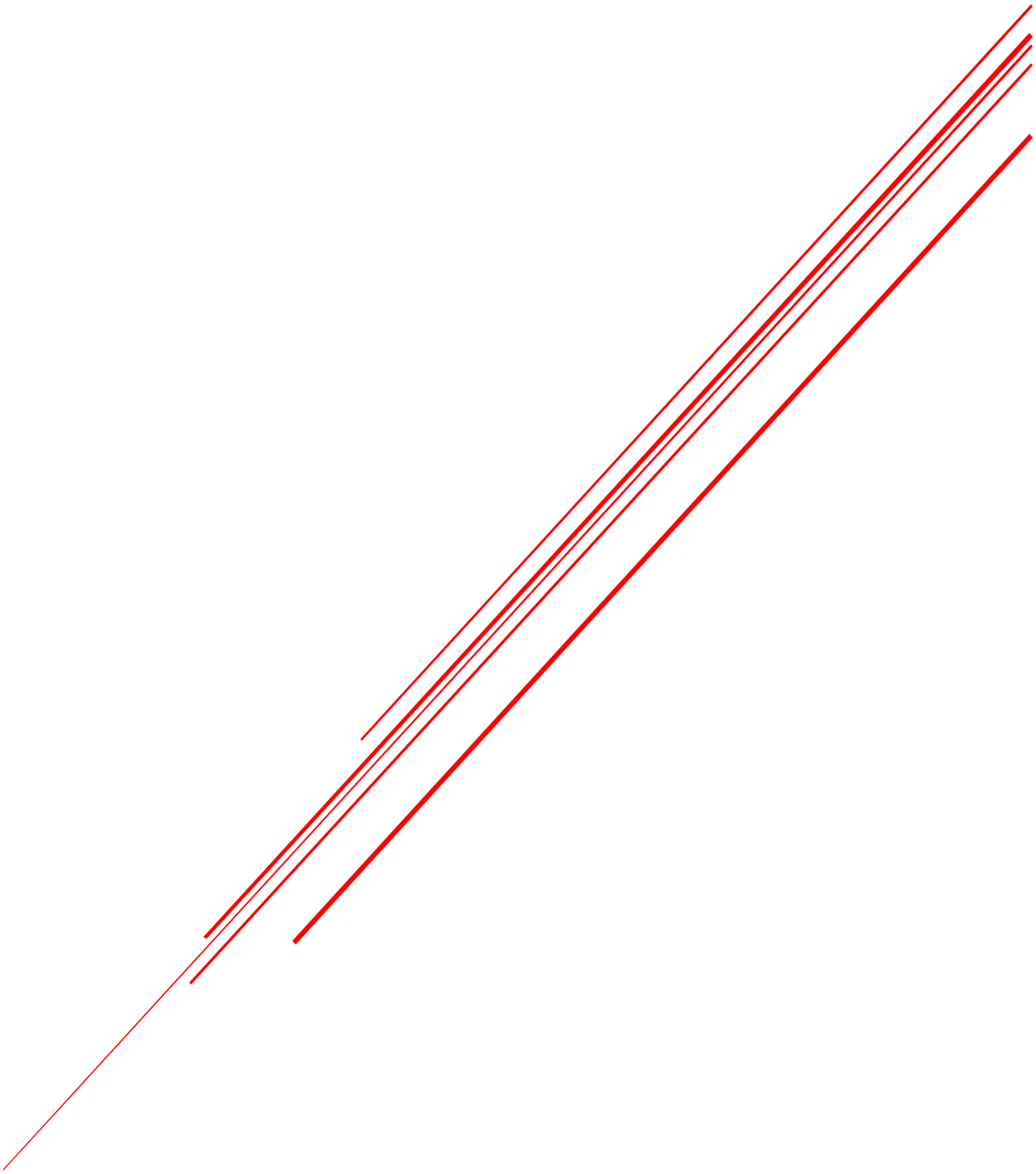
MOTION-CONTROLLED LED MATRIX

New Devices Lab project of A10



Nikita Kuprins & David Scholten

9 January 2025

Contents

1 Introduction

The Wii, PlayStation, Xbox, Nintendo Switch, Game Boy and the Nintendo DS. All of them are very famous consoles and have many similarities. One of them is that all of them use controllers and/or buttons to play games on a screen, but none of the consoles are only based on motion. Some of them do have motion functions, like the Wii and the Nintendo Switch, but still require buttons in menus or for certain actions.

For the course *New Devices Lab* in the Computer Science program at Radboud University, the goal is to design and implement a new device. The device described in this report, is - contrary to the consoles mentioned before - a console completely based on motion and can be used by just tilting the screen.

The report will describe and explain the approach and the device, the reasons behind choices made and include a reflection on these choices. It should give a clear understanding of how the device works, how it is created and how it is supposed to be used.

In chapter 2, the goal of the project and the approach towards this goal will be discussed. The approach will be split up into three parts, namely the feasibility study, the minimum viable product and the final product.

Next up, chapter 3 will be about the electrical and mechanical design of the device, where first paragraph will show the wiring of the device, paragraph 3.2 will give the specifics about the mechanical design of the device and paragraph 3.3 will discuss the combination of the two.

Subsequently, in chapter 4, the software design will be explained. This will be split into the motion controller in 4.1 and the console itself in 4.2. Both paragraphs will contain UML diagrams and finally a flow chart will be provided. Then in 4.3 some projects will be credited.

After the device has been explained in chapter 2, 3 and 4, chapter 5 will contain a report on the process, where the choices made on the technical implementation, the learning and implementation process and the sustainability will all be discussed and improvements on these aspects will be mentioned.

Finally, in chapter 6, a step-by-step explanation for users on how to use the product in every scenario will be provided.

2 Goal and approach

In this chapter, the process towards the final project will be discussed. The first paragraph will be a short explanation of the goal and then the approach to this goal will be described.

2.1 Goal

The goal of the project is mainly for users to be able to have fun with the games that are created in the project, but also to work on their hand-eye coordination. The final product allows users to be able to train their coordination on either or both of their hands and do this while trying to get their high score in games like *Snake* and *Pong*.

When the controller is attached to the screen, the movement of the blocks in *Cubes*, the snake in *Snake* and the pong bat in *Pong* follow gravity. When made safer for children, this could possibly help them understand gravity and movement, but is not the main goal of the product.

2.2 Approach

Firstly, paragraph 2.2.1 will be about the feasibility study. After that, the making of the minimal viable project will be described in paragraph 2.2.2. Finally, the steps towards the final version will be discussed in paragraph 2.2.3.

2.2.1 Feasibility study

In this paragraph, the different components and concepts that were tested before the making of the minimum viable product are discussed. These are the BNO055, 64x32 LED matrix and socket communication, respectively. In the subparagraphs, the boards that were used will also be mentioned.

2.2.1.1 BNO055

The initial idea was to create a game on a monitor or laptop and make it motion controlled with a hand gesture sensor which communicates through a Wi-Fi connection. The feasibility study started off with testing the BNO055 accelerometer and gyroscope functions to test its suitability for the hand gesture sensor. For this is was connected to the LOLIN D1 mini PRO.

With code that returned directions for certain angles, for example “LEFT” if the rotation over the x-axis was -15 or less degrees, the sensor turned out to work well and was not very complicated. It took just over one session to get this working, so the decision was made to continue with this sensor, but try to make the project more complicated in another way.

2.2.1.2 64x32 LED Matrix

It was then decided to make the game on a programmable 64x32 LED matrix instead of a monitor or laptop screen. To get the screen working, the LOLIN Wemos D1 mini, the Arduino MEGA ADK and the Raspberry Pi Pico. After having the same issue with all of these boards, where several parts of the screen were not lighting up correctly (see figure 1) and checking the wires and the code a lot of times, the realization came that the issue might not be in the boards, the wires and/or the code, but in the matrix itself.

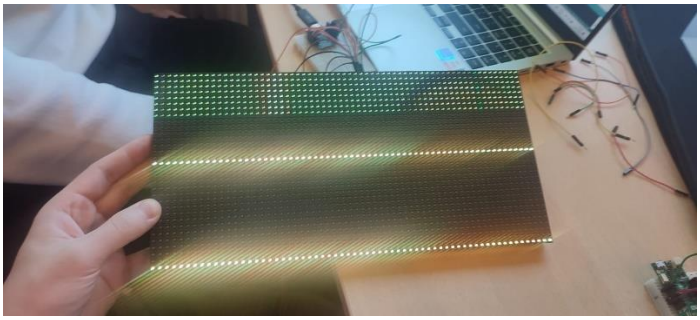


Figure 1: Matrix working incorrectly

After four practical sessions of struggling with the first matrix, a different matrix was tested and worked immediately. This cost more time than necessary, but in the end there was a working matrix with both the Arduino MEGA ADK – which was initially intended to be in the project, but did not have Wi-Fi - and the Raspberry Pi Pico, which also did not have Wi-Fi, but this was easily fixable by using the Raspberry Pi Pico W. As programming this matrix seemed complicated, but not too complicated, the decision was made to continue with this project, combining it with the BNO055.

2.2.1.3 Socket communication

Our project required a persistent network communication between the matrix and the motion controller so between Raspberry Pi Pico W and LOLIN D1 mini PRO. In one of the first practical sessions the HTTP protocol was used and explained but it did not suit our project, since HTTP is mainly used for hypermedia documents and has request/response functionality, which we do not need. Besides, it is on the application layer in the Internet protocol suite, but for our project, we did not need any overhead, so the transport layer was enough. The initial choice was TCP but it turned out to be a mistake and it will be explained why in the final project part, since we did realize it for our feasibility study.

When trying to implement the TCP communication we had issues and it took some effort to get the socket communication to work. Every time our connection was open, it was not closed after the termination of the program even though we used the close function. This would result in the board not being able to find and connect to the NDL_24G network anymore or accept the new connection, unless a ‘flash_nuke.uf2’ file – which completely resets the entire board to its initial settings – was put onto the board and all

the files and libraries would be put back after this. This took a few minutes every time and when it had to be done multiple times per session, this was obviously quite annoying. The teacher assistant suggested that we try to call the garbage collector manually, since CiruitPython still has some issues with that. It turned out to almost fix the issue but we still had the same problem sometimes. After reading the documentation for the *socketpool* library we could not find anything useful and for some reason decided to look into the Python documentation for the *socket* library. For our surprise we found there an important part of the documentation highlighted as "**Note**", which was not in the documentation of the CircuitPython, and this part was: "*close() releases the resource associated with a connection but does not necessarily close the connection immediately.*". Hence, we tried to add a small delay after calling the close function, and the issue was gone completely.

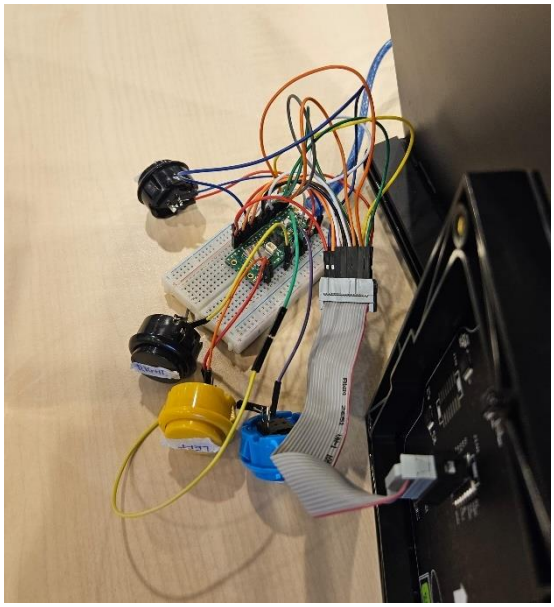


Figure 2: Board with buttons

2.2.2 Minimum viable product (MVP)

This paragraph provides insights into the making and coding of the minimum viable product (which will be abbreviated to MVP from now on). Subparagraph 2.2.2.1 will be about the communication, subparagraph 2.2.2.2 about the games and finally, 2.2.2.3 will contain information on the combination of the two.

2.2.2.1 Communication

An essential part of the project is communication between multiple components. For this, the final version will use a Wi-Fi connection. However, when trying to test the code that was written for the games, it quickly became clear that having to make a connection with the BNO055 every time would take some time and sometimes the issue discussed in subparagraph 2.1.3 could occur. To make sure that the games could be tested without having to make a connection, the games were made to use buttons first (see figure 2).

2.2.2.2 Games

For the MVP, we chose to recreate Snake and Pong games. The choice was based on the fact that we already had some experience in making them.

2.2.3 Final product

After making the MVP work, the next step was to make the games work with the BNO055 sensor and fix socket communication issues.

As it was already mentioned, TCP turned out to be a bad choice. The persistent communication was working without any errors, but sometimes there were periods

where we did not receive any data from the motion controller for a second or so. For the user experience it is really bad, since a user could be at a critical point in a game and not be able to move for a second resulting in game over. It was not easy to find the core of the problem, especially since we always used TCP in previous projects, but in the end, we realized that due to reliability, order and error-check, network communication had some unnecessary delays. Hence, it was decided to switch to the UDP protocol, which was even easier to implement, so we got rid of some code making it easier to understand the network part of our project.

As the boards we used were not supposed to be connected to laptops in the final product, one of the group members brought a power bank to provide electricity to them and a frame that could be used by the user to hold the screen and connect the power bank to was constructed. The BNO055 was stuck on the power bank. To make it more interesting, the idea came up to be able to connect the controller to the screen, but also to be able to use it separately. This meant glue or tape was not a great option. Luckily, there were stickers with Velcro (Dutch: klittenband), which made it easy to connect and disconnect the controller to the screen. As the controller would be upside down when connected to the screen, the code for the sensor needed some changes, so it would still give the values needed for the games. After this was done, the software for the final product had one final refactoring and was finished.

3 Electrical and mechanical design

This chapter will be about the hardware of the project, split into three paragraphs. Paragraph 3.1 will be about the electrical design, including wiring diagrams and parts that were used. After that, the mechanical design of the device will be discussed in paragraph 3.2 and finally, in paragraph 3.3, the combination of the two will be shown with a diagram.

3.1 Electrical design

The electrical part of the device exists of two smaller devices, namely the 64 x 32 LED Matrix connected to the Raspberry Pi Pico W and the BNO055 sensor connected to the LOLIN D1 mini PRO 2.0.0. Both will have their own subparagraph in this paragraph.

3.1.1 LED Matrix with Raspberry Pi Pico W

To provide enough processing power and Wi-Fi, it was chosen to use the Raspberry Pi Pico W for the LED matrix. In figure 3, it is shown how the wiring between the microcontroller and the LED matrix HUB75 IN-port is done.

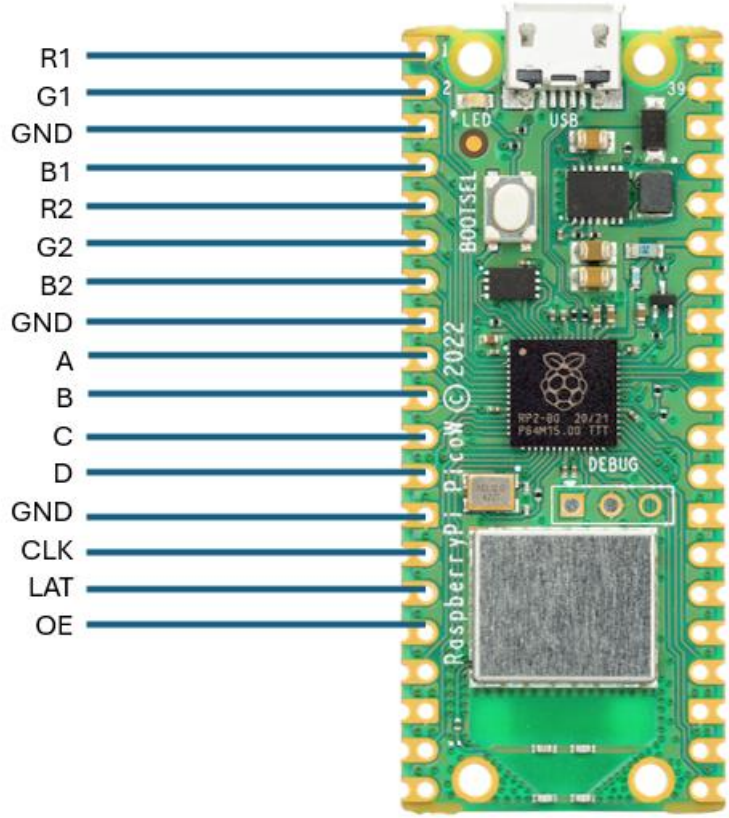


Figure 3: Wiring LED Matrix (Source picture of Raspberry Pi Pico W: Raspberry Pi Ltd, 2024)

This way was chosen, because all the GND pins in the HUB75 port are now connected to a GND pin (pins 3, 8 and 13) of the Raspberry Pi Pico W, and all the other pins in the HUB75 have a connection from a GPIO pin.

3.1.2 BNO055 and LOLIN D1 mini PRO 2.0.0

As the microcontroller connected to the sensor would not need a lot of processing power and memory to be able to extend the project if wanted, the choice was made to use a PRO version of an ESP8266 board, namely the LOLIN D1 mini PRO 2.0.0. This was then connected to the BNO055 sensor on a breadboard, following the wiring scheme in figure 4.

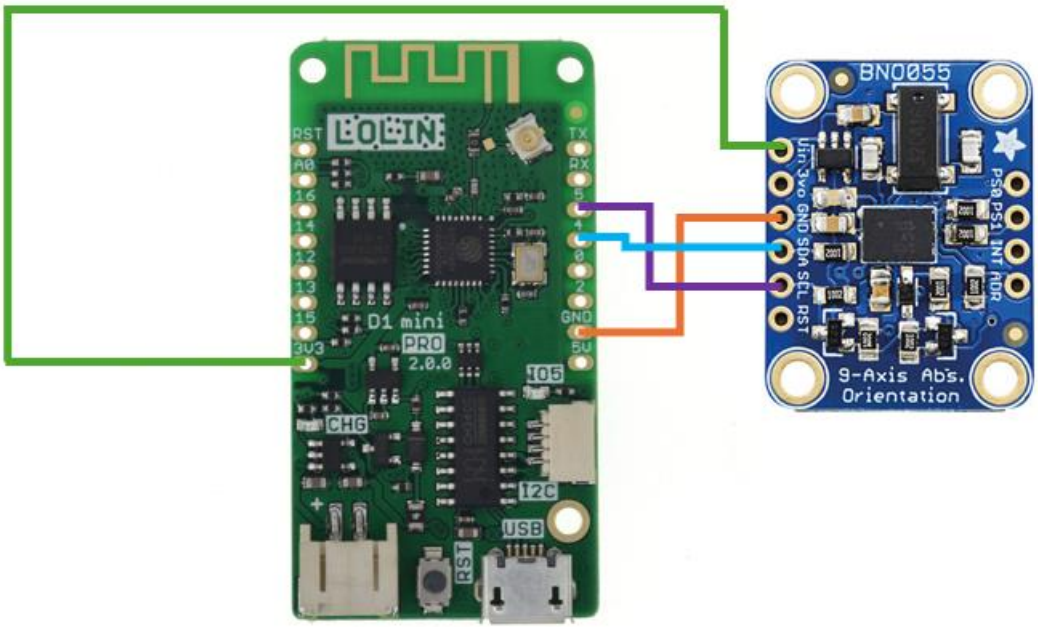


Figure 4: BNO055 wiring. (Source LOLIN Board: WEMOS, n.d.; source BNO055: reichelt elektronik GmbH, n.d.)

This wiring scheme makes sure the BNO055 has a V_{in}, which comes from the 3V3 pin of the D1 mini PRO. It also connects the GND, SDA and SCL pins from the sensor and the board, making sure these work in harmony with each other. As the two boards use Wi-Fi to work together, no wired connection is needed between the boards.

3.2 Mechanical design

As the user also had to be able to control the matrix by moving the matrix itself, a frame was made to hold the matrix and also to connect the BNO055 and the power bank to it. Velcro stickers were then added to both the power bank and the frame (see figure 5), so that the power bank can be attached to the frame, but can also be unattached. To make sure that no damage would be done and the matrix could be reused, some tape was used to reinforce the connection

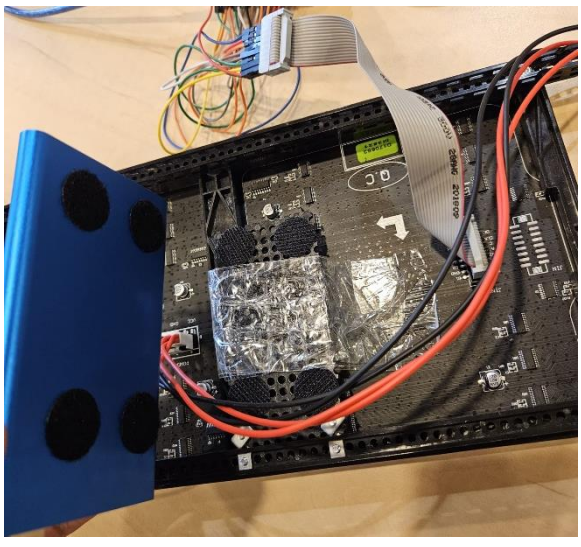


Figure 5: Velcro stickers

between the frame and the ‘powerbank-holder’ which is screwed onto the frame. In a next version, this would have to be fixed with something that looks better.

3.3 Total design

In figure 6, a schematic overview of the full electrical and mechanical design is given. This shows which components are connected and in what way. The double-sided arrows represent the wiring from figure 3 and 4. The dashed arrow represents the Velcro connection.

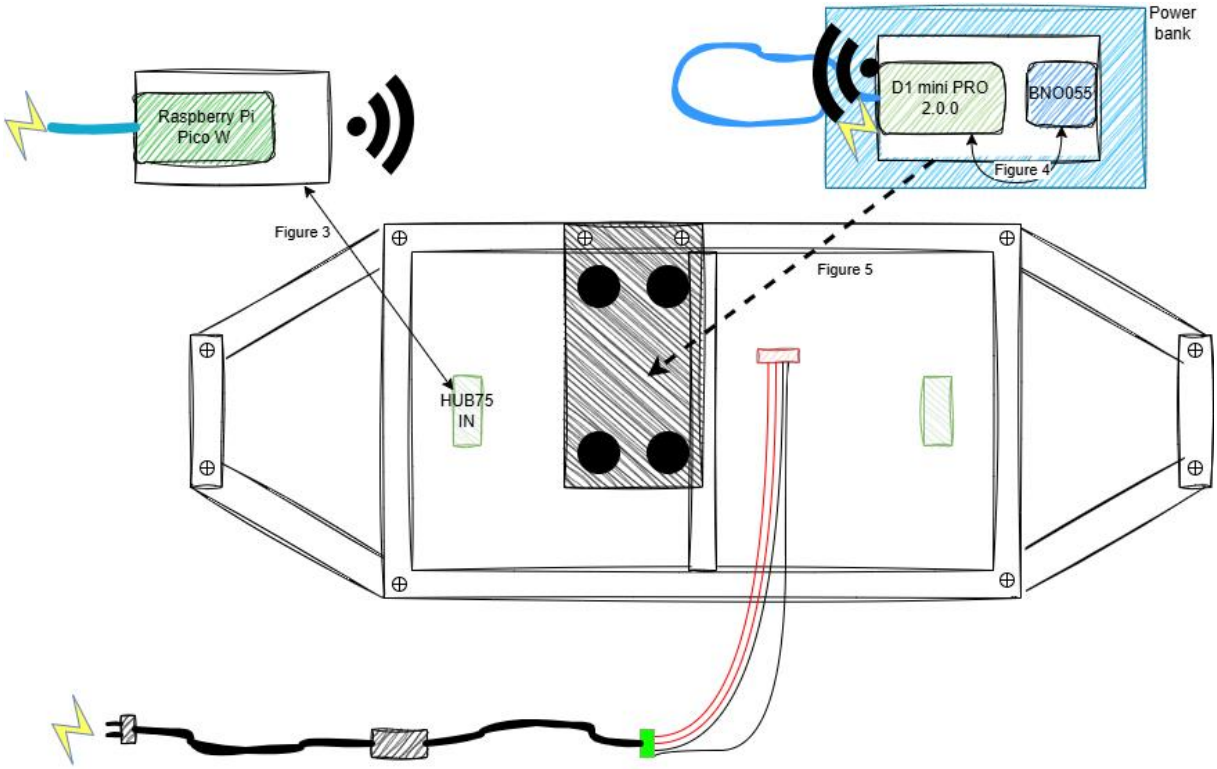


Figure 6: Diagram of device

In figure 7, it is shown what the device looks like in real life. Here it is possible to see the frame and the wiring.

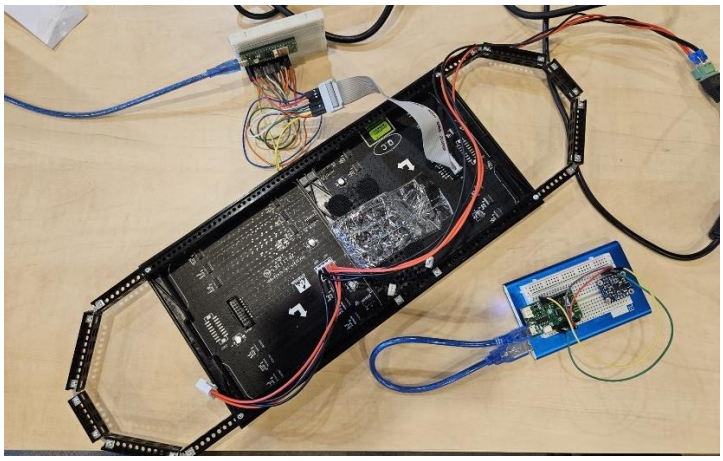


Figure 7: Device in real life

4 Software design

For the software design choice, we use flow charts and UML diagrams for better explanation. We start with the motion controller design in paragraph 4.1 and then explain with the matrix design, which is in paragraph 4.2. Finally, we will credit some other projects which were used in some way in the project.

4.1 Motion controller

Our code for the motion controller is quite simple, so there was no need to make any OOP design choices and we wrote almost everything in one file using setup, loop, and auxiliary functions. The language choice was C++, as this was what we used at the start of the course. To understand how to use the Adafruit BNO055 library we looked at the example code provided by the library. The code in the example obviously did not fully suit us but it was helpful enough to start writing.

It was mentioned that we almost wrote everything in one file. The only part that was not actually written in the main file was the Wi-Fi SSID and password, as this is secret information. One of the course requirements was to use GitLab, and when pushing the code to not leak the secret information we would have to make SSID and password fields empty each time we push, which would have been annoying. Ignoring the leak of the secret information would be fine for the scope of this course, as the repository is not public, but this would not be a good practice in general. Hence, as soon as we realized that we should not push it, we decided to create a configuration file for our secrets and *gitignore* it.

The code starts from the setup function where we connect to Wi-Fi and initialize BNO055. Then, in the infinite loop, we get an event from the BNO055, adjust the coordinates if upside down(when we manually attach it to the matrix), create a UDP packet and send it to the IP and port of the server. Before we switched to UDP, in the TCP we had an additional check for the open connection in the loop. If the connection was closed, then we had to reconnect. However, since UDP is connectionless, we do not have this check and just send our data. This simplified the code a little. The iteration ends with some delay before the next iteration. After a few tests, we decided to stick to a 100-millisecond delay. Below in figure 8 is a flow chart, where we represent the whole process.

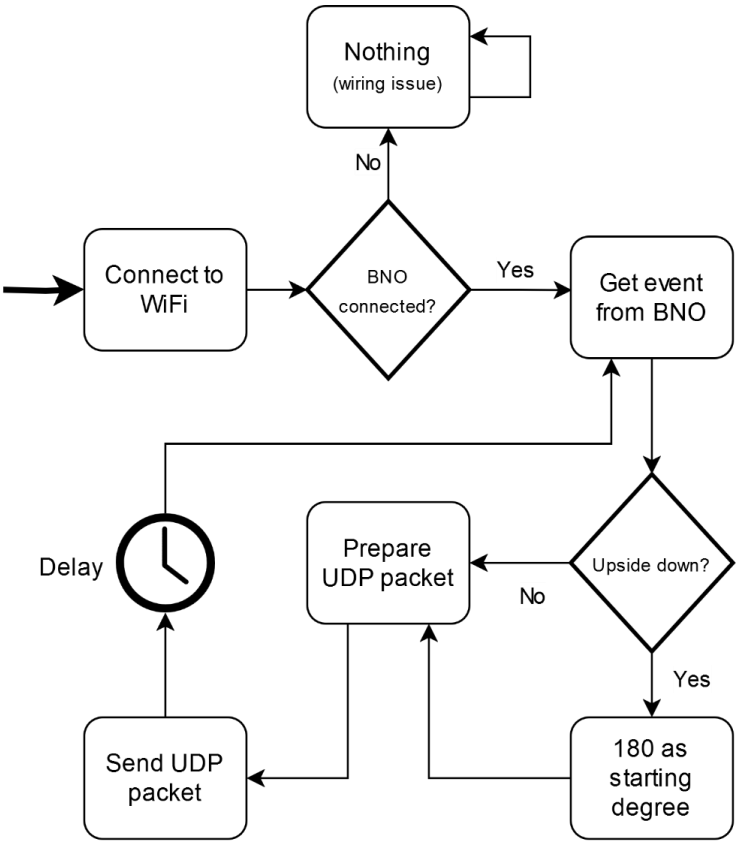


Figure 8: Flow chart motion controller

4.2 Console

Our code for the console(the part with the matrix) is more complex in comparison to the motion controller. Here we had to design a proper structure of the code using OOP principles. For example, we use inheritance or we separate a view from the game functionality. It becomes more clear later when looking at the UML diagrams.

Instead of using C++, we use CircuitPython. There was no specific reason to use it apart from convenience. During our feasibility study, the first library that worked was in CircuitPython and we continued to write in this language.

To understand how to use the matrix we had to look at the following libraries: CircuitPython Display Text and CircuitPython Bitmap Font Library. The code in the examples did not fully suit us but it was helpful enough to start writing. For the font we used MatrixLight8 from https://github.com/trip5/Matrix-Fonts. For the secret information Wi-Fi SSID and password, we used the same approach as was for the motion controller.

In general, the code can be split into 2 big parts: motion data processing and console (which consists of games, main menu and game over). In theory, we could have used a single while loop and call functions from there but it would have created a mess in the code, which would make it hard to read and extend the code. Besides, it would be a bad practice in general, as usually when needing such separation people use asynchronous(non-blocking) style or multithreading. Multithreading was not available

for the CircuitPython on our Raspberry Pi Pico W. Besides, it would have created an additional complexity due to race conditions. Hence, we had to use asynchronous style provided by *asyncio* library.

Before diving into the main part we would like to show the properties that are used in the entire application. It is important to mention again that CircuitPython has no enums, so we had to use enum-like classes.

- *Direction's* class fields are constants for each direction and a few maps: from string to direction, from direction to the speed (e.g. *LEFT* is *(x:-1,y:0)*) and from one direction to the opposite direction. Besides, the class contains all necessary functions related to direction functionality. For example, convert *State.orientation* to direction. Another example, take two directions and return true if they are opposite. The class is easy to extend if needed.
- *State* module contains actually 3 classes and each represents a state.
    - *Phase's* class fields are just *MENU* and *GAME*.
    - *GameOption's* class fields are just *SNAKE, PONG, CUBES* and a string array that contains a string of each game name.
    - *State's* class fields are direction and orientation(motion controller data). It also has auxiliary functions to update that state based on the input. For example, when passing a string we update the orientation to *(-10,10)* value if the argument of a function is in the following format: *"z:-10,y:10"*.
- *Secrets* module contains all constants that we cannot leak
- *Constants* module contains all constant except the secrets. One of the most important constant there is *WORLD_SIZE*, which is used to scale the game if needed (e.g 3x3 for pixel and not 1x1)
- *Color's* class fields are constant for each colour and a map for the hex value for each colour. Besides, the class contains all necessary functions related to colour functionality (e.g get random colour).

Properties make it very easy to separate miscellaneous things from other parts of the application. For example, if we want to add another constant we just go to the constants file and define there a new constant, or change the configuration there if we want to change it for a few games. Another example, if we want to add a new colour, then we know that all colours are defined in the properties in the Color class. The UML diagram for these properties is shown in figure 9 on the next page.

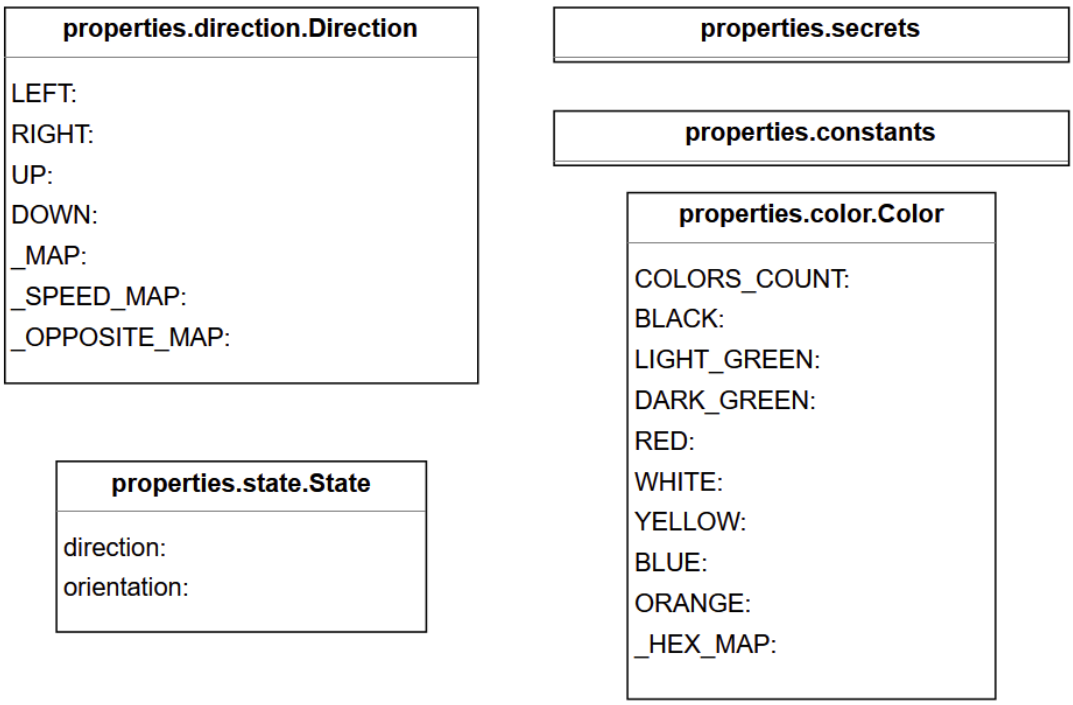


Figure 9: UML diagram properties

Now, we explain the main flow design choice. The idea is to create two asyncio tasks:

1. For the controller, which can be either buttons or UDP server.
2. For the console to run games, main menu, game over.

Then, we gather on these tasks and each of them uses asyncio to not block the other task. This approach separates the code nicely.

This functionality is implemented in the *code* module. It has 2 main functions. One is for the controller with the buttons and another is for the UDP server. In both cases, we first initialize the matrix class.

The matrix class, of which the UML diagram is shown in figure 10, is used to set up the matrix, draw pixels, update tile grids or refresh the display. In the set up part, we configure the pins for the display and create colours pallet, tile grid and group. Each of these is configured in a separate auxiliary function. What each of these is used for can be read in the library mentioned previously. To draw pixels we wrote many auxiliary functions with reusability and extensibility in mind (e.g *draw_square* which uses *draw_custom_square* which uses *draw_pixel*).

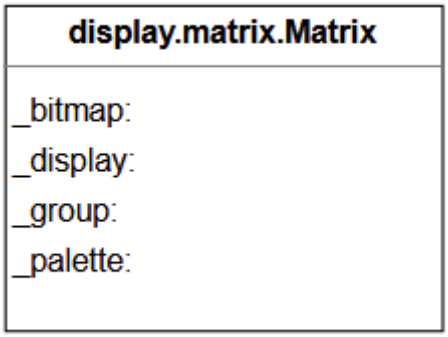


Figure 10: UML diagram display

After that, we try to set up the server or the buttons, and create asyncio task.

- For the server, we first try to connect to Wi-Fi. The Wi-Fi functionality is separated from the server. Their theme may look alike but actually, there is no need to combine them both in one class and create a mess by doing that. Besides, it is possible to have a different server class later and recreating the Wi-Fi part in it would not be good according to reusability principles.
    - *CustomWifi* class is quite simple: it imports secret constants from the secrets module, and has 2 functions. The first function is used to connect to Wi-Fi. The second function is used only for debugging to scan networks and it was proved to be useful a few times, so we kept it.
    - *Server* class used to be more difficult when using TCP but now it has become much simpler when using UDP due to connectionless communication. The constructor of a class has a callback function. This callback function is used to update the *State* class.
      In general, the *Server* class contains functionality to set up and asynchronously run the server. When running the server we loop until the termination of a program. In each iteration, we try to read data into the buffer and if it is correct we call the callback function.

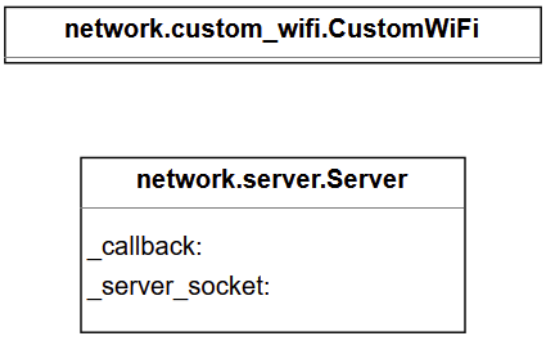


Figure 11: UML diagram server

– For the buttons, we create 4 instances of *Button* in the *buttons_controller* with *LEFT, RIGHT, DOWN, UP* directions. *buttons_controller* checks in a while-loop if some buttons were pressed and if so we call the callback function, until the termination of a program.

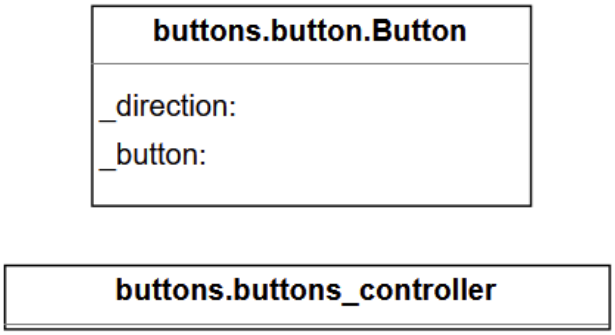
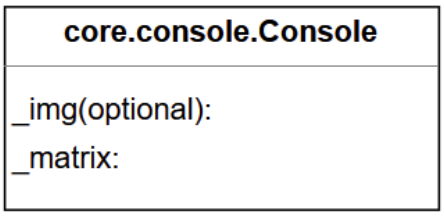


Figure 12: UML diagram buttons

After the controller task is set up, we set up the console(games, main menu, game over), and create asyncio task of it. The console has an optional image parameter to show the logo, which we can set using the function. In the case with buttons we do not call it, as we worked with buttons only for convenient development and adding a logo to it would only introduce an additional delay.

In the running function of the *Console* class, before running the while-loop we initialize *MainMenu* and *GameOver* classes. Of all these classes, the UML diagram can be seen in figure 13. *Console* can be seen as the core and *MainMenu* and *GameOver* as scenes. Each scene is separated from each other in a class, and each scene has matrix as the parameter in the constructor, as scene has to draw pixels.

In the while-loop, we choose what to do depending on the phase. If phase is *MENU*, then we show the menu and asyncio await for the user to select the game. If phase is *GAME _RUNNING*, then we asyncio await on a game. If await of the game returns info that we lost, then we show the game over scene, otherwise, we go to the menu scene and repeat again. In the game over scene, we can go to the menu scene or restart the game.



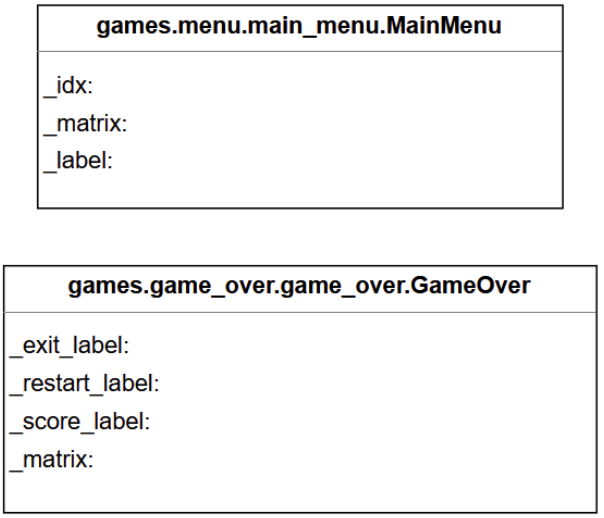


Figure 13: UML diagram games classes

After the main flow of the project was explained, we can dive into our OOP design of the games. For each game, we stick to the same design principles and each game nicely separates the controller and the view:

- *Game* is the controller of the game which runs asynchronously the while-loop until the end of the game. Each *Game* inherits the *GameParent* class, which makes it possible to easily introduce some function or field for each game, without repeating the code. See figure 14.

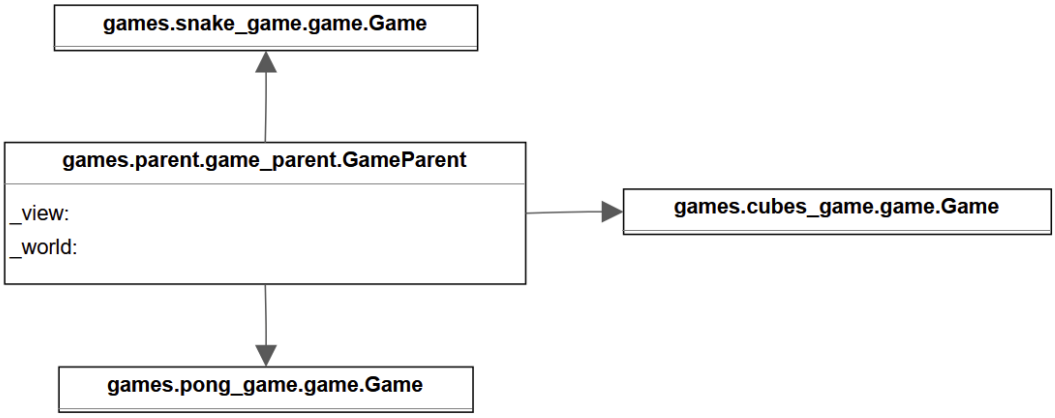


Figure 14: UML diagram games.game

- *World* is the state of the game. Each *World* inherits the *WorldParent* for the same reason as *Game*. In figure 15, this is shown in a UML diagram.

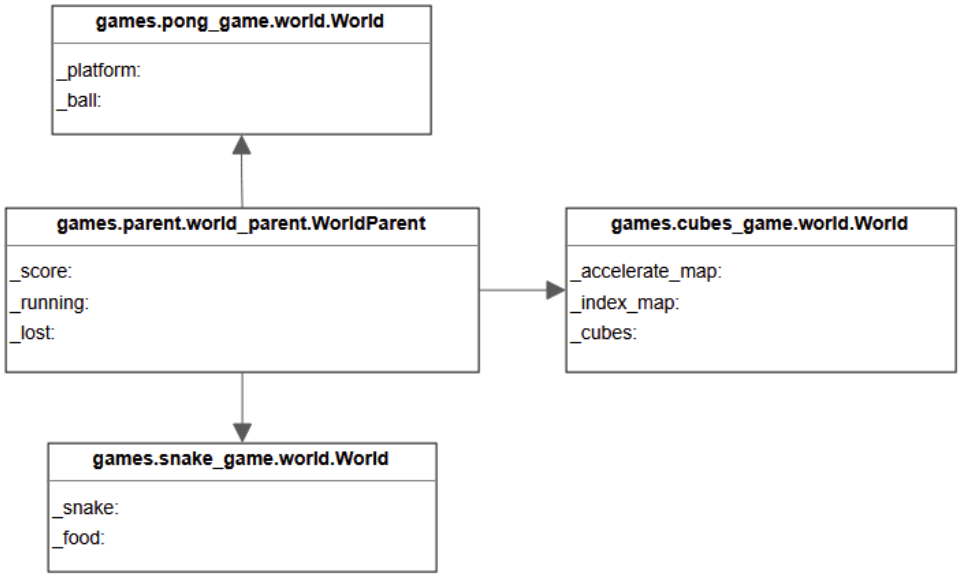


Figure 15: UML diagram games.world

– *View* is the view of the game. Here we have 2 parent classes and one parent class is a subclass of another parent class. *ViewParent* is the first parent class and only cubes game inherits directly. *ViewBorder* is the second parent class and it inherits *ViewParent*. The reason for that is that some games views have borders and some do not. Inheritance here is used for the same reasons as previously and shown in figure 16.

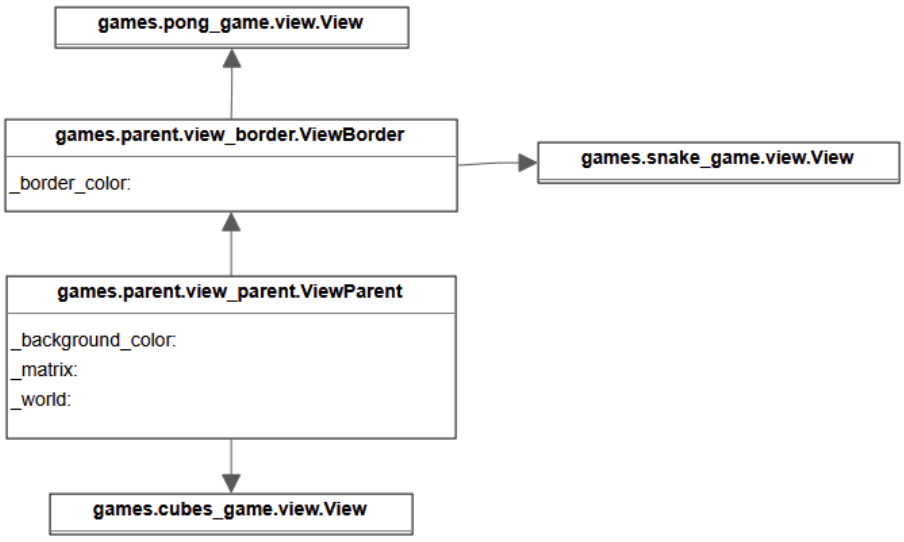


Figure 16: UML diagram games.view

– *Entity* is the entity in the game. Each instance of it is created in *World* classes and has world as the parameter to get access to the state of the game (e.g *Snake* should know through the world if it on the *Food* position). Each Entity has coordinate and it can be seen on the diagram that *EntityParent* inherits *Pos* class, which inherits *Coords*. The *Coords* and *Pos* classes could be combined in one but we decided to separate them.
Some entities should have collision functions and some not. Those that should inherit *EntityCollision* class, which inherits *EntityParent* class. For example, again *Snake* can collides with the *Food*, so it needs collision functions, but *Food* does not, since *Snake* already has them. How this works is shown in the UML diagram in figure 17.

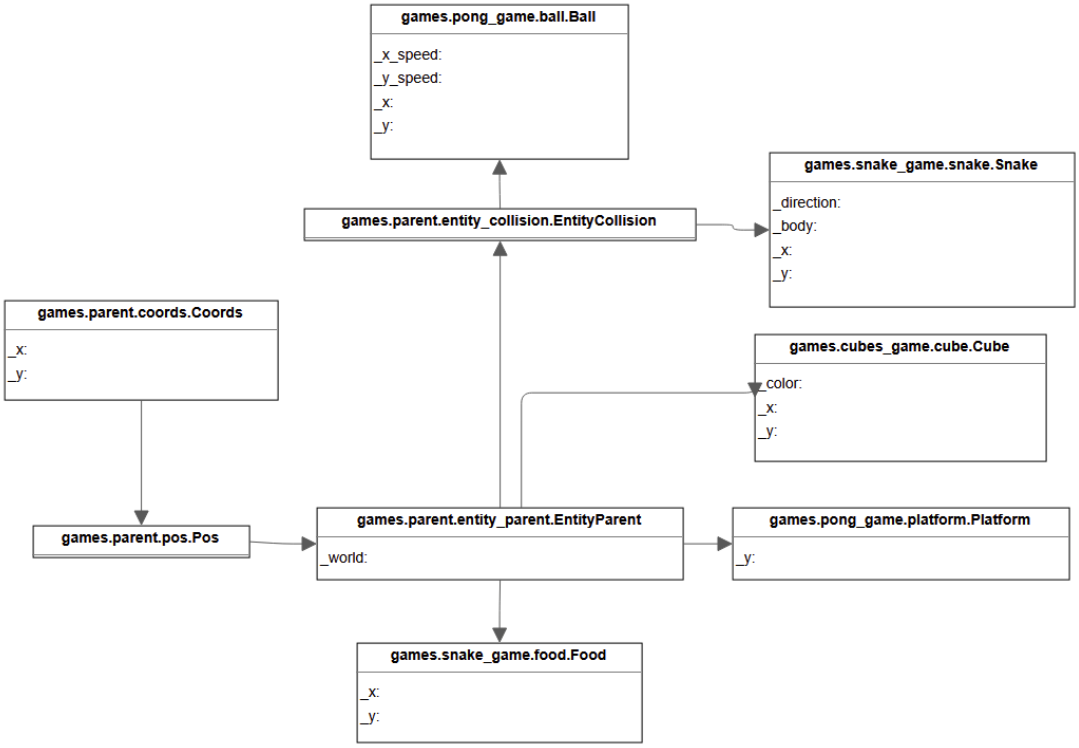


Figure 17: UML diagram game entities

After discussing the UML diagrams we provide the flow chart (figure 18). The flow chart is more complex than the one for the controller due to the asynchronous style. To represent it we have *Event Loop* at the centre which coordinates which task should run. We also use different colours to make everything more clear.

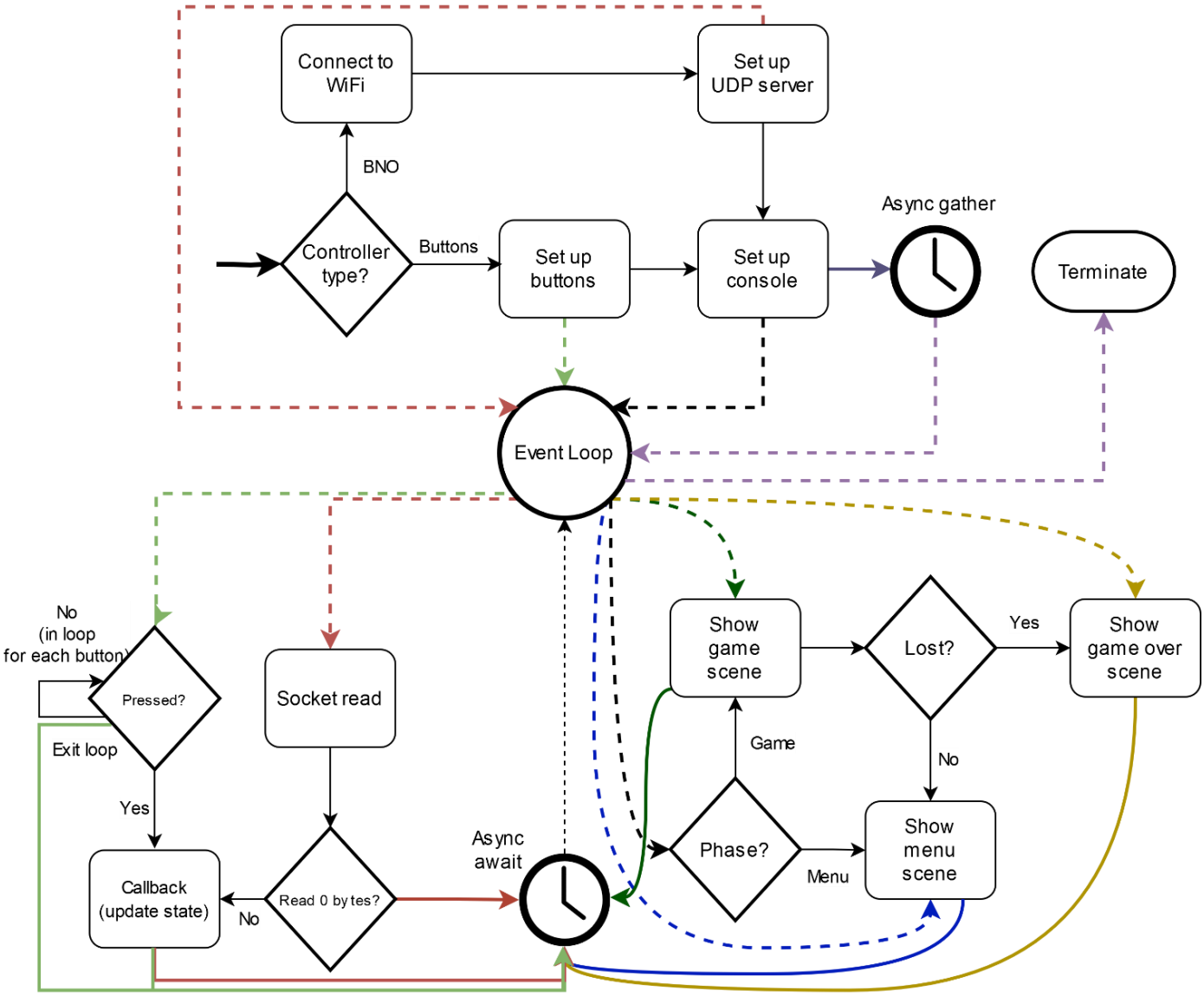


Figure 18: flow chart

4.3 Overview of used projects

In this paragraph, we credit the projects of other people and how we used them.

1. Epler, J. (2020, April 20). *RGB LED Matrices with CircuitPython*. Retrieved from Adafruit: https://learn.adafruit.com/rgb-led-matrices-matrix-panels-with-circuitpython/example-two-line-colorful-text-scroller

   Used to learn the basics of CircuitPython and RGB matrices, such as scrolling, which was later used for all games.

2. Trip5 & lubeda. (2024). *Matrix-Fonts*. Retrieved from GitHub: https://github.com/trip5/Matrix-Fonts

   Used as font to display text on the matrix.

3. LeBlanc-Williams, M. (2023, October 11). *Adafruit MatrixPortal M4*. Retrieved from Adafruit: https://learn.adafruit.com/adafruit-matrixportal-m4

   Used as inspiration for the cubes game. However, the code was written without looking at their code, as they use a different board and a MatrixPortal, so we had to design our own code for the board we had together with the motion-controller.

5 Process Report

In this chapter we will reflect on our process (so far) and what we could have done better in this process. For this, we will first talk about the technical implementation, then the learning and implementation process will be discussed and the final paragraph will be about the sustainability of the process.

5.1 Technical implementation

Our current project does what it is supposed to do and we think it looks alright. We focused a lot on the code quality and the functionality of the games. The wiring is sometimes a bit messy, but with the amount and the placing of wires that is needed for the matrix, we think this was not really avoidable. We are happy with the code and how the code works. The games we made look nice (for the matrix we have) and work as intended. The sensor also sends the correct z and y value to the matrix, even if it is upside down like it is in the final version of the project. The Wi-Fi connection and the complexity of the code does create some delay between the input and the output. This is something that might have been better with either Bluetooth or a wired connection. For sustainability and time purposes and not knowing if this would actually improve the performance, we decided not to order the Raspberry Pi Pico with Bluetooth. If we were to do it again, we could have tried this in the feasibility study, but because of the broken screen, there was no time for this.

5.2 Learning and implementation process

We learned a lot from the project, especially that a project that seemed quite doable at the beginning of the course becomes quite hard if you get multiple setbacks. When we started out the feasibility study with the BNO055 sensor, we realized our original plan at the start of the course was not complicated enough. This was not an issue, as we knew part of the feasibility study was that you could change, simplify or – in our case – complicate the project if necessary. We made the decision to involve an LED matrix and make games for this, which can be controlled by the BNO055 sensor.

When the first LED matrix did not work (see paragraph 2.2) and only showed a few pixels, we were worried that our project might not be feasible and that we were doing a lot wrong. This was luckily not the case as the matrix was broken, but it held us occupied for a few sessions and also made us doubt the project. In the end it turned out well for us, but it did make us extremely careful for our next screen, as we obviously did not want this one to break as well. Next time a thing does not work, we will probably still start with looking at our code and wiring to check if that is wrong, but will be more aware that the issue could also be in the materials that we use.

Coding the games on a matrix took a lot more time and debugging than programming these games normally for a laptop, phone etc., also because it was our first time using

CircuitPython. CircuitPython misses some libraries and functions that would have been useful, for example enumerations and NumPy, but we think we do now have a lot of good quality code, which is also extendable. For safety, we uploaded our code to GitLab often. We also chose to include more of a demonstration of how our code works with the blocks-falling animation and made a menu to switch between the games and this.

We are happy with the games we have, but focusing on making three games took a lot of time, which we could also have used to incorporate different sensors or functionalities and make one big game. This would have been a risk as we would not be able to test the feasibility of all the elements and we might have ended up with an end project that would not work. However, it might have also increased the complexity of our project. Complexity was an issue in our project all the time. We wanted to start off with a simple project and, if possible, make it more complicated along the way. We learned that this is a good way to attack the problem, but might also take away a bit of ambition. Our project was complicated to make for us, but might not look that way and we think we might be able to do something more unique in a next project. We could have done better on the creativity, but we are happy we were able to recreate the idea we had in mind before starting on the MVP.

5.3 Sustainability

In this paragraph, we will explain our decisions made on sustainability in the process. We will first look at the hardware and mechanics choices in 5.3.1 and then discuss the sustainability in the software in paragraph 5.3.2. In the end we will also discuss possible improvements in 5.3.3.

5.3.1 Hardware and mechanics

From the start of the project, we tried to use as much elements as possible from the inventory list to avoid having unnecessary costs, shipping and materials. We did order one sensor, namely a gyroscope, which we did not end up using as is turned out the BNO055 already had a built-in gyroscope. This was obviously not a great decision in terms of sustainability. For making the frame around the matrix, we used materials from the Totem building boxes instead of trying using the 3D printer, which could have wasted quite a lot of plastic. To make the frame, we had to saw some pieces of plastic into smaller pieces, but the frame can be disassembled after the project and all materials could be reused. This would probably not have been the case if we 3D printed it.

Some components, like the breadboard and the Velcro, had a sticker side and are stuck onto other components with that side. As we removed the covers from the sticker side, these components can probably not be reused, or only by adding additional glue. However, to get to this point the stickers would have to be removed from the components they are stuck on to. We think this is not too hard with enough force and maybe a sharp object, but can cause some issues for reusability.

We also tried to choose boards that we thought have enough data and processing power for our project with the Raspberry Pi Pico W and the LOLIN D1 mini PRO instead of using a full Raspberry Pi and an ESP32 board, which have better processors, but will also probably use more energy.

5.3.2 Software

In the software we added a few things to help with energy consumption and hence sustainability. For example, when someone loses with Snake or Pong, they get sent to a screen asking to play again, instead of just restarting the game which has a much higher complexity than showing a menu over and over. This way, we save energy when players stop playing or do not pay attention. The same goes for Cubes, where we make sure that the game stops if the cubes are stationary for ten seconds. This way there is lower power consumption.

In the controller, the BNO055 automatically will send signals less frequently and go into a low power mode (Mischianti, 2023) if no movement was detected for more than 5 seconds and once it notices there is movement again, it will start sending signals more frequently again. This could be changed by setting a standard mode in the code for the BNO055, but we decided to keep it this way as it also helps with sustainability and power usage.

5.3.3 Improvements

Although we tried to make our project sustainable, there are a few areas where we could have improved on it. Ordering a gyroscope even though we already had a sensor with a gyroscope function was a waste of money and materials. In hindsight, we did not use any of the other functions of the BNO055 sensor so we could have used this gyroscope instead of the BNO055.

Also, we are not completely sure whether the first LED matrix was already broken or that we have accidentally broken it in some way. We think we did not break it, but it is always good to be careful when connecting wires and applying pressure on things when you have to attach or remove something from it. We tried to be even more careful on the second screen, which still works.

Other than that, our project does not really contribute to sustainability in society, apart from the thought that people could get happy from playing games and a happy society is a sustainable society, but this seems a bit far-fetched. It does not help against climate change, does not increase life expectancy or anything like that. The type of project we made does not really allow us to do this and we do not really know on how we would be able to make this kind of impact by improving our project, but for a future project it would be nice to have this.

Finally, if we were to do this again, we could do more research into which boards use the least amount of power and materials for the end goal of the project. For this project we guessed that the boards we used would be good enough for our project, but not too good, but did not do a lot of research into this. Maybe we could even have used some boards that use even less power, which would help with the sustainability.

6 User documentation

There are two possible ways to use the device. You can either use the controller as a real controller or connect the controller to the screen and use the screen both as a controller and as a screen. You can even switch between the two in the middle of the session. In this chapter, both ways will be explained.

6.1 Setup

1. Check if there are no loose wires. If there are any loose wires, do NOT plug in the device.
2. Check if both power banks are charged.
3. Connect the microcontroller on the breadboard which is stuck on a power bank to that power bank.
4. Connect the other microcontroller on the other breadboard to the other power bank.
5. Plug in the LED Matrix into a power socket.

6.2 Controller not connected to screen

The LED matrix should be turned on now. You should see a menu showing the name of a game and some arrows. The next steps are:

1. Hold the controller in your hand, with the USB cable towards yourself and hold your hand in a horizontal position.
2. Raise the top of the controller to go to the next game (up). Lower the top of the controller to go back (down).
3. When you have found the game you want to play, tilt your hand to the right to start the game.
4. Snake: try to eat the apple without hitting the walls or your snake in the process. To change directions, these steps can be taken:
    - Move up; raise the top of the controller.
    - Move down; lower the top of the controller.
    - Move right; tilt your hand towards the right.
    - Move left; tilt your hand towards the left.

    You can only make 90 degrees turns. The game is over when you hit your snake or the walls.

    Pong: make sure the ball does not hit the wall on the left side of the screen by moving the pong bat up and down. This can be done in the same way as in Snake. If the ball hits the left wall, the game is over.

Cubes: Move your hand in any way and see how the cubes fall. They should move towards the place you are pointing the controller to. The game stops if you stop moving.

5. When the game is over, you enter a menu. Follow the instructions here.
6. If you are done playing, unplug the microcontrollers and the LED Matrix to save power.

6.3 Controller connected to screen

Use the Velcro on the back of the LED matrix and power bank to connect the screen to the matrix. Test if they are connected properly. Hold the screen horizontally. The next steps will be almost the same as in paragraph 6.2, where the only difference is that you have to hold the handles of the screen now, and movement is based on gravity. So, if you want to move left, lower the left side of the screen, if you want to move down, lower the bottom of the screen and so on.

Bibliography

Mischianti, R. (2023, January 9). *BNO055: power modes, accelerometer and motion interrupt – 4*. Retrieved from Renzo Mischianti: https://mischianti.org/bno055-power-modes-accelerometer-and-motion-interrupt-4/

Raspberry Pi Ltd. (2024, October 15). Raspberry Pi Pico W Datasheet.

reichelt elektronik GmbH. (n.d.). *Ontwikkelaarsborden - Sensor met verschillende functies, BNO055*. Retrieved from reichelt: https://www.reichelt.com/nl/nl/shop/product/ontwikkelaarsborden_-_sensor_met_verschillende_functies_bno055-235479?country=nl&CCTYPE=private&LANGUAGE=nl

WEMOS. (n.d.). *D1 mini Pro*. Retrieved from WEMOS: https://www.wemos.cc/en/latest/d1/d1_mini_pro.html