Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

University of Zurich[UZH]

Building a Graphical User Interface for a novel Brain-Machine Interface system (GAIA)

Semester Project Report

Yassine Taoudi Benchekroun
ytaoudi@ethz.ch

https://code.ini.uzh.ch/ytaoud/gaia-gui-interface

Institute of Neuroinformatics
Univerisy of Zurich // ETH Zürich

Supervisors:
Matteo Cartiglia, Giacomo Indiveri

March 1, 2024

Acknowledgements

I would like to thank Matteo Cartiglia for his guidance, patience and availability during the realization of this project. I've always felt like working towards a common objective with him, which made all lines of code written significantly more meaningful. I now only wish I could have done more...

Abstract

The *Globally Asynchronous Intelligent Array* Project, or GAIA, aims to design a Multi-Electrode Array (MEA) leveraging event-based computation, in order to overcome the data volume and data transfer bottlenecks in traditional MEAs. The fabrication of the GAIA chip is now completed, and it is currently being post-processed for biological compatibility. To prepare for its testing, the Neuromorphic Cognitive Systems (NCS) group is currently working on the final aspects of the GAIA hardware setup and the software development to control its operation and analyze the collected data. The objective of this semester project was to build a Graphical User Interface in order to enable the testing of the GAIA chip. Specifically, chip biases need to be tuned, and subsequent "events" need to be visualized at a millisecond resolution, with the possibility to record and replay these for analysis. Due to the high amount of data coming out of the Field Programmable Gate Array (FPGA) and the real-time constraints of the visualization, we have naturally opted for C++ and the popular and well documented ImGui framework, which enabled us to meet the technical objectives and build a functional Graphical User Interface (GUI).

Contents

Acknowledgements **i**

Abstract **ii**

1 Introduction **1**

1.1 GAIA: A novel Micro Electrode Array 1

1.2 Objectives and deliverable of the semester project 3

1.3 Software choice for the project: from Python/Tkinter to C++/Imgui 3

1.3.1 Initial attempt with Python and Tkinter 3

1.3.2 Choosing C++ and ImGui 4

1.3.3 FPGA interfacing with OpalKelly Front Panel 4

1.3.4 Visualization with Implot 4

2 Code Walk-trough **6**

2.1 GUI architecture 6

2.2 Event Handler 7

2.2.1 Time stamp of events 7

2.2.2 Address and Polarity of Events 8

2.3 Real Time Event Visualization 8

2.3.1 Algorithmic logic 8

2.3.2 Visualization design 9

2.4 Replay Event Visualization 9

2.4.1 Algorithmic logic 9

2.4.2 Visualization design 11

2.5 Bias control 11

3 Conclusion and Self-Assessment **14**

Bibliography **15**

CHAPTER 1

Introduction

1.1 GAIA: A novel Micro Electrode Array

Neural recording systems are a central component to Brain-Machine Interface (BMI). In these systems, the electrical activity of cells is continually monitored and recorded. The systems are very complex, have a large number of electrodes, and provide very good spatial and temporal resolution.

Current limitations in the data transfer between the recording sites and the data analysis location impose a trade-off between the number of electrodes and the achievable temporal resolution. State-of-the-art systems adopt complex data routing schemes, multiple readout circuits, and expensive computational resources to increase their capability. Although this approach has led to important advances in fundamental neuroscience research, clearly, the scalability of this approach is problematic.

To overcome this difficulty, the Neuromorphic Cognitive Systems (NCS) group designed a new biosensor that uses a novel approach to neural monitoring: the circuits record data only when there is a change in electrical activity, avoiding the collection of large amounts of non-interesting data. Since electrogenic cells are quiet for a large fraction of time, recording data uniquely when there are significant changes in the electrical activity naturally reduces the amount of data collected. The approach proposed aims to have a very large impact in the neuroscience community, as its design would enable the analysis of large-scale neural populations. The project, called *Globally Asynchronous Intelligent Array* or **GAIA**, is a collaboration between the NCS laboratory of The University of Zurich and the Bio Engineering laboratory of ETH Zurich. The GAIA family of biosensors comprises of three different circuits of increasing complexity: GAIA1 has a single electrode, GAIA9 is a 3x3 electrode array, and GAIA4096 comprises a large64x64 electrode array.

The GAIA project is now in a crucial state: the fabrication of the GAIA chip is completed, and it is currently being post-processed for biological compatibility. To prepare for its testing, the NCS group is currently working on the final

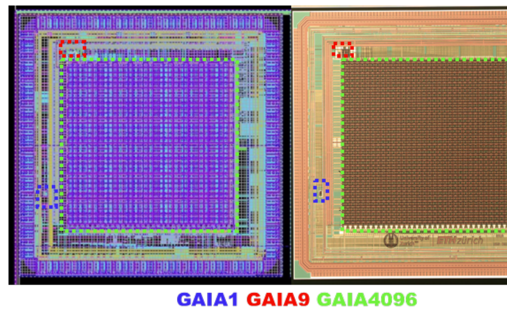

Figure 1.1: Pictures of the GAIA chip. On the left a view of the layout from CADENCE, on the right a view of the fabricated chip. GAIA1 is highlighted in blue, GAIA9 in red and GAIA4096 is green.

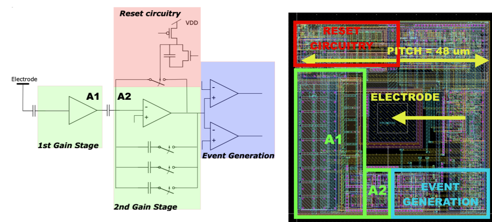

Figure 1.2: Pictures of the GAIA pixel circuits and layout.

aspects of the GAIA hardware setup and the software development to control its operation and analyze the collected data. A Graphical User Interface (GUI) is subsequently needed to enable efficient testing of the GAIA chip. Specifically, chip biases need to be tuned, and subsequent "events" need to be visualized at a millisecond resolution, with the possibility to record and replay these for analysis

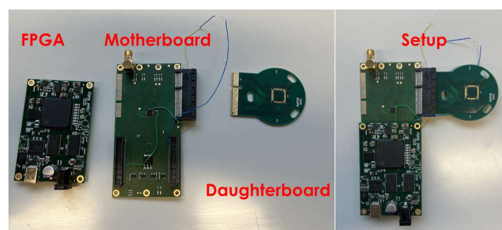

Figure 1.3: Pictures of the GAIA testing setup.

1.2 Objectives and deliverable of the semester project

The GUI needed for this project required a few important features, and needed to satisfy multiple technical requirements. The features needed were:

- **Bias Control Center:** should enable modification of the ADC biases on the chip with sliders, as well as buttons to turn on or off the digital switches.
- **Live Event Visualization:** should enable live event visualization in adapted fashion for GAIA1, GAIA9 and GAIA4096 events. It should also be possible to save events in files in order to replay them.
- **Event Replay:** should enable replay of events saved, with time slowing down feature to facilitate characterization. The replay should also allow for slowing down of event visualization for ease of debugging.
- **ADC Data visualization:** Should allow live visualization of incoming Analog to Digital Converter data to better understand chip function and issues.

The most challenging technical difficulty that needed to be overcome was related to the visualization speed, given the very high amount of data transmitted from the FPGA, simultaneously through GAIA1, GAIA9 and GAIA4096. Events are transmitted from the FPGA at a time resolution $\delta t = 10\mu s$, and visualization should be at an FPS as close to this FPGA resolution as possible.

Testing need to be conducted by connecting the chip through USB with through the OpalKelly Front Panel API [1], integrated to the C++ code.

1.3 Software choice for the project: from Python/Tkinter to C++/Imgui

1.3.1 Initial attempt with Python and Tkinter

As building a GUI was new to me, and as we were not initially certain what the exact technical requirements of the GUI were, I started working with the straightforward Tkinter Python module [2]. Very quickly, we realized that this was not the right tool: live visualization of 5+ simultaneous graphs significantly affected the refresh rate of the GUI to an fps on the order of 0.5 seconds. This was a much lighter data output as the one we expected from the chip and already the fps was much slower than what we needed. It therefore became clear that we needed to change tool and attempt working with C++, which is much more challenging to work with but significantly faster. This also meant that the FPGA interfacing, initially built on Python had to be converted to C++.

1.3.2 Choosing C++ and ImGui

C++ thus became the natural choice after our initial attempt with Python. The popular *Dear ImGui* (ImGui) framework [3] was recommended to us - it so happens that it is also used in a SynSense (an INI spinoff startup) visualization software for their DVS.

Dear ImGui is a C++ GUI library commonly used in game developement. The project is open-source software, licensed under MIT license.

Dear ImGui is a bloat-free graphical user interface library for C++. It outputs optimized vertex buffers that you can render anytime in your 3D-pipeline enabled application. Dear ImGui is designed to enable fast iterations, content creation tools and visualization / debug tools (as opposed to UI for the average end-user). It is designed to favor simplicity and productivity toward this goal. ImGui achieves this by using what is called Immediate Mode GUI paradigm [4], where widgets are created and drawn on each frame, as opposed to the more traditional approach of first creating a widget and adding callbacks to it. Some of the benefits of this paradigm are your UI “lives closer” to data and that it allows for fast prototyping. Dear ImGui is particularly suited to integration in games engine (for tooling), real-time 3D applications, full screen applications, embedded applications, or any applications on consoles platforms where operating system features are non-standard. Dear ImGui comes with lots of widgets like windows, labels, input boxes, progress bars, buttons, sliders, trees, etc. The image below shows some example:

1.3.3 FPGA interfacing with OpalKelly Front Panel

Opal Kelly’s FrontPanel software is designed to provide controllability and observability for FPGA designs. Its design allows users to describe their own control panels using industry-standard XML descriptions of components such as LEDs, hex displays, push buttons, toggle buttons, triggers, and so on. The components then connect to endpoints within the user’s FPGA design. Once connected, the interface details are transparent. FrontPanel handles all interaction between the virtual controls and the FPGA internals.

Matteo built the C++ GAIA interfacing API through Opal Kelly Front Panel - which was subsequently used for receiving the events and controlling the biases.

1.3.4 Visualization with Implot

Plots built on the GUI were built using ImPlot [5], which is an immediate mode, GPU accelerated plotting library for Dear ImGui. ImPlot is well suited for visualizing program data in real-time or creating interactive plots, and requires minimal code to integrate. It was thus perfectly suited to our needs. Just like

Figure 1.4: Imgui Example Widgets.

ImGui, it does not burden the end user with GUI state management, avoids STL containers and C++ headers, and has no external dependencies except for ImGui itself.

Chapter 2

Code Walk-trough

The ImGui pipeline was built following the example application for GLFW + OpenGL 3 (GLFW is a cross-platform general purpose library for handling windows, inputs, OpenGL/Vulkan/Metal graphics context creation, etc.). From this model, the ImGui Demo and documentation, as well as the ImPlot library, I selected the most adapted widgets for our visualization needs and first tested them with basic "dummy" functionalities. In this chapter, I will present the overall final structure of the functioning GUI, as well as the specific structure and logic of the functionalities developed.

2.1 GUI architecture

Below is the pseudo-code of the overall structure of the program:

Algorithm 1 Overall Code Structure

```
→ Create GLFW Window
→ Set Dear Imgui Context
while Window open do
  if Chip Connected then
    → Render Live visualization panel
    → Render Bias Control
    → Listen for events
    if Events received then
      → Process events
      → Show events
    end if
  else
    → Render Event Replay Panel
  end if
end while
```

ImGUI runs by default at 60fps - while we could get it to run faster, we

decided to keep this default FPS for simplicity and as it was enough for our live visualization needs. We print out this FPS in the GUI, as it is calculated dynamically with an ImGUI function by computing the elapsed time between the current frame and the previous one. As such, we are able to control that the GUI keeps on functioning at 60 FPS during execution, and does not, for example, lag when high number of events are sent. The 60 FPS also means that we have a maximum precision of $\delta t = 1/60s$ - there is thus no point in attempting to visualize the events at the $\delta t = 10\mu s$ resolution of the FPGA.

2.2 Event Handler

The FPGA sends events, through the OpalKelly Front Panel API. All events transmitted are composed of a time stamp, and an address/polarity (except GAIA 9 events which do not have polarity with the adress of events).

2.2.1 Time stamp of events

Time stamps are 16 bit values with resolution of 10 μs. When both ts and address are equal to 0xFFFF (65535), it means that $65535 * 10us$ has passed. This is called an "*overflow event*", and indicates a reset of the ts to 0. We should also notes that, typically, "bursts" of events arrive together at the same time. From this, it became clear the time stamps must be processed in an efficient and clear way in order to convert them into clock time, which will simplify visualization and debugging. The algorithm for processing the time stamps is the following:

Algorithm 2 Time stamp processing

$\rightarrow$ *Set OverflowEventCount to* 0
if *Events Received* **then**
  for *eventinEvents* **do**
    if *Event is overflow* **then**
      $\rightarrow OverflowEventCount += 1$
    else
      $\rightarrow$ *convert ts to seconds*
      $\rightarrow$ *add OverflowEventCount events equivalent in seconds*
      $\rightarrow$ *Reset OverflowEventCount to* 0
    end for
  end if
end if
if *Chip Reset* **then**
  $\rightarrow$ *Reset ts calculation* 0
end if

The time stamps are stored in a 2D C++ vector *gaia_events* (*loaded_gaia_events* for the replay visualization) along with the corresponding addresses.

2.2.2 Address and Polarity of Events

The FPGA returns, along with the corresponding time stamp, the address and polarity of events, regardless of their origin (GAIA1, GAIA9 and GAIA4096). Sorting must then be done in order to visualize on the adequate visualization panel, as well as the adequate polarity.

- **GAIA1:** 0xC000 represent ON events, and 0xC001 represent OFF events.
- **GAIA9:** In the current GAIA chip, no Polarity transmitted with the event data. Only the address is transmitted, with the following logic: 0X4000 - 0X4007 for pixel 1 to 8, and 0X400F for pixel 9. Numbering is done from top left to bottom right.
- **GAIA4096:** Transmitted as 13 bit value, with the following structure: ADD[0:12] = Polarity (1 bit)- X address (6bits) -Y address (6bits). Address (0,0), like in GAIA9 is on the top, left.

In the live visualization, the addresses and polarity are sorted right after the events are received (together with the corresponding processed time stamp). They're subsequently used to populate the GAIA1, GAIA9 and GAIA4096 vectors, which are used for visualization.

2.3 Real Time Event Visualization

2.3.1 Algorithmic logic

The Pseudo code below outlines the logic used for the live visualization.

Algorithm 3 Real Time Visualization Logic

```
→ Initialize event vectors for visualization
→ Start ImPlot context
while Open do
  for i = Last_Event_shown → i = Last_Event_received do
    → Classify Address and Polarity of events
    → Populate relevant event vetors
  end for
  → Populate ImPlot Widgets
end while
```

For the live visualization, events are shown as soon as they are received. Code execution time is not used for the visualization logic, but it is nevertheless displayed on the GUI for information.

A critical element to note for the live visualization is that we only need to show one spike/event per frame, even if we receive more events in a span time shorter than our FPS. This makes sense as we cannot possibly visualize more than one pixel event per frame. We thus do not attempt to show every spike, which could potentially lead to slowing down the refresh rate.

2.3.2 Visualization design

For ease of visualization, I have built a single visualization panel that shows simultaneously the visualization of GAIA1, GAIA9 and GAIA4096. Inspired by Dynamic Vision Sensor (DVS) softwares such as jAER [6], I designed, for each, a "matrix" that would light up in red or green (depending on the polarity) in the corresponding address. For GAIA1, this was done with two independent single unit matrices - one for the ON and the other one for the OFF events. The reason I used two is simply for aesthetic purposes rather than for a specific functionality. For GAIA9 and GAIA4096, single matrices of respective size 3x3 and 64x64 were positioned spiking in the correct address (e.g. event ON at pixel 0,0 would fire green on the top left). This is done using the *PlotHeatmap* Implot function. This simple visualization choice allows to minimize the computing time as events can be popped off the visualization once they have been shown for one frame - which would not be the case with rolling graphs for example. Furthermore, they are perfect for visualizing where on the chip the event come from, especially in 4096.

For GAIA1 however, another visualization widget was added in order to further help debug and characterize the chip behaviour: a rolling graph (using the "PlotLine" ImPlot function). This is computationally feasible as the number of events GAIA1 is not very data-savvy, and only one rolling graph suffices (Which would not be the case for GAIA9 and GAIA4096). The GUI user can also change the "history window" of the rolling graph visualization, i.e. until when in the past do we want to keep showing the events (e.g. 30 seconds, 45 seconds etc...).

2.4 Replay Event Visualization

2.4.1 Algorithmic logic

An event saving functionality was added to the live event visualization, in order to allow the user to save the last 15 seconds, 30 seconds, or all the events since the beginning of run time. Events get saved as a text file, with each line holding

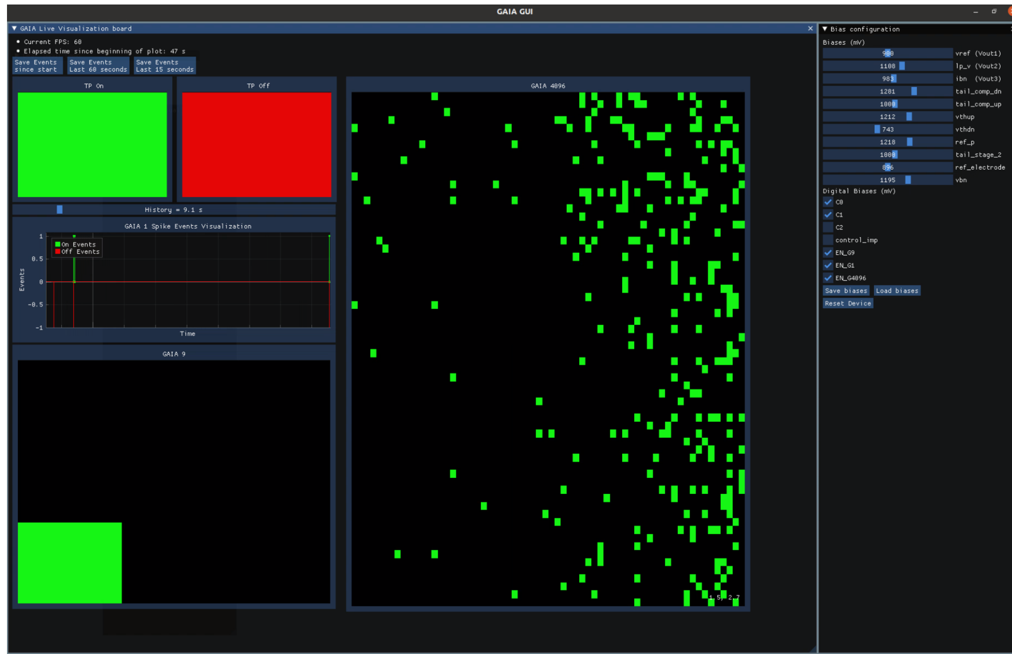

Figure 2.1: GAIA Live Visualization panel. On the top left, we can see the two pixels for ON and OFF GAIA1 pixel. Below is the rolling graph visualization for GAIA1, along with it's time history selection. Below is the GAIA9 Visualization Matrix. On the right is the GAIA4096 visualization matrix. On the far right, we can see the Bias Control center, which is typically opened alongside the live visualization. Time since beginning of runtime (or since last reset) is printed, as well as the visualization FPS. We observe a burst of events on GAIA1, GAIA9 and GAIA4096. We see both ON and OFF events on the test pixel because both events happened at least once during the 1/60 s window (since last frame) - thus both appear at the same time on the graph visualization and the matrix visualization.

an address/polarity and corresponding time stamp. For simplicity, instead of saving the FPGA time stamps and doing the conversion again, we directly save the converted time stamp. This allows to save computation during the replay.

The critical element allowing the live visualization is that a timer is started with the onset of the program (with the C++ Chrono library [7]). We then proceed to loop over the event file chosen by the user - as soon as our run time goes past some of the events ts, it means we are lagging in terms of events visualization, and subsequently proceed to show all the events for which time is past the current execution time.

The Pseudo code below presents this algorithmic logic:

Algorithm 4 Replay Event Visualization Logic

$\rightarrow$ *Initialize event vectors for visualization*
$\rightarrow$ *Start ImPlot context*
$\rightarrow$ *Start real running time*
while *Open* **do**
  for $i = Last_Event_shown \rightarrow i = Last_File_Event$ **do**
    if $ts_event < current\ time$ **then**
      $\rightarrow$ *Classify Address and Polarity of events*
      $\rightarrow$ *Populate relevant event vetors*
      $\rightarrow$ *Populate ImPlot Widgets*
    end if
  end for
end while

2.4.2 Visualization design

The visualization design of the event replay is the same as the live visualization panel. Only addition is the ability to slow replay time for debugging ease.

2.5 Bias control

The bias control was built in order to allow modifying biases and turn on and off switches. This is extremely useful observe in real time the impact of each bias on the behaviour of the chip. This was done in a relatively straightforward way provided the OpalKelly Front Panel API was set up correctly. It then was only a matter of implementing sliders and buttons with ImGUI which value were dynamically updated through the GUI. Functionalities to save and subsequently load sets of Biases were also added, simply by saving the current value of all biases/switches into a text file, and loading new values to these variables (and updating the values on the chip). Testing of the the Bias Control was done with

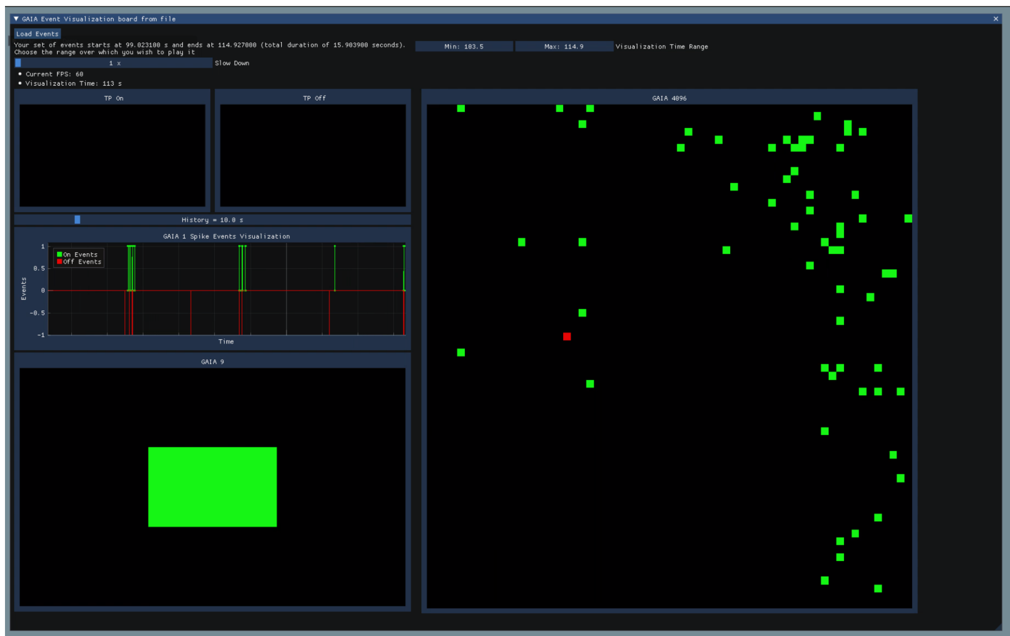

Figure 2.2: GAIA Replay Visualization panel - similar to the live visualization. On the top, the controls for slowing down time (up to 100x) and reducing the replay window. We also see the dynamically computed FPS and time stamps at which we're visualizing (as this event file records event that occurred seconds previous to 114 seconds of run time - hence the 99 to 114 seconds.

by measuring voltage of the different bias points with a multi meter, as well as observing the LEDs switching ON and OFF for the digital biases. The design of the bias control can be seen on the right hand side of figure 2.2. The analog biases are tuned with sliders, and the digital ones with switch. The bias control center also allows to load and save sets of biases. It is also where the chip can be reset.

Chapter 3

Conclusion and Self-Assessment

The final deliverable met the initial expectations and allows Matteo and any future collaborators to integrate new functionalities and expand on the current ones. By the end of the project, we were convinced that the choice of using ImGUI/C++ was the right one, as it showed promising performance on all the expected technical challenges. The GUI is delivered in a state ready to begin characterization of the chip - which was the initial goal. However, we should note some issues that have been encountered during the execution of this project, and some drawbacks of the final deliverable. Mainly, this project took much longer than originally planned because of the initial time spent with Python, as well as the overlapping class projects that I had to finalize during the same time window - the latter significantly slowed my progress and did not allow me to develop everything I wanted to, and affected the quality of some of the features. Not being perfectly fluent in C++, it also took me a lot of time to get to up to speed, and I realize that some lines of code written can be mediocre or even poor practice (MakeFile issues, dependencies, namespace etc, memory management...). However my biggest regret with this project was my inability to work on anything around the GUI - when originally discussing things with Matteo, my goal was to help him with the GUI (which I did) and at the same time learn about as many other aspects of the project as possible, mainly with the electronics components. The pressure of the other class projects consumed too much time and energy off me and left me unable to devote time to study and learn about the other aspects of Matteo's work on GAIA.

Bibliography

[1] "Opalkelly front panel," https://opalkelly.com/products/frontpanel/, accessed: 2022-07-27.

[2] "Tkinter," https://docs.python.org/3/library/tkinter.html, accessed: 2022-07-27.

[3] "Imgui," https://github.com/ocornut/imgui, accessed: 2022-07-27.

[4] "Immediate-mode graphical user interfaces," https://caseymuratori.com/blog_0001, accessed: 2022-07-27.

[5] "Implot," https://github.com/epezent/implot, accessed: 2022-07-27.

[6] "Dvs java tools for address-event representation (aer) neuromorphic processing." https://github.com/SensorsINI/jaer, accessed: 2022-07-27.

[7] "C++ chrono library," https://en.cppreference.com/w/cpp/chrono, accessed: 2022-07-27.