248, 2022. DOI: 10.1684/epd.2021.1396.

[23] G. Pfurtscheller and F. L. da Silva, "Event-related EEG/MEG synchronization and desynchronization: Basic principles," *Clinical Neurophysiology*, vol. 110, no. 11, pp. 1842–1857, Nov. 1999. DOI: 10.1016/s1388-2457(99)00141-8. [Online]. Available: https://doi.org/10.1016/s1388-2457(99)00141-8.

[24] *Openvibe designer 64 bit*, Computer software, version 3.5.0, Inria & Mensia Technologies SA, 2023. [Online]. Available: http://openvibe.inria.fr/.

[25] E. C. Lalor, S. P. Kelly, C. Finucane, *et al.*, "Steady-state VEP-based brain-computer interface control in an immersive 3d gaming environment," *EURASIP Journal on Advances in Signal Processing*, vol. 2005, no. 19, Nov. 2005. DOI: 10.1155/asp.2005.3156. [Online]. Available: https://doi.org/10.1155/asp.2005.3156.

[26] J. Kalra, P. Mittal, N. Mittal, *et al.*, "How visual stimuli evoked p300 is transforming the brain–computer interface landscape: A prisma compliant systematic review," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 31, pp. 1429–1439, 2023. DOI: 10.1109/TNSRE.2023.3246588.

[27] G. F. Woodman, "A brief introduction to the use of event-related potentials in studies of perception and attention.," *Attention, perception psychophysics*, vol. 72, no. 8, pp. 2031–2046, 2010. DOI: 10.3758/APP.72.8.2031.

[28] R. Singla, "Evolving bci therapy - engaging brain state dynamics," in IntechOpen, 2018, ch. SSVEP-Based BCIs. DOI: 10.5772/intechopen.75693.

[29] A. M. Norcia, L. G. Appelbaum, J. M. Ales, B. R. Cottereau, and B. Rossion, "The steady-state visual evoked potential in vision research: A review," *Journal of Vision*, vol. 15, no. 6, p. 4, 2015. DOI: 10.1167/15.6.4.

[30] M. Labecki, R. Kus, A. Brzozowska1, T. Stacewicz, B. S. Bhattacharya, and P. Suffczynski, "Nonlinear origin of ssvep spectra—a combined experimental and modeling study," *Frontiers in Computational Neuroscience*, vol. 10, no. 129, 2016. DOI: 10.3389/fncom.2016.00129.

[31] D. Zhu, J. Bieger, G. G. Molina, and R. M. Aarts, “A survey of stimulation methods used in ssvep-based bcis,” *Computational Intelligence and Neuroscience*, vol. 2010, no. Article ID 702357, 2010. DOI: 10.1155/2010/702357.

[32] Y. Xu, “The posterior parietal cortex in adaptive visual processing,” *Trends in Neurosciences*, vol. 41, no. 11, pp. 806–822, Nov. 2018. DOI: 10.1016/j.tins.2018.07.012. [Online]. Available: https://doi.org/10.1016/j.tins.2018.07.012.

[33] *Frontal Eye Field - an overview | ScienceDirect Topics — sciencedirect.com*, https://www.sciencedirect.com/topics/medicine-and-dentistry/frontal-eye-field, [Accessed 15-Jun-2023].

[34] *Supplementary Eye Field - an overview | ScienceDirect Topics — sciencedirect.com*, https://www.sciencedirect.com/topics/medicine-and-dentistry/supplementary-eye-field, [Accessed 15-Jun-2023].

[35] Z. Lin, C. Zhang, W. Wu, and X. Gao, “Frequency recognition based on canonical correlation analysis for SSVEP-based BCIs,” *IEEE Transactions on Biomedical Engineering*, vol. 53, no. 12, pp. 2610–2614, Dec. 2006. DOI: 10.1109/tbme.2006.886577. [Online]. Available: https://doi.org/10.1109/tbme.2006.886577.

[36] A. F. Rossi, L. Pessoa, R. Desimone, and L. G. Ungerleider, “The prefrontal cortex and the executive control of attention,” *Experimental Brain Research*, vol. 192, no. 3, pp. 489–497, Nov. 2008. DOI: 10.1007/s00221-008-1642-z. [Online]. Available: https://doi.org/10.1007/s00221-008-1642-z.

[37] S. Ray, E. Niebur, S. S. Hsiao, A. Sinai, and N. E. Crone, “High-frequency gamma activity (80–150hz) is increased in human cortex during selective attention,” *Clinical Neurophysiology*, vol. 119, no. 1, pp. 116–133, Jan. 2008. DOI: 10.1016/j.clinph.2007.09.136. [Online]. Available: https://doi.org/10.1016/j.clinph.2007.09.136.

[38] Y. Wang, X. Chen, X. Gao, and S. Gao, “A benchmark dataset for SSVEP-based brain–computer interfaces,” *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 25, no. 10, pp. 1746–1752, Oct. 2017. DOI: 10.1109/tnsre.2016.2627556. [Online]. Available: https://doi.org/10.1109/tnsre.2016.2627556.

[39] U. Martens, P. Wahl, U. Hassler, U. Friese, and T. Gruber, “Implicit and explicit contributions to object recognition: Evidence from rapid perceptual learning,” *PLoS ONE*, vol. 7, no. 10, M. Ptito, Ed., e47009, Oct. 2012. DOI: 10.1371/journal.pone.0047009. [Online]. Available: https://doi.org/10.1371/journal.pone.0047009.

A

Python scripts for the interface

A.1. Initial interface code

```
import ctypes
import random
import sys
from time import sleep

import matplotlib
import mne
import pygame
from matplotlib import pyplot as plt
from matplotlib.lines import Line2D
from matplotlib.pyplot import subplots

ctypes.windll.shcore.SetProcessDpiAwareness(1)
pygame.init()
pygame.font.init()
info = pygame.display.Info()
screen_width, screen_height = info.current_w, info.current_h
screen = pygame.display.set_mode((screen_width, screen_height))

# Global variables
playerScore = 0
AIScore = 0
playerLives = 3
gameState = 'start_menu'
sensor_change = 1

# Color definitions
black = (0, 0, 0)
white = (255, 255, 255)
red = (255, 0, 0)
blue = (0, 0, 255)
orange = (255, 119, 34)
yellow = (255, 225, 0)
green = (102, 204, 0)

# Create background to overwrite each frame
background = pygame.Surface((screen.get_width(), screen.get_height()))
background.fill(black)

# Frame rate
gameClock = pygame.time.Clock()
FPS = 120

# Speeds
ballSpeed = 3
rectangleSpeed = 6
cursorSpeed = 2
```

```python

# Object dimensions
ballWidth = 15
ballHeight = 15
paddleWidth = 10
paddleHeight = 100

BrickWidth = 120
BrickHeight = 50

# Check type of collision between the ball and a brick
def check_which_collision(ball, brick):
    if ball.speedX > 0:
        delta_x = ball.rect.right - brick.left
    else:
        delta_x = brick.right - ball.rect.left
    if ball.speedY > 0:
        delta_y = ball.rect.bottom - brick.top
    else:
        delta_y = brick.bottom - ball.rect.top
    if abs(delta_x - delta_y) < 10:
        ball.speedX *= -1
        ball.speedY *= -1
    if delta_x > delta_y:
        ball.speedY *= -1
    elif delta_x < delta_y:
        ball.speedX *= -1


# Class for paddles
class Rectangle(pygame.sprite.Sprite):
    def __init__(self, color, width, height, speed, posX, posY):
        super().__init__()
        self.color = color
        self.width = width
        self.height = height
        self.speed = speed
        self.posX = posX
        self.posY = posY
        self.rect = pygame.Rect(posX, posY, width, height)

        self.image = pygame.Surface([width, height])
        self.image.fill(black)
        self.image.set_colorkey(black)

        pygame.draw.rect(self.image, color, [0, 0, width, height])

    # Move the paddle to the left but no further than the left field boundary
    def move_player_left(self):
        if self.rect.left > 0:
            self.rect.x -= self.speed
        return

    # Move the paddle to the right but no further than the right field boundary
    def move_player_right(self):
        if self.rect.right < screen_width:
            self.rect.x += self.speed
        return

    # Move the paddle up but no further than the upper field boundary
    def move_player_up(self):
        if self.rect.top > 0:
            self.rect.y -= self.speed
        return

    # Move the paddle down but no further than the lower field boundary
    def move_player_down(self):
        if self.rect.bottom < screen_height:
            self.rect.y += self.speed
        return

```

```

# Class for the ball
class Ball(pygame.sprite.Sprite):
    def __init__(self, color, size, speedX, speedY, posX, posY, width, height):
        super().__init__()
        self.color = color
        self.size = size
        self.speedX = speedX
        self.speedY = speedY
        self.posX = posX
        self.posY = posY
        self.width = width
        self.height = height
        self.rect = pygame.Rect(posX, posY, size, size)
        self.invisibleRect = pygame.Rect(posX, posY, size, size)
        self.speedX = speedX
        self.playerCollisions = 0

        self.image = pygame.Surface([width, height])
        self.image.fill(black)
        self.image.set_colorkey(black)

        pygame.draw.ellipse(self.image, white, [0, 0, width, height])

    # Update the ball for pong based on events
    def updateBall_pong(self, player, AI):
        global AIScore
        global playerScore

        # Update ball speed based on which collision:
        # - Collision with upper boundary or lower boundary
        if self.rect.top < 0 or self.rect.bottom > screen_height:
            self.speedY *= -1

        # - Collision with left or right boundary --> player or AI gets a point, ball resets
            to starting position
        elif self.rect.left < 0:
            AIScore += 1
            self.reset()
        elif self.rect.right > screen_width:
            playerScore += 1
            self.reset()

        # - Collision with player's or AI's paddle
        elif self.rect.colliderect(AI) or self.rect.colliderect(player):
            self.speedX *= -1

        # Update ball position based on speed
        self.rect.x += self.speedX
        self.rect.y += self.speedY
        return

    # Update the ball for breakout based on events
    def updateBall_breakout(self, player):
        global playerLives

        # Update ball speed based on which collision:
        # - Collision with upper boundary
        if self.rect.top < 0:
            self.speedY *= -1

        # - Collision with lower boundary --> player loses a life, ball resets to starting
            position
        elif self.rect.bottom > screen_height:
            playerLives -= 1
            self.reset()

        # - Collision with left or right boundary
        elif self.rect.left < 0 or self.rect.right > screen_width:
            self.speedX *= -1

```

```
        # - Collision with player's paddle
        elif self.rect.colliderect(player):
            if self.speedX > 0:
                delta_x = self.rect.right - player.rect.left
            else:
                delta_x = player.rect.right - self.rect.left
            if self.speedY > 0:
                delta_y = self.rect.bottom - player.rect.top
            else:
                delta_y = player.rect.bottom - self.rect.top
            if abs(delta_x - delta_y) < 10:
                self.speedX *= -1
                self.speedY *= -1
            elif delta_x > delta_y:
                self.speedY *= -1
            elif delta_x < delta_y:
                self.speedX *= -1

        # Update ball position based on speed
        self.rect.x += self.speedX
        self.rect.y += self.speedY
        return

    # Reset ball to starting position, launch ball into random direction
    def reset(self):
        sleep(2)
        self.rect = pygame.Rect(self.posX, self.posY, self.size, self.size)
        self.speedX *= random.choice([-1, 1])
        self.speedY *= random.choice([-1, 1])


# Class for button (menu options)
class ButtonRect:
    def __init__(self, width, height, posX, posY):
        self.width = width
        self.height = height
        self.posX = posX
        self.posY = posY
        self.invisibleRect = pygame.Rect(posX, posY, width, height)


# Class for text on buttons (menu options)
class ButtonText:
    def __init__(self, color1, color2, posX, posY):
        self.color1 = color1
        self.color2 = color2
        self.color = color1
        self.posX = posX
        self.posY = posY

    # Text is a certain color if cursor hovers over the button
    def Button_hover(self, words):
        self.color = self.color2
        self.Button_render(words)
        return

    # Text is a certain color is cursor does not hover over the button
    def Button_unhover(self, words):
        self.color = self.color1
        self.Button_render(words)
        return

    # Render and blit the text of the button on the screen
    def Button_render(self, words):
        text = font.render(words, True, self.color)
        screen.blit(text, (self.posX, self.posY))
        return


# Lists that contain all the sprites intended for use in the games
all_sprites_list_pong = pygame.sprite.Group()
```

```
all_sprites_list_breakout = pygame.sprite.Group()

# Font sizes
font = pygame.font.Font(None, 36)
large_font = pygame.font.Font(None, 60)


# PONG:
# Render scores for the player and AI
scorePlayer = font.render(str(playerScore), True, red)
scoreAI = font.render(str(AIScore), True, red)

# Create button (menu option) for pong
pongRect = ButtonRect(80, 20, screen_width / 2 - 50, screen_height / 2)
pongText = ButtonText(white, blue, screen_width / 2 - 50, screen_height / 2)
pongText.Button_render('PONG')

# Create objects
player_pong = Rectangle(white, paddleWidth, paddleHeight, rectangleSpeed, 40, screen_height /
     2)
AI = Rectangle(white, paddleWidth, paddleHeight, rectangleSpeed, screen_width - 50,
    screen_height / 2)
ball = Ball(white, 15, ballSpeed, -ballSpeed, screen_width / 2, screen_height / 2, ballWidth,
     ballHeight)
pygame.mouse.set_pos(random.randint(0, screen_width - 20), random.randint(0, screen_height -
    20))
pygame.display.flip()

# Add objects to list
all_sprites_list_pong.add(player_pong)
all_sprites_list_pong.add(AI)
all_sprites_list_pong.add(ball)


# BREAKOUT:
# Render lives of the player
livesPlayer = font.render(str(playerLives), True, red)

# Create button (menu option) for breakout
breakoutRect = ButtonRect(80, 20, screen_width / 2 - 82, screen_height / 2 + 50)
breakoutText = ButtonText(white, blue, screen_width / 2 - 82, screen_height / 2 + 50)
breakoutText.Button_render('BREAKOUT')

# Create objects
player_breakout = Rectangle(white, paddleHeight, paddleWidth, rectangleSpeed, screen_width /
    2, screen_height - 40)
pygame.mouse.set_pos(random.randint(0, screen_width - 20), random.randint(0, screen_height -
    20))
pygame.display.flip()

brick_colors = [red, orange, yellow, green, blue]

# Add objects to list
brick_list = [pygame.Rect(40 + i * (BrickWidth + 10), 40 + j * (BrickHeight + 10), BrickWidth
    , BrickHeight) for i in
              range(14) for j in range(4)]
all_sprites_list_breakout.add(player_breakout)
all_sprites_list_breakout.add(ball)


# CALIBRATION:
# Create button (menu option) for calibration screen
calRect = ButtonRect(80, 20, screen_width / 2 - 100, screen_height / 2 + 100)
calibrationText = ButtonText(white, blue, screen_width / 2 - 100, screen_height / 2 + 100)
calibrationText.Button_render('CALIBRATION')

# Create montage and info
standard_montage = mne.channels.make_standard_montage('biosemi16')
n_channels = len(standard_montage.ch_names)
info = mne.create_info(standard_montage.ch_names, 250, 'eeg')
info.set_montage(standard_montage)
```

```python
# channel_names = ['Fp1', 'Fp2', 'F4', 'Fz', 'F3', 'T7', 'C3', 'Cz', 'C4', 'T8', 'P4', 'Pz',
    'P3', 'O1', 'Oz', 'O2']
channel_groups = [[0, 1, 2], [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]

# Create legend
legend_elements = [Line2D([0], [0], marker='o', label='Connected sensor',
                          markerfacecolor="#000080", markersize=22),
                   Line2D([0], [0],
                          lw=0),
                   Line2D([0], [0], marker='o', label='Unconnected sensor',
                          markerfacecolor="#800000", markersize=22)
                   ]

# Create display of electrode impedances
impRect = ButtonRect(80, 20, 1200, 150)
impText = ButtonText(white, white, 1200, 150)
impText.Button_render('Electrode impedances')

fp1Rect = ButtonRect(80, 20, 1150, 200)
fp1Text = ButtonText(white, white, 1150, 200)
fp1Text.Button_render('Fp1:')

fp2Rect = ButtonRect(80, 20, 1150, 250)
fp2Text = ButtonText(white, white, 1150, 250)
fp2Text.Button_render('Fp2:')

f4Rect = ButtonRect(80, 20, 1150, 300)
f4Text = ButtonText(white, white, 1150, 300)
f4Text.Button_render('F4:')


# Game loop
while True:
    gameClock.tick(FPS)

    # Exit interface if escape is pressed
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()


    # START MENU:
    if gameState == 'start_menu':

        # Exit interface if escape is pressed
        keys = pygame.key.get_pressed()
        if keys[pygame.K_ESCAPE]:
            pygame.quit()
            sys.exit()

        # If mouse hovers over an option, let its text change color
        # Go to the respective game state if the option gets selected
        mousePOS = pygame.mouse.get_pos()
        if pongRect.invisibleRect.collidepoint(mousePOS[0], mousePOS[1]):
            pongText.Button_hover('PONG')
            breakoutText.Button_unhover('BREAKOUT')
            calibrationText.Button_unhover('CALIBRATION')
            mouse_pressed = pygame.mouse.get_pressed(num_buttons=3)[0]
            if mouse_pressed:
                gameState = 'pong_game'
        elif breakoutRect.invisibleRect.collidepoint(mousePOS[0], mousePOS[1]):
            breakoutText.Button_hover('BREAKOUT')
            pongText.Button_unhover('PONG')
            calibrationText.Button_unhover('CALIBRATION')
            mouse_pressed = pygame.mouse.get_pressed(num_buttons=3)[0]
            if mouse_pressed:
                gameState = 'breakout_game'
        elif calRect.invisibleRect.collidepoint(mousePOS[0], mousePOS[1]):
            breakoutText.Button_unhover('BREAKOUT')
            pongText.Button_unhover('PONG')
```

```
            calibrationText.Button_hover('CALIBRATION')
            mouse_pressed = pygame.mouse.get_pressed(num_buttons=3)[0]
            if mouse_pressed:
                gameState = 'calibration'
        else:
            pongText.Button_unhover('PONG')
            breakoutText.Button_unhover('BREAKOUT')
            calibrationText.Button_unhover('CALIBRATION')

        # Display the buttons (menu options)
        screen.blit(background, (0, 0))
        breakoutText.Button_render('BREAKOUT')
        pongText.Button_render('PONG')
        calibrationText.Button_render('CALIBRATION')
        pygame.display.flip()


    # CALIBRATION SCREEN:
    elif gameState == 'calibration':

        # Exit interface if escape is pressed, go back to start menu if backspace is pressed
        keys = pygame.key.get_pressed()
        if keys[pygame.K_ESCAPE]:
            pygame.quit()
            sys.exit()
        if keys[pygame.K_BACKSPACE]:
            gameState == 'start_menu'

        # Plot the electrode layout
        if sensor_change:
            fig, ax = subplots(1,1)
            mne.viz.plot_sensors(info, show_names=True, ch_groups=channel_groups, linewidth=0.5, show=False, axes=ax)
            plt.legend(handles=legend_elements, loc='upper right', bbox_to_anchor=(1.2,1.1), frameon= False)
            plt.savefig("sensors.png", dpi=150)
            sensor_change = 0
            sensors = pygame.image.load("sensors.png")

        # Display the electrode impedances
        screen.blit(background, (0, 0))
        screen.blit(sensors, (20, 200))
        impText.Button_render('Electrode impedances')
        fp1Text.Button_render('Fp1:')
        fp2Text.Button_render('Fp2:')
        f4Text.Button_render('F4:')
        pygame.display.flip()


    # PONG:
    elif gameState == 'pong_game':

        keys = pygame.key.get_pressed()
        # Exit interface if escape is pressed, go back to start menu if backspace is pressed
        # Move player's paddle up if up key is pressed and down if down key is pressed
        if keys[pygame.K_DOWN]:
            player_pong.move_player_down()
        if keys[pygame.K_UP]:
            player_pong.move_player_up()
        if keys[pygame.K_BACKSPACE]:
            gameState = 'start_menu'
        if keys[pygame.K_ESCAPE]:
            pygame.quit()
            sys.exit()

        # Introduce imperfections in the AI
        # - AI predicts the ball position if the ball is going towards the AI's paddle
        #   and passed the first quarter of the screen
        # - AI executes a movement 70% of the time
        if ball.speedX > 0 and ball.rect.x > 0.25 * screen_width:
            if ball.rect.y < AI.rect.top and AI.rect.top > 0:
```

```
                bias = random.randint(1, 10)
                if bias <= 7:
                    AI.move_player_up()
            if ball.rect.y > AI.rect.bottom and AI.rect.bottom < screen_height:
                bias = random.randint(1, 10)
                if bias <= 7:
                    AI.move_player_down()

        # Print all updated objects on the screen
        screen.blit(background, (0, 0))
        scorePlayer = font.render(str(playerScore), True, red)
        scoreAI = font.render(str(AIScore), True, red)
        screen.blit(scorePlayer, (0.25 * screen_width, 0.1 * screen_height))
        screen.blit(scoreAI, (0.75 * screen_width, 0.1 * screen_height))
        all_sprites_list_pong.update()
        all_sprites_list_pong.draw(screen)
        ball.updateBall_pong(player_pong, AI)

        # Go to the end screen if either the player or AI reaches a score of 3
        if AIScore == 3 or playerScore == 3:
            gameState = 'end_screen'

        pygame.display.flip()


    # BREAKOUT:
    elif gameState == 'breakout_game':

        # Exit interface if escape is pressed, go back to start menu if backspace is pressed
        # Move player's paddle left if left key is pressed and right if right key is pressed
        keys = pygame.key.get_pressed()
        if keys[pygame.K_LEFT]:
            player_breakout.move_player_left()
        if keys[pygame.K_RIGHT]:
            player_breakout.move_player_right()
        if keys[pygame.K_BACKSPACE]:
            gameState = 'start_menu'
        if keys[pygame.K_ESCAPE]:
            pygame.quit()
            sys.exit()

        # Remove a brick if it has been hit with the ball
        hit_index = ball.rect.collidelist(brick_list)
        if 0 <= hit_index <= 14 * 4:
            check_which_collision(ball, brick_list[hit_index])
            brick_list.pop(hit_index)

        # Print all updated objects on the screen
        screen.blit(background, (0, 0))
        livesPlayer = font.render(str(playerLives), True, red)
        screen.blit(livesPlayer, (screen_width - 40, screen_height - 40))
        all_sprites_list_breakout.update()
        ball.updateBall_breakout(player_breakout)
        all_sprites_list_breakout.draw(screen)
        [pygame.draw.rect(screen, brick_colors[0], brick_list[i]) for i in range(len(
            brick_list))]

        # Go to the end screen if either the player has no lives left or all the brick have
            been eliminated
        if playerLives == 0 or len(brick_list) == 0:
            gameState = 'end_screen'

        pygame.display.flip()


    # END SCREEN:
    elif gameState == 'end_screen':

        # Create a message with the result of the game
        if playerLives == 0:
            message1 = "You lost"
```

```
            message2 = "Score:"
            message3 = "  {}".format(str(11 * 4 - len(brick_list)))
        elif len(brick_list) == 0:
            message1 = "You won!"
            message2 = ""
            message3 = ""
        elif AIScore == 3:
            message1 = "You lost"
            message2 = "Score"
            message3 = "{} - {}".format(playerScore, AIScore)
        elif playerScore == 3:
            message1 = "You won!"
            message2 = "Score"
            message3 = "{} - {}".format(playerScore, AIScore)

        # Render messages
        text1 = large_font.render(message1, True, red)
        text2 = font.render(message2, True, red)
        text3 = font.render(message3, True, red)

        # Display messages
        screen.blit(background, (0, 0))
        screen.blit(text1, (screen_width / 2 - 100, screen_height / 2 - 100))
        screen.blit(text2, (screen_width / 2 - 50, screen_height / 2))
        screen.blit(text3, (screen_width / 2 - 40, screen_height / 2 + 50))

        pygame.display.flip()
```

A.2. Concurrent interface code

```
import threading
import multiprocessing
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from time import sleep

import ctypes
import random
import sys
from time import sleep

import matplotlib
import mne
import pygame
from matplotlib import pyplot as plt
from matplotlib.lines import Line2D
from matplotlib.pyplot import subplots


# Check type of collision between the ball and a brick
def check_which_collision(ball, brick):
    if ball.speedX > 0:
        delta_x = ball.rect.right - brick.left
    else:
        delta_x = brick.right - ball.rect.left
    if ball.speedY > 0:
        delta_y = ball.rect.bottom - brick.top
    else:
        delta_y = brick.bottom - ball.rect.top
    if abs(delta_x - delta_y) < 10:
        ball.speedX *= -1
        ball.speedY *= -1
    if delta_x > delta_y:
        ball.speedY *= -1
    elif delta_x < delta_y:
        ball.speedX *= -1


# Class for paddles
```

```
class Rectangle(pygame.sprite.Sprite):
    def __init__(self, color, width, height, speed, posX, posY):
        super().__init__()
        self.color = color
        self.width = width
        self.height = height
        self.speed = speed
        self.posX = posX
        self.posY = posY
        self.rect = pygame.Rect(posX, posY, width, height)

        self.image = pygame.Surface([width, height])
        self.image.fill(black)
        self.image.set_colorkey(black)

        pygame.draw.rect(self.image, color, [0, 0, width, height])

    # Move the paddle to the left but no further than the left field boundary
    def move_player_left(self):
        if self.rect.left > 0:
            self.rect.x -= self.speed
        return

    # Move the paddle to the right but no further than the right field boundary
    def move_player_right(self):
        if self.rect.right < screen_width:
            self.rect.x += self.speed
        return

    # Move the paddle up but no further than the upper field boundary
    def move_player_up(self):
        if self.rect.top > 0:
            self.rect.y -= self.speed
        return

    # Move the paddle down but no further than the lower field boundary
    def move_player_down(self):
        if self.rect.bottom < screen_height:
            self.rect.y += self.speed
        return


# Class for the ball
class Ball(pygame.sprite.Sprite):
    def __init__(self, color, size, speedX, speedY, posX, posY, width, height):
        super().__init__()
        self.color = color
        self.size = size
        self.speedX = speedX
        self.speedY = speedY
        self.posX = posX
        self.posY = posY
        self.width = width
        self.height = height
        self.rect = pygame.Rect(posX, posY, size, size)
        self.invisibleRect = pygame.Rect(posX, posY, size, size)
        self.speedX = speedX
        self.playerCollisions = 0

        self.image = pygame.Surface([width, height])
        self.image.fill(black)
        self.image.set_colorkey(black)

        pygame.draw.ellipse(self.image, white, [0, 0, width, height])

    # Update the ball for pong based on events
    def updateBall_pong(self, player, AI):
        global AIScore
        global playerScore
        # Update ball speed based on which collision:
        # - Collision with upper boundary or lower boundary
```

```
        if self.rect.top < 0 or self.rect.bottom > screen_height:
            self.speedY *= -1

        # - Collision with left or right boundary --> player or AI gets a point, ball resets to starting position
        elif self.rect.left < 0:
            AIScore += 1
            self.reset()
        elif self.rect.right > screen_width:
            playerScore += 1
            self.reset()

        # - Collision with player's or AI's paddle
        elif self.rect.colliderect(AI) or self.rect.colliderect(player):
            self.speedX *= -1

        # Update ball position based on speed
        self.rect.x += self.speedX
        self.rect.y += self.speedY
        return

    # Update the ball for breakout based on events
    def updateBall_breakout(self, player):
        global playerLives
        # Update ball speed based on which collision:
        # - Collision with upper boundary
        if self.rect.top < 0:
            self.speedY *= -1

        # - Collision with lower boundary --> player loses a life, ball resets to starting position
        elif self.rect.bottom > screen_height:
            playerLives -= 1
            self.reset()

        # - Collision with left or right boundary
        elif self.rect.left < 0 or self.rect.right > screen_width:
            self.speedX *= -1

        # - Collision with player's paddle
        elif self.rect.colliderect(player):
            if self.speedX > 0:
                delta_x = self.rect.right - player.rect.left
            else:
                delta_x = player.rect.right - self.rect.left
            if self.speedY > 0:
                delta_y = self.rect.bottom - player.rect.top
            else:
                delta_y = player.rect.bottom - self.rect.top
            if abs(delta_x - delta_y) < 10:
                self.speedX *= -1
                self.speedY *= -1
            elif delta_x > delta_y:
                self.speedY *= -1
            elif delta_x < delta_y:
                self.speedX *= -1

        # Update ball position based on speed
        self.rect.x += self.speedX
        self.rect.y += self.speedY
        return

    # Reset ball to starting position, launch ball into random direction
    def reset(self):
        sleep(2)
        self.rect = pygame.Rect(self.posX, self.posY, self.size, self.size)
        self.speedX *= random.choice([-1, 1])
        self.speedY *= random.choice([-1, 1])


# Class for button (menu options)
```

```
class ButtonRect:
    def __init__(self, width, height, posX, posY):
        self.width = width
        self.height = height
        self.posX = posX
        self.posY = posY
        self.invisibleRect = pygame.Rect(posX, posY, width, height)


# Class for text on buttons (menu options)
class ButtonText:
    def __init__(self, color1, color2, posX, posY):
        self.color1 = color1
        self.color2 = color2
        self.color = color1
        self.posX = posX
        self.posY = posY

    # Text is a certain color if cursor hovers over the button
    def Button_hover(self, words):
        self.color = self.color2
        self.Button_render(words)
        return

    # Text is a certain color is cursor does not hover over the button
    def Button_unhover(self, words):
        self.color = self.color1
        self.Button_render(words)
        return

    # Render and blit the text of the button on the screen
    def Button_render(self, words):

        text = font.render(words, True, self.color)
        screen.blit(text, (self.posX, self.posY))
        return


def interface():
    ctypes.windll.shcore.SetProcessDpiAwareness(1)
    pygame.init()
    pygame.font.init()
    info = pygame.display.Info()
    global screen_width
    global screen_height
    screen_width, screen_height = info.current_w, info.current_h
    global screen
    screen = pygame.display.set_mode((screen_width, screen_height))

    # Global variables
    global playerScore
    global AIScore
    global playerLives
    playerScore = 0
    AIScore = 0
    playerLives = 3
    gameState = 'start_menu'
    sensor_change = 1

    # Color definitions
    global black, white, red, blue, orange, yellow, green
    black = (0, 0, 0)
    white = (255, 255, 255)
    red = (255, 0, 0)
    blue = (0, 0, 255)
    orange = (255, 119, 34)
    yellow = (255, 225, 0)
    green = (102, 204, 0)

    # Create background to overwrite each frame
    background = pygame.Surface((screen.get_width(), screen.get_height()))
```

```
    background.fill(black)

    # Frame rate
    gameClock = pygame.time.Clock()
    FPS = 120

    # Speeds
    ballSpeed = 3
    rectangleSpeed = 6
    cursorSpeed = 2

    # Object dimensions
    ballWidth = 15
    ballHeight = 15
    paddleWidth = 10
    paddleHeight = 100

    BrickWidth = 120
    BrickHeight = 50

    # Lists that contain all the sprites intended for use in the games
    all_sprites_list_pong = pygame.sprite.Group()
    all_sprites_list_breakout = pygame.sprite.Group()

    # Font sizes
    global font
    font = pygame.font.Font(None, 36)
    large_font = pygame.font.Font(None, 60)

    # PONG:
    # Render scores for the player and AI
    scorePlayer = font.render(str(playerScore), True, red)
    scoreAI = font.render(str(AIScore), True, red)

    # Create button (menu option) for pong
    pongRect = ButtonRect(80, 20, screen_width / 2 - 50, screen_height / 2)
    pongText = ButtonText(white, blue, screen_width / 2 - 50, screen_height / 2)
    pongText.Button_render('PONG')

    # Create objects
    player_pong = Rectangle(white, paddleWidth, paddleHeight, rectangleSpeed, 40,
        screen_height / 2)
    AI = Rectangle(white, paddleWidth, paddleHeight, rectangleSpeed, screen_width - 50,
        screen_height / 2)
    ball = Ball(white, 15, ballSpeed, -ballSpeed, screen_width / 2, screen_height / 2,
        ballWidth, ballHeight)
    pygame.mouse.set_pos(random.randint(0, screen_width - 20), random.randint(0,
        screen_height - 20))
    pygame.display.flip()

    # Add objects to list
    all_sprites_list_pong.add(player_pong)
    all_sprites_list_pong.add(AI)
    all_sprites_list_pong.add(ball)

    # BREAKOUT:
    # Render lives of the player
    livesPlayer = font.render(str(playerLives), True, red)

    # Create button (menu option) for breakout
    breakoutRect = ButtonRect(80, 20, screen_width / 2 - 82, screen_height / 2 + 50)
    breakoutText = ButtonText(white, blue, screen_width / 2 - 82, screen_height / 2 + 50)
    breakoutText.Button_render('BREAKOUT')

    # Create objects
    player_breakout = Rectangle(white, paddleHeight, paddleWidth, rectangleSpeed,
        screen_width / 2, screen_height - 40)
    pygame.mouse.set_pos(random.randint(0, screen_width - 20), random.randint(0,
        screen_height - 20))
    pygame.display.flip()
```

```
    brick_colors = [red, orange, yellow, green, blue]

    # Add objects to list
    brick_list = [pygame.Rect(40 + i * (BrickWidth + 10), 40 + j * (BrickHeight + 10), BrickWidth, BrickHeight) for i in
                  range(14) for j in range(4)]
    all_sprites_list_breakout.add(player_breakout)
    all_sprites_list_breakout.add(ball)

    # CALIBRATION:
    # Create button (menu option) for calibration screen
    calRect = ButtonRect(80, 20, screen_width / 2 - 100, screen_height / 2 + 100)
    calibrationText = ButtonText(white, blue, screen_width / 2 - 100, screen_height / 2 + 100)
    calibrationText.Button_render('CALIBRATION')

    # Create montage and info
    standard_montage = mne.channels.make_standard_montage('biosemi16')
    n_channels = len(standard_montage.ch_names)
    info = mne.create_info(standard_montage.ch_names, 250, 'eeg')
    info.set_montage(standard_montage)
    # channel_names = ['Fp1', 'Fp2', 'F4', 'Fz', 'F3', 'T7', 'C3', 'Cz', 'C4', 'T8', 'P4', 'Pz', 'P3', 'O1', 'Oz', 'O2']
    channel_groups = [[0, 1, 2], [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]

    # Create legend
    legend_elements = [Line2D([0], [0], marker='o', label='Connected sensor',
                              markerfacecolor="#000080", markersize=22),
                       Line2D([0], [0],
                              lw=0),
                       Line2D([0], [0], marker='o', label='Unconnected sensor',
                              markerfacecolor="#800000", markersize=22)
                       ]

    # Create display of electrode impedances
    impRect = ButtonRect(80, 20, 1200, 150)
    impText = ButtonText(white, white, 1200, 150)
    impText.Button_render('Electrode impedances')

    fp1Rect = ButtonRect(80, 20, 1150, 200)
    fp1Text = ButtonText(white, white, 1150, 200)
    fp1Text.Button_render('Fp1:')

    fp2Rect = ButtonRect(80, 20, 1150, 250)
    fp2Text = ButtonText(white, white, 1150, 250)
    fp2Text.Button_render('Fp2:')

    f4Rect = ButtonRect(80, 20, 1150, 300)
    f4Text = ButtonText(white, white, 1150, 300)
    f4Text.Button_render('F4:')

    # Game loop
    while True:
        gameClock.tick(FPS)

        # Exit interface if escape is pressed
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()

        # START MENU:
        if gameState == 'start_menu':

            # Exit interface if escape is pressed
            keys = pygame.key.get_pressed()
            if keys[pygame.K_ESCAPE]:
                pygame.quit()
                sys.exit()

            # If mouse hovers over an option, let its text change color
```

```
            # Go to the respective game state if the option gets selected
            mousePOS = pygame.mouse.get_pos()
            if pongRect.invisibleRect.collidepoint(mousePOS[0], mousePOS[1]):
                pongText.Button_hover('PONG')
                breakoutText.Button_unhover('BREAKOUT')
                calibrationText.Button_unhover('CALIBRATION')
                mouse_pressed = pygame.mouse.get_pressed(num_buttons=3)[0]
                if mouse_pressed:
                    gameState = 'pong_game'
            elif breakoutRect.invisibleRect.collidepoint(mousePOS[0], mousePOS[1]):
                breakoutText.Button_hover('BREAKOUT')
                pongText.Button_unhover('PONG')
                calibrationText.Button_unhover('CALIBRATION')
                mouse_pressed = pygame.mouse.get_pressed(num_buttons=3)[0]
                if mouse_pressed:
                    gameState = 'breakout_game'
            elif calRect.invisibleRect.collidepoint(mousePOS[0], mousePOS[1]):
                breakoutText.Button_unhover('BREAKOUT')
                pongText.Button_unhover('PONG')
                calibrationText.Button_hover('CALIBRATION')
                mouse_pressed = pygame.mouse.get_pressed(num_buttons=3)[0]
                if mouse_pressed:
                    gameState = 'calibration'
            else:
                pongText.Button_unhover('PONG')
                breakoutText.Button_unhover('BREAKOUT')
                calibrationText.Button_unhover('CALIBRATION')

            # Display the buttons (menu options)
            screen.blit(background, (0, 0))
            breakoutText.Button_render('BREAKOUT')
            pongText.Button_render('PONG')
            calibrationText.Button_render('CALIBRATION')
            pygame.display.flip()


        # CALIBRATION SCREEN:
        elif gameState == 'calibration':

            # Exit interface if escape is pressed, go back to start menu if backspace is pressed
            keys = pygame.key.get_pressed()
            if keys[pygame.K_ESCAPE]:
                pygame.quit()
                sys.exit()
            if keys[pygame.K_BACKSPACE]:
                gameState == 'start_menu'

            # Plot the electrode layout
            if sensor_change:
                fig, ax = subplots(1, 1)
                mne.viz.plot_sensors(info, show_names=True, ch_groups=channel_groups, linewidth=0.5, show=False,
                                     axes=ax)
                plt.legend(handles=legend_elements, loc='upper right', bbox_to_anchor=(1.2, 1.1), frameon=False)
                plt.savefig("sensors.png", dpi=150)
                sensor_change = 0
                sensors = pygame.image.load("sensors.png")

            # Display the electrode impedances
            screen.blit(background, (0, 0))
            screen.blit(sensors, (20, 200))
            impText.Button_render('Electrode impedances')
            fp1Text.Button_render('Fp1:')
            fp2Text.Button_render('Fp2:')
            f4Text.Button_render('F4:')
            pygame.display.flip()


        # PONG:
```

```
        elif gameState == 'pong_game':

            keys = pygame.key.get_pressed()
            # Exit interface if escape is pressed, go back to start menu if backspace is pressed
            # Move player's paddle up if up key is pressed and down if down key is pressed
            if keys[pygame.K_DOWN]:
                player_pong.move_player_down()
            if keys[pygame.K_UP]:
                player_pong.move_player_up()
            if keys[pygame.K_BACKSPACE]:
                gameState = 'start_menu'
            if keys[pygame.K_ESCAPE]:
                pygame.quit()
                sys.exit()

            # Introduce imperfections in the AI
            # - AI predicts the ball position if the ball is going towards the AI's paddle
            #   and passed the first quarter of the screen
            # - AI executes a movement 70% of the time
            if ball.speedX > 0 and ball.rect.x > 0.25 * screen_width:
                if ball.rect.y < AI.rect.top and AI.rect.top > 0:
                    bias = random.randint(1, 10)
                    if bias <= 7:
                        AI.move_player_up()
                if ball.rect.y > AI.rect.bottom and AI.rect.bottom < screen_height:
                    bias = random.randint(1, 10)
                    if bias <= 7:
                        AI.move_player_down()

            # Print all updated objects on the screen
            screen.blit(background, (0, 0))
            scorePlayer = font.render(str(playerScore), True, red)
            scoreAI = font.render(str(AIScore), True, red)
            screen.blit(scorePlayer, (0.25 * screen_width, 0.1 * screen_height))
            screen.blit(scoreAI, (0.75 * screen_width, 0.1 * screen_height))
            all_sprites_list_pong.update()
            all_sprites_list_pong.draw(screen)
            ball.updateBall_pong(player_pong, AI)

            # Go to the end screen if either the player or AI reaches a score of 3
            if AIScore == 3 or playerScore == 3:
                gameState = 'end_screen'

            pygame.display.flip()


        # BREAKOUT:
        elif gameState == 'breakout_game':

            # Exit interface if escape is pressed, go back to start menu if backspace is pressed
            # Move player's paddle left if left key is pressed and right if right key is pressed
            keys = pygame.key.get_pressed()
            if keys[pygame.K_LEFT]:
                player_breakout.move_player_left()
            if keys[pygame.K_RIGHT]:
                player_breakout.move_player_right()
            if keys[pygame.K_BACKSPACE]:
                gameState = 'start_menu'
            if keys[pygame.K_ESCAPE]:
                pygame.quit()
                sys.exit()

            # Remove a brick if it has been hit with the ball
            hit_index = ball.rect.collidelist(brick_list)
            if 0 <= hit_index <= 14 * 4:
                check_which_collision(ball, brick_list[hit_index])
                brick_list.pop(hit_index)

```

```
            # Print all updated objects on the screen
            screen.blit(background, (0, 0))
            livesPlayer = font.render(str(playerLives), True, red)
            screen.blit(livesPlayer, (screen_width - 40, screen_height - 40))
            all_sprites_list_breakout.update()
            ball.updateBall_breakout(player_breakout)
            all_sprites_list_breakout.draw(screen)
            [pygame.draw.rect(screen, brick_colors[0], brick_list[i]) for i in range(len(
                brick_list))]

            # Go to the end screen if either the player has no lives left or all the brick
                have been eliminated
            if playerLives == 0 or len(brick_list) == 0:
                gameState = 'end_screen'

            pygame.display.flip()


        # END SCREEN:
        elif gameState == 'end_screen':

            # Create a message with the result of the game
            if playerLives == 0:
                message1 = "You lost"
                message2 = "Score:"
                message3 = "  {}".format(str(11 * 4 - len(brick_list)))
            elif len(brick_list) == 0:
                message1 = "You won!"
                message2 = ""
                message3 = ""
            elif AIScore == 3:
                message1 = "You lost"
                message2 = "Score"
                message3 = "{} - {}".format(playerScore, AIScore)
            elif playerScore == 3:
                message1 = "You won!"
                message2 = "Score"
                message3 = "{} - {}".format(playerScore, AIScore)

            # Render messages
            text1 = large_font.render(message1, True, red)
            text2 = font.render(message2, True, red)
            text3 = font.render(message3, True, red)

            # Display messages
            screen.blit(background, (0, 0))
            screen.blit(text1, (screen_width / 2 - 100, screen_height / 2 - 100))
            screen.blit(text2, (screen_width / 2 - 50, screen_height / 2))
            screen.blit(text3, (screen_width / 2 - 40, screen_height / 2 + 50))

            pygame.display.flip()


def init():
    line.set_data([], [])
    return line,


def animate(i):
    x = np.linspace(0, 4, 1000)
    y = np.sin(2 * np.pi * (x - 0.01 * i))
    line.set_data(x, y)
    return line,


if __name__ == '__main__':
    fig = plt.figure()
    ax = plt.axes(xlim=(0, 4), ylim=(-2, 2))
    line, = ax.plot([], [], lw=3)

    anim = FuncAnimation(fig, animate, init_func=init,
```

```
                         frames=200, interval=20, blit=False)

    # Create and start the print thread
    # interface_thread = threading.Thread(target=interface)
    # interface_thread.start()

    interface_process = multiprocessing.Process(target=interface)
    interface_process.start()
    # Start the animation
    plt.show()

    # Wait for the print thread to finish
    # interface_thread.join()
    interface_process.join()
```

B

Python scripts for plots

B.1. PSD, topomaps and raw EEG plotting

```python

import math

import numpy as np
from matplotlib import pyplot as plt, cm
import mne
import pandas as pd
from scipy import signal
from matplotlib.colors import BoundaryNorm
from matplotlib.ticker import MaxNLocator
import time
import matplotlib.animation as ani


# get correct data columns from csv file by dropping certain elements
def data_csv(path: str, drop: list[str]):
    df = pd.read_csv(path)
    df.drop(drop, axis=1, inplace=True)
    Xt = df.to_numpy()
    t = Xt[:, 0]
    X = Xt[:, 1:]
    n = X.shape[1]
    return t, X  # return time array and data array (n_times, n_channels)


# create info object from MNE library. Info hold information needed for creating MNE raw
    object
def createInfo(channel_names, fs):
    channel_types = 'eeg'
    # montage = 'biosemi16'

    info = mne.create_info(channel_names, fs, channel_types)

    return info


# plot PSD
def plot_PSD(raw):
    raw.compute_psd().plot(picks="data", exclude="bads")
    plt.show()
    return


# plot raw eeg signals
def plot_raw(raw, ch_picks):
    raw.plot(n_channels=len(ch_picks), scalings='auto', title='EEG data',
             show=True, block=False, show_scrollbars=False, duration=5)
```

```
    return


# plot 16 short time averaged topomaps to show behaviour in time
def topomap_temporal(raw):
    x = raw._data
    times_seconds = np.arange(22.4, 22.4 + 17 * 0.1, 0.1)  # Define the time points in
        seconds
    times_samples = (times_seconds * raw.info['sfreq']).astype(int)  # Convert time points to
        sample indices
    n_subplots = min(len(times_samples) - 1, 16)
    fig, axes = plt.subplots(4, 4, figsize=(12, 12))  # Adjust the subplot layout as per your
        preference

    for i in range(n_subplots):
        ax = axes[i // 4, i % 4]  # Adjust the indexing for subplot layout
        t0 = times_samples[i]
        t1 = times_samples[i + 1]
        mne.viz.plot_topomap(x[:, t0:t1].mean(axis=1), raw.info, axes=ax, show=False)
        ax.set_title(f'Time: {times_seconds[i]:.1f}s-{times_seconds[i + 1]:.1f}s')
    return


# plot power topomap at specific frequency
def plot_power_topomap(raw, sfreq):
    x = raw._data
    n_samples = x.shape[1]
    times = np.arange(n_samples) / sfreq

    # power spectrum of all channels
    power_spectrum = np.abs(np.fft.fft(x, axis=1)) ** 2

    # find index of frequency of interest
    frequency_of_interest = 15  # Frequency of interest (Hz)
    freq_index = int(frequency_of_interest * n_samples / sfreq)

    # get power values of freq of interest for all channels
    power_at_freq = power_spectrum[:, freq_index]

    # plot power topomap
    fig, ax = plt.subplots()
    mne.viz.plot_topomap(power_at_freq, raw.info, axes=ax, show=True)
    return


# plot power topomap at specific frequency bands
def plot_PSD_bands_topomap(raw):
    raw.compute_psd().plot_topomap()
    plt.show()
    return


# plot potential topomap averaged over time (you lose temporal dynamics)
def plot_potential_topomap(raw):
    x = raw._data
    fig, ax = plt.subplots(figsize=(10, 10))
    im, cm = mne.viz.plot_topomap(x.mean(axis=1), raw.info, contours=0, axes=ax, show=False)
    # colorbar
    ax_x_start = 0.8
    ax_x_width = 0.04
    ax_y_start = 0.1
    ax_y_height = 0.7
    cbar_ax = fig.add_axes([ax_x_start, ax_y_start, ax_x_width, ax_y_height])
    clb = fig.colorbar(im, cax=cbar_ax)
    clb.ax.set_title('Volt', fontsize=10)
    plt.show()
    return


# plot small segment of eeg data of one channel
def plot_zoomed_in_raw(raw, pick):
```

```
    raw.pick(pick)
    x = raw._data
    x = x[0]  # fix format of x for plt.plot
    # prepare data for plot
    Nx = x.size
    t = np.arange(0, Nx / fs, 1 / fs)
    # plot data
    plt.plot(t, x)
    plt.ylabel('Amplitude  [V]')
    plt.xlabel('Time [s]')
    plt.title('Checkerboard test 01 data segment')
    plt.show()
    return


# create a montage/info needed for plotting
# biosemi16 = standard 16 channel config using 10-20 system
standard_montage = mne.channels.make_standard_montage('biosemi16')
n_channels = len(standard_montage.ch_names)
info = createInfo(channel_names=standard_montage.ch_names, fs=250)
info.set_montage(standard_montage)
# names of biosemi16 channels
ch_names = ['Fp1', 'Fp2', 'F4', 'Fz', 'F3', 'T7', 'C3', 'Cz', 'C4', 'T8', 'P4', 'Pz', 'P3', '
    O1', 'Oz', 'O2']
# measurement group only used these for motor imagery
motor_imagery = [0, 1, 3, 7, 6, 8, 12, 10]
# we only used these for visual functions
visual = [4, 2, 12, 11, 10, 13, 14, 15]
drop = ['Event Id', 'Event Date', 'Event Duration', 'Channel 9',
        'Channel 10', 'Channel 11', 'Epoch']
fs = 250
# time and data array
t, X = data_csv('short_hair_flickering2.csv', drop)
# mne expects data(n_channels, n_times) so need to transpose
x_tr = np.transpose(X)

# mne still expects 16 channels of data due to biosemi 16 config
# create empty array with zeros and fill in the channels we used
data = np.zeros((n_channels, x_tr.shape[1]))
for j, data_idx in enumerate(visual):
    data[data_idx, :] = x_tr[j, :]

# create raw object, contains data and other information
raw = mne.io.RawArray(data, info)
# mne expects data in V not uV so scale the data
raw.apply_function(lambda x: x * 1e-6)

# pick certain channels only
ch_picks = visual
ch_pick_names = ch_names[ch_picks[0]]
raw.pick(ch_picks)

# take only a portion of total data. First 10 to 20s were rest
raw.crop(tmin=25, tmax=50, include_tmax=True)
plot_PSD(raw)
```

C

Python scripts for the topographic maps

C.1. Initial script

```
import numpy as np
from matplotlib import pyplot as plt
import mne
import pandas as pd
import time
import matplotlib.animation as ani

# Function to get the data from our own measurements
# Only works with no labels
def data_csv(path: str, drop: list[str]):
    df = pd.read_csv(path)
    df.drop(drop, axis=1, inplace=True)
    Xt = df.to_numpy()  # Containing time and channel data
    t = Xt[:, 0]
    X = Xt[:, 1:]
    n = X.shape[1]
    return t, X


# Creates info about the sensors and measurement methods
def createInfo(channel_names, fs):
    channel_types = 'eeg'
    info = mne.create_info(channel_names, fs, channel_types)

    return info


# Creates the topomap
def EEG_topo(i):
    ax.cla()
    for j, data_idx in enumerate([0, 1, 3, 7, 6, 8, 12, 10]):
        data[data_idx] = x_tr[j, i]

    evokedArray = mne.EvokedArray(data, info)
    evokedArray.set_montage(standard_montage)

    mne.viz.plot_topomap(evokedArray.data[:, 0], evokedArray.info, axes=ax)

# Creates the montage
standard_montage = mne.channels.make_standard_montage('biosemi16')
n_channels = len(standard_montage.ch_names)
info = createInfo(channel_names=standard_montage.ch_names, fs=250)

# Extracts data from the file and prepares it for the topomaps
```

```
drop = ['Event Id', 'Event Date', 'Event Duration', 'Channel 9', 'Channel 10', 'Channel 11',
    'Epoch']
t, X = data_csv('Thib_lefthand_filtered_1.csv', drop)
x_tr = np.transpose(X)
data = np.zeros((n_channels, 1))

# Creates the animation of the topomaps
fig, ax = plt.subplots(figsize=(10, 10))
animator = ani.FuncAnimation(fig, EEG_topo, frames=15, interval=500)
plt.show()
```

C.2. Script adjusted for OpenVibe

```
import numpy
from matplotlib import pyplot as plt
import matplotlib
import mne
import time


# Box class that inherits from OVBox
class MyOVBox(OVBox):
    def __init__(self):
        OVBox.__init__(self)
        self.signalHeader = None
        self.fig=plt.figure(figsize=(10,10))
        self.standard_montage = mne.channels.make_standard_montage('biosemi16')
        self.n_channels = len(self.standard_montage.ch_names)
        self.info = mne.create_info(ch_names=self.standard_montage.ch_names, sfreq=250,
            ch_types='eeg', verbose=None)
        self.info.set_montage(self.standard_montage)
        self.data = numpy.zeros((self.n_channels,32))

    # The process method will be called by openvibe on every clock tick
    def process(self):
       # Iterate over all the input chunks in the input buffer
       for chunkIndex in range( len(self.input[0]) ):
          # If it is a header, it is saved
          if(type(self.input[0][chunkIndex]) == OVSignalHeader):
             self.signalHeader = self.input[0].pop()


          # If it is a buffer, it gets popped and put in a numpy array at the right
              dimensions
          elif(type(self.input[0][chunkIndex]) == OVSignalBuffer):
             chunk = self.input[0].pop()
             numpyBuffer = numpy.array(chunk).reshape(tuple(self.signalHeader.dimensionSizes)
                 )
             chunk = OVSignalBuffer(chunk.startTime, chunk.endTime, numpyBuffer.tolist())

             # Put the data of the channels corresponding to the sensors in the data array
             for j, data_idx in enumerate([0,1,3,7,6,8,12,10]):
                 self.data[data_idx,:]=numpyBuffer[j]

             # Average the data of each channel
             data_avg=numpy.zeros((16,1))
             for n in range(16):
                 data_avg[n] = numpy.average(self.data[n,:])

             # Create an evoked object and set the montage
             evokedArray = mne.EvokedArray(data_avg, self.info)
             evokedArray.set_montage(self.standard_montage)

             # Clear the window and create a new subplot
             plt.clf()
             ax=plt.subplot(1,1,1)

             # Plot the topomap in the new subplot
             mne.viz.plot_topomap(evokedArray.data[:,0], self.info,axes=ax)

```

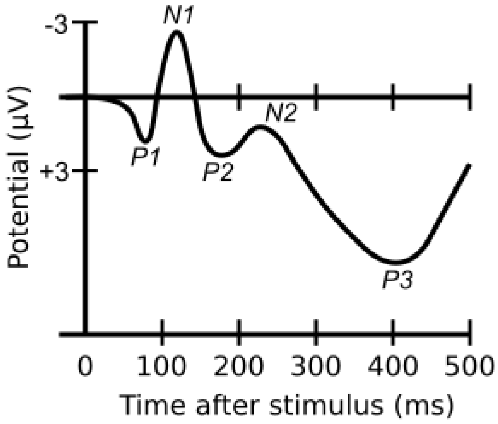
```



            # At the end-of-stream, print a message
            elif(type(self.input[0][chunkIndex]) == OVSignalEnd):
                self.output[0].append(self.input[0].pop())
                print("End of signal")

# Notify OpenVibe that the box instance 'box' is now an instance of MyOVBox.
box = MyOVBox()
```

D
SSVEP research

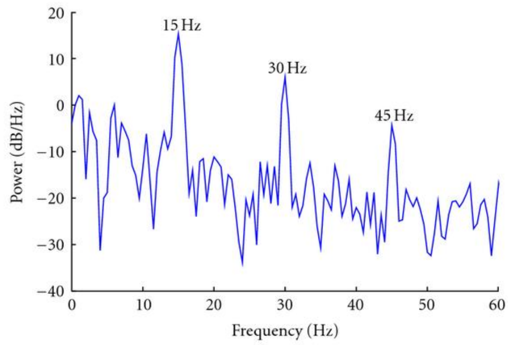
In some papers, visually evoked potential (VEP or SSVEP for steady state version) based BCIs are researched and implemented, such as in [25]. (SS)VEPs are induced by alternating patterns or flickering objects. Similar to motor imagery, SSVEPs caused by certain stimulus frequencies are often used as a paradigm for BCIs. SSVEPs are generally speaking easily identifiable in the power spectra of the EEG channels. Although SSVEPs do not have a direct impact on motor imagery control, they can have a major influence on the attention span and meditation level, which can directly have a positive effect on motor imagery control in motor based BCI. A positive effect on motor imagery control could contribute to the 70% accuracy requirement as stated in Chapter 2. But also, since the (fun) experience of the user is of importance to the interface subgroup and with the future of the BAP project in mind, new game ideas should be thought of, as well as ways to combine certain BCI paradigms instead of just using motor imagery. Therefore it was decided to conduct some research into the possibility of including SSVEP elements in the games by, for instance, making the ball constantly flicker or even creating new games that combine motor imagery tasks and SSVEP elements. The latter is also described in [2] for instance. The paper discusses numerous examples such as the classic Tetris game, where left and right motor imagery (MI) makes the brick move left or right and the user can stare at a flickering object in the corner to rotate the object. The research into the working of SSVEPs and whether they can be identified with a simple setup using the current headset and data acquisition will be shown in the coming sections. The results will be used in order to predict if including/combining SSVEP elements along with MI would be feasible to implement in case of further development of the BAP project . This part of the report should be treated as research and not as part of the design process according to the requirements, hence why it has been placed in the appendix.

D.1. Visually evoked potentials (VEP)

From EEG data, several types of visual responses can be observed; one of these responses are event related potentials (ERP). ERPs show up as electric responses in the EEG data when a specific event occurs in one's brain. Specifically, ERPs are described as "the variations in brain voltage that occur after the commencement of a distinct visual, auditory, or other sensory stimuli, as well as signals activating the motor preparation, motor execution, or covert mental functions" in [26]. P100 (or P1) is a commonly-studied ERP component; it is related to the sensory and perceptually processing of visual stimuli and is observed in the evoked waveform as a positive peak at about 100 ms after the stimulus [27]. Since P100 is evoked due to a visual stimulus, the component is considered a visual evoked potential (VEP). Another VEP is the steady state visually evoked potential (SSVEP). The SSVEP is a potential that is evoked by a visual stimulus that changes/flickers with a frequency of 6 Hz or higher [28], but should preferably be above 10 Hz [29]. The SSVEP shows up in frequency spectra as peaks at the frequencies that equal the rate of flickering and its harmonics [28]. The SSVEP also appears as peaks at the harmonics because of nonlinear characteristics of neuronal populations. With each higher integer-multiple of the harmonics, the peaks become less prominent (see [30] and also figure D.1b.



(a) P100 (P1)



(b) SSVEP of a stimulus with a 15 Hz flickering rate [31]

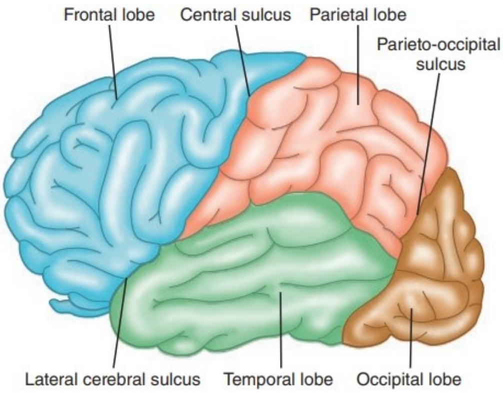
Figure D.1: Two commonly-known visual evoked potentials

D.2. VEP tests

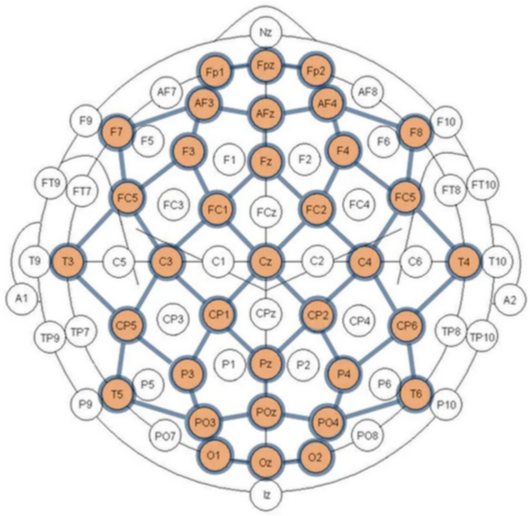
In order to determine if incorporating SSVEPs into the games or combining SSVEPs with motor imagery in new games would be doable in the future of the BAP project, a simple test that induces SSVEPs while measuring EEG data should be conducted. The EEG data from the test should be analyzed to see if the stimulus frequency can indeed be identified using the currently used OpenBCI headset and the data acquisition techniques employed by the measurement group to ensure quality data. After consulting literature, a very simple test setup was found where a subject can simply be subjected to a flickering object on a computer screen. The next sections will discuss the electrode setup and results for the test.

D.2.1. EEG electrodes setup

For this test, the same 10-20 system is used as for the motor imagery tests done by the measurement subgroup. The 10-20 system, as shown in Figure D.2, is especially designed to capture all the different interesting brain regions with specific electrode placements on the scalp. The electrodes names are a reference to certain brain regions, where Figure D.2 also shows these main human brain regions and their names. For example, in the O1 and O2 electrodes, the letter 'O' stands for the occipital region and the number 1 indicates that it is the most left position; an increase in number means that the electrode is more to the right. The 'z' channels such as Pz and Oz are usually used as reference electrodes used to filter out noise. The available 8 electrodes (see measurement subgroup report) for the test were placed on O1, O2, Oz, P3, P4, Pz, F3 and F4, because these are positioned on brain regions that each (to some extent) play a role in visual functions. Table D.1 gives the region and visual related function of each of the mentioned electrodes (see the table caption for more description). For both upcoming tests though, it is expected to see the SSVEP response from the O1, O2 and Oz electrodes so those will be focused on for the results.



(a) Brain regions)



(b) 10-20 EEG electrode system

Figure D.2: Brain regions and electrode placement

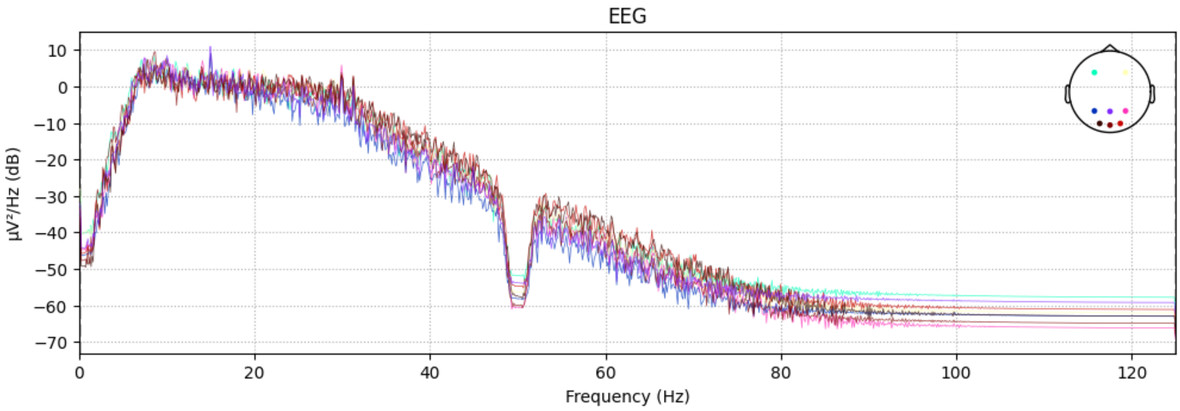
Table D.1: EEG Electrodes and Associated Brain Regions. The occipital cortex, also known a the visual cortex, is widely known to mainly be involved in visual processing, hence its name visual cortex. The parietal cortex is mainly involved in visual spatial processing [32]. The Frontal cortex contains the frontal eye field (FEF) and supplementary eye field (SEF) regions which play a small role in visuals; namely visual fixation (gaze) and eye movements [33][34]. For the tests though, the regions of interest are mainly the electrodes positioned on the occipital cortex. The other electrodes were included since there are 8 electrodes available.

EEG Electrode	Associated Brain Region	Visual related function(s)
O1	(Left) Occipital Lobe	Visual processing and perception
O2	(Right) Occipital Lobe	Visual processing and perception
Oz	(Midline) Occipital Lobe	Visual processing and perception
P3	(Left) Parietal Lobe	Visual spatial processing
P4	(Right) Parietal Lobe	Visual spatial processing
Pz	(Midline) Parietal Lobe	Visual spatial processing
F3	(left) frontal lobe	visual fixation, eye movement
F4	(right) frontal lobe	visual fixation, eye movement

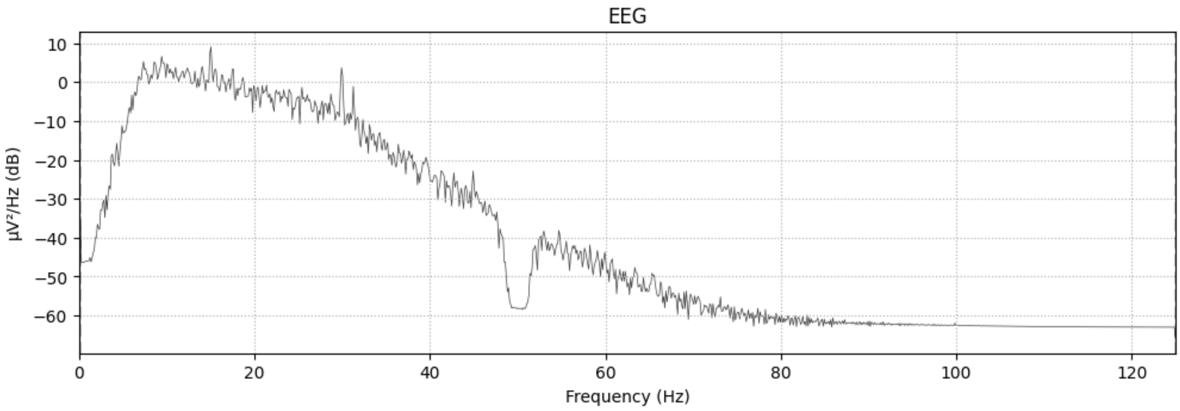
D.2.2. Flicker test

As was also discussed previously, the test revolving around SSVEPs involves showing a flickering object at 15 Hz. The decision for using 15 Hz is due the fact that in literature it was stated that 10 Hz and above will show the best result [29], so the choice fell on a slightly higher number as to not be right on the edge.

Figure D.3 shows the power spectrum from all 8 channels that resulted from the flicker test in Figure D.3a, as well as the induvidual channel Pz. PSD topographic maps indicating brain activity are shown in Section E.3. Peaks in the PSD can clearly be seen around 15 Hz, 30 Hz and even a small one around 45 Hz. It should be noted that there are rather large slopes at the sides of the spectrum, which are caused by a 7-30 Hz bandpass filter employed by the measurement group before sending over the data. Still, the relative peak compared to the neighboring frequencies at 45 Hz can be seen. An odd observation from these results is the fact that only the Pz and P3 channel really show a response to the flickering stimulus, whereas the occipital channels do not show all that much response. Figures E.2 and E.3 in Appendix E.2 show the expected results.



(a) All channels



(b) Pz channel only

Figure D.3: EEG power spectrum of 15 Hz flicker test

An interesting thing to show that was not necessarily the goal to show for this part of the report are the figures in Appendix E.4. These figures are the results of a "faulty" SSVEP trial where during the trial, people were talking directly to the right of the subject. This trial showed a rather high relative increase of power activity from the F3 electrode. This can be seen by comparing the PSD of the bad trial (Figure E.7a) to the previously showed PSD of the other trial (Figure D.3a), in both of which the F3 electrode is represented by the light blue line. Although there is a band-pass filter applied by the measurement subgroup from 7-30 Hz, the relative increase in power recorded from the F3 electrode is still visible. This seems especially so in the higher frequencies 40-120 Hz. The code for some plotting functions used for the data in this chapter can be viewed in appendix B.

D.3. Conclusion

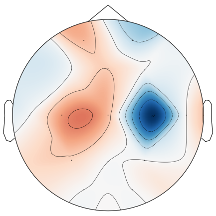
From the test it was expected to see a clear 15 Hz peak in the frequency spectrum as well as a peak at its harmonic frequencies (see Figure D.1b). Figures E.2 and E.3 both show the expected topographic maps; relatively much more activity in the occipital region (O1, O2 and Oz electrodes in the test setup) compared to the other channels. The harmonic frequencies were indeed visible as peaks in the power spectrum density (PSD) of all channels. Strangely though, the peaks were prominent at the Pz and P3 electrodes while only the 30 Hz peak seemed to be visible for the O1, O2 and Oz peaks. Several PSD topographic maps can also be found in Appendix E.3. Figure E.4 shows which brain regions showed the most activity for 15 Hz. As became also apparent from the PSD itself, only the Pz and P3 electrodes seem to indicate a response from the 15 Hz stimulation frequency. Unfortunately, due to time constraints (see timeline in Appendix F), conducting more tests to see why this is the case was not possible. There could be many reasons as to why the results are not as expected, such as individual subject differences, general noise, no referencing, high impedance (meaning bad connection) for the occipital electrodes, test setup and test environment. So now going back to the purpose of this research chapter; whether or not SSVEPs could be easily detected using the current headset, data acquisition setup and filtering methods (7-30 Hz bandpass employed by the measurement subgroup) in order to predict whether incorporating SSVEPs elements in games for future reference of the BAP project. Some SSVEP responses are seen from certain channels but not the channels that they were most expected to be from. As such, the conclusion for now will have to be that using SSVEPs would not be doable since any SSVEP detection algorithm such as spectral analysis (PSDA) and CCA (see [35]) would fail to actually detect the SSVEPs. It should be said however that more tests should be conducted to try and get better SSVEP responses in order to fully determine if using SSVEP elements in the games would be doable in the future.

As briefly mentioned in the results of the SSVEP flicker test, a trial where people were making auditory noise directly to the right of the subject. The increased range of frequencies from the F3 electrode are often referred to as gamma waves which mainly consist of frequencies in the range of 30-100 Hz. These higher frequencies as well as the frontal lobe are both known to be related to attention functionalities within the human brain, which is also mentioned in [36] and [37]. Additionally, since the brain functions in a contra-lateral way, which simply means that the right side of the body is controlled by the left half of the brain and vice versa, it does make sense that the F3 electrode positioned on the left frontal lobe shows heightened activity with respect to gamma waves from the distraction to the right of the subject. The fact that such a distraction can already very much influence the brain waves, shows the importance of attention and focus when it comes to brain wave operated BCIs. This might prove a significant challenge in case the full BCI system is integrated and operational in the future.

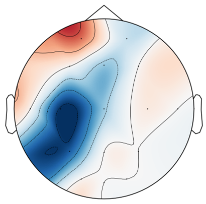
E

Figures

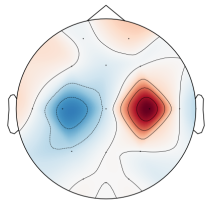
E.1. Topographic mapping samples



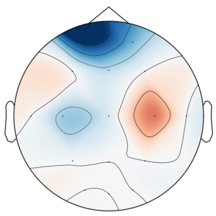
(a) t = 0 - 0.128 s



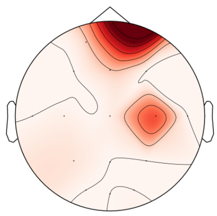
(b) t = 0.128 - 0.256 s



(c) t = 0.256 - 0.384 s



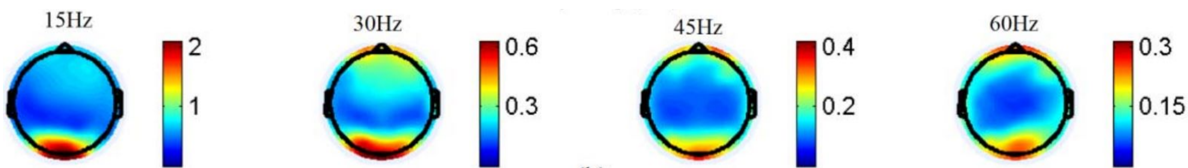
(d) t = 0.384 - 0.512 s



(e) t = 0.512 - 0.640 s

Figure E.1: Left hand clench topographic mapping samples

E.2. SSVEP literature figures



(b)

Figure E.2: Figure of topographic map at harmonic SSVEP frequencies from [38]

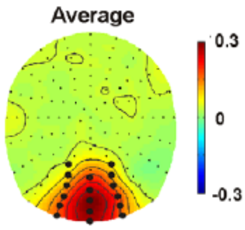


Figure E.3: Figure of topographic map of 15Hz SSVEP frequency from [39]

E.3. Flicker trial

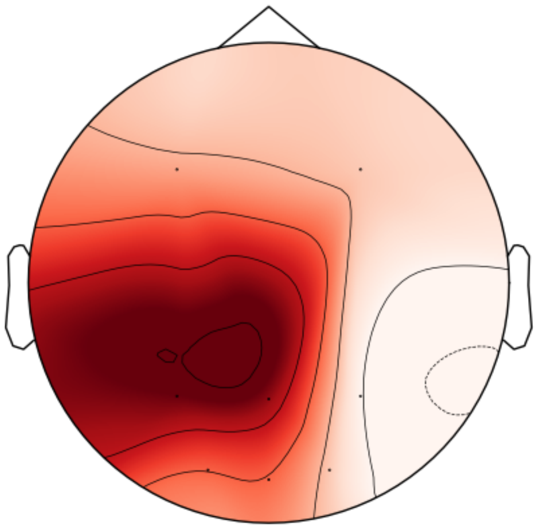


Figure E.4: PSD topographic map at 15Hz

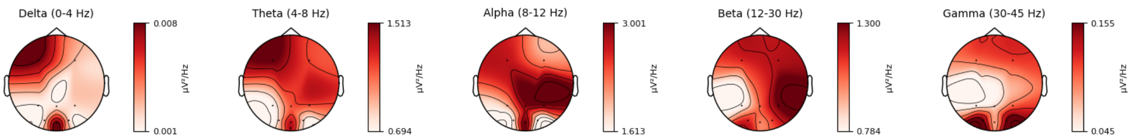


Figure E.5: SSVEP PSD topographic map of several frequency bands

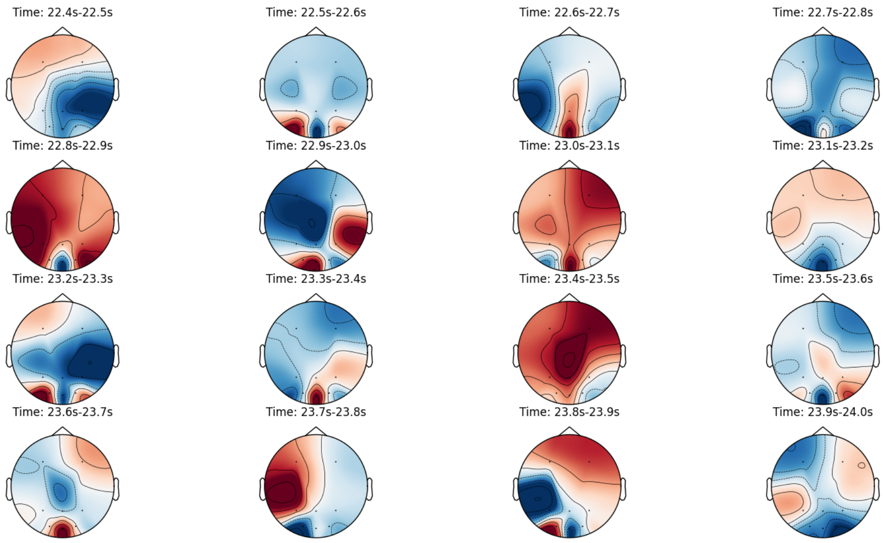
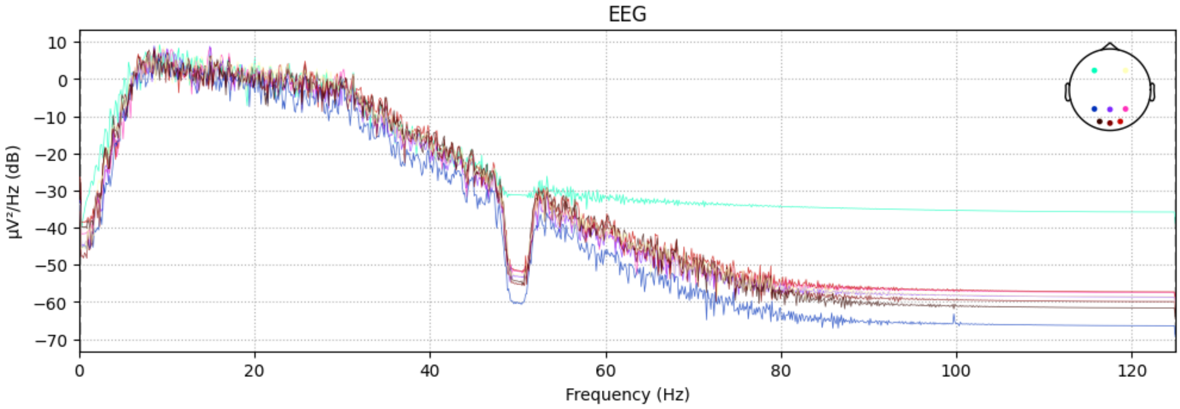
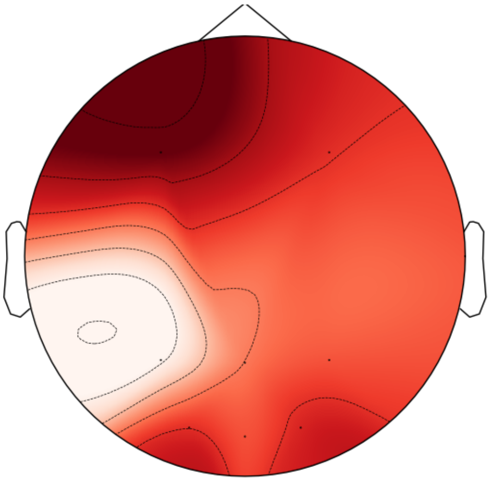


Figure E.6: temporal dynamics with topograhpic maps of a segment of EEG data from the 15Hz SSVEP test

E.4. Bad flicker trial



(a) Bad SSVEP flicker trial Power spectrum



(b) Bad SSVEP flicker trial topomap

Figure E.7: EEG power spectrum of bad 15Hz flicker trial

F
Timetable

Table F.1: Executed tasks for each week

Week	Tasks
1	Literature study and general orientation
2	Start on coding pong for keyboard control
3	Finish coding of pong and complete code breakout for keyboard
4	Complete code animation of topographic mapping for recorded data and start- and end screens
5	Complete code calibration screen, research working with OpenVibe and research concurrency
6	Adjusting code for OpenVibe and research visual feedback
7	Make measurements for VEP tests, implement concurrency and start on thesis
8	Finish thesis, start integrating entire BCI and receive impedances
9	Finish integrating entire BCI and start preparing for final presentation and grand final
10	Final presentation and grand final