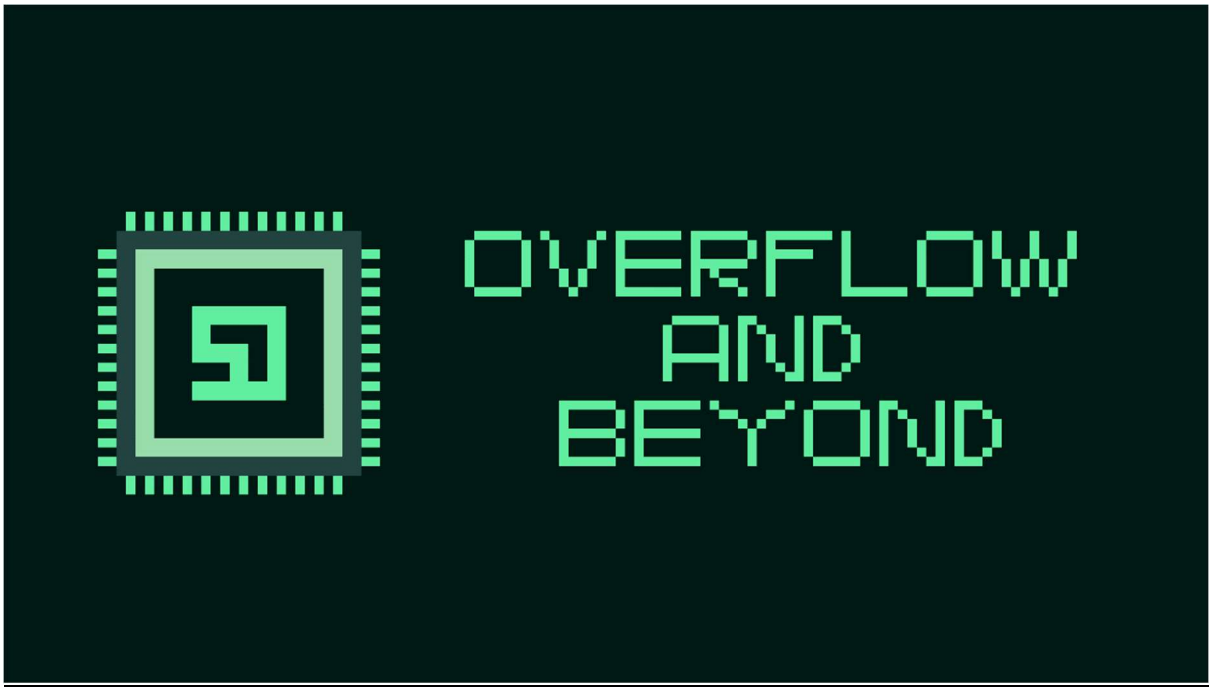


PART 1: INTRODUCTION

This is the project devlog :

Within this manual, all the technical aspects regarding the operation of the object will be explained in detail, from a mathematical, hardware and software point of view.

These aspects mentioned above will be treated in order, one at a time, in their entirety.

The pages of this manual will also feature links to external resources that can be used to expand its effectiveness.

Good Reading

ESP32 ORTHOGONAL PROJECTOR MANUAL © 2023 by OVERFLOW is licensed under CC BY 4.0

PART 2: MATHEMATICAL OPERATION

To render the figures to be orthogonally projected onto the screen, the renderer uses a rendering pipeline composed of the following three phases: rotation, projection and drawing. The first phase, rotation, uses specific formulas to rotate the three-dimensional points on the three different axes (X,Y ,Z), a rotation that occurs one axis at a time, which allows you to rotate the figure as you wish. The second phase, projection, recovers the three-dimensional points that have been previously rotated and projects them, transforming them into points with two-dimensional coordinates. In the current version of the renderer, two projection algorithms coexist, which can be chosen at will: there are an algorithm capable of implementing orthogonal projection (which ignores the Z coordinate of three-dimensional points) and an algorithm capable of implementing perspective projection (with focal length = 1), useful for understanding the true orientation of the figures more easily. The third and final phase, drawing, takes the two-dimensional points already projected, draws them on the screen and connects them with lines. At the end of this procedure, the projection of the starting three-dimensional figure is obtained.

Let us now analyze in detail the functioning of each of these phases:

Phase 1 >>> rotation: {

Rotation is one of the most important, if not the most important, functions that the renderer makes available. This takes up a huge part of the source code, as well as being the component that took the longest to create. There would be several ways to explain this topic, but I think the best way is to start from the beginning, just like I did, which allows you to understand what the different advantages and disadvantages of the different implementations are.

RO.1 >>> https://en.wikipedia.org/wiki/Rotation_matrix

By searching online for the topic "3d rotation ", it is very easy to come across this Wikipedia page (RO.1), which seems to be precisely the solution to the problem we are trying to solve: we are immediately presented with a mathematical formula capable to rotate a certain number of three-dimensional points on all three axes at the same time, the only problem is that it is too, too complex.

Even if you could figure something out it would still be too complicated to program, as well as take too long for the processor to execute. It is therefore necessary to find a simpler approach to the problem, an approach which, if we pay a little attention, is already presented to us at the beginning of the page, even if not completely.

In fact, at the beginning of the article, two-dimensional rotation is also discussed, which, even just by taking a quick look, without dwelling on the individual formulas, appears much, much

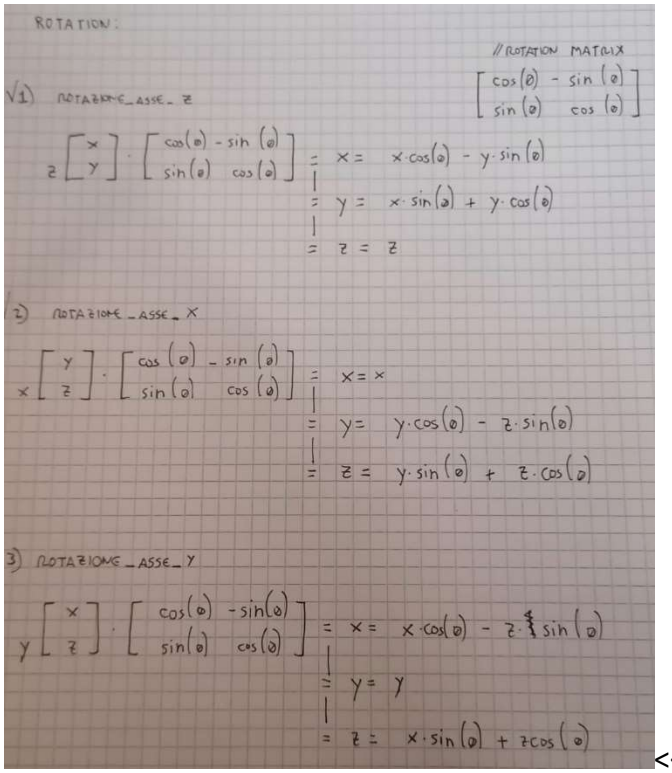
simpler. Going into more detail we discover that, here too, as for three-dimensional rotations, we are presented with the rotation matrix, as well as formulas derived from the multiplication of the matrix itself

There is no comparison: the two-dimensional rotation matrix appears extremely simpler than the three-dimensional rotation matrix: the difference is enormous. Then analyzing in detail what is presented to us, we then notice a very important detail: from what we are shown, matrices are nothing more than a way of representing formulas, which can be derived by multiplying the matrices themselves. This last factor is much more crucial than you might think, given that having the possibility of using "pure" formulas instead of the entire rotation matrix allows you to greatly simplify the development, as well as the performance of the processor.

Although this discovery has greatly simplified the question, one aspect still remains to be clarified: how can one or more two-dimensional rotations be used to obtain a three-dimensional rotation? The solution to this simple question literally took me months, and not because I didn't know how to use the formula that had already been provided to me by Wikipedia, but because I didn't know how to obtain the formulas for the other axes.

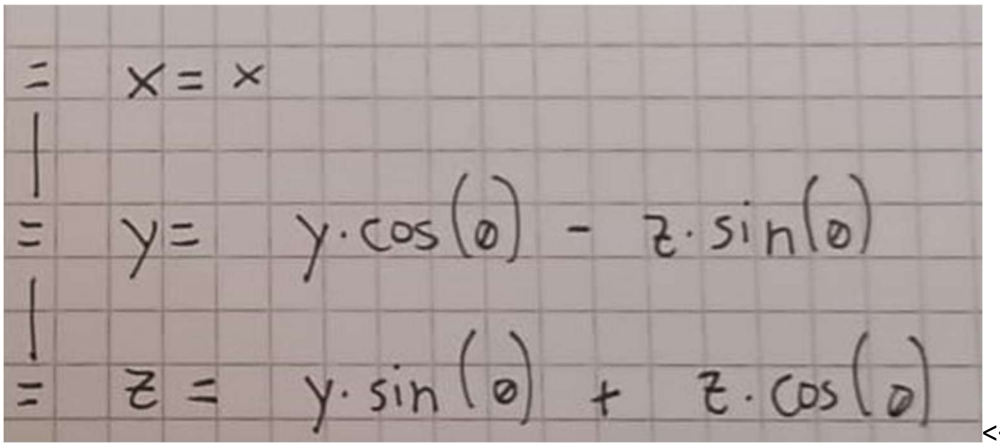
Paying attention to one of the images included in the article, it is easy to realize that the formula provided to us is capable of performing a rotation only on the Z axis, which represents a problem, given that to give the With the illusion of an orthogonal projection, it is necessary to have complete freedom in the rotation of the starting three-dimensional figures.

After not being able to find the formulas for the other rotations on the net, I had to do the only thing I could: I obtained them starting from the rotation matrix we talked about on the previous page, trying to guess the same procedure that was been used in the article, and adapt it to my needs, which turned out to be a success, even if it took weeks. At the end of my attempts I found myself with all the formulas I needed for the rotation (IM.5).

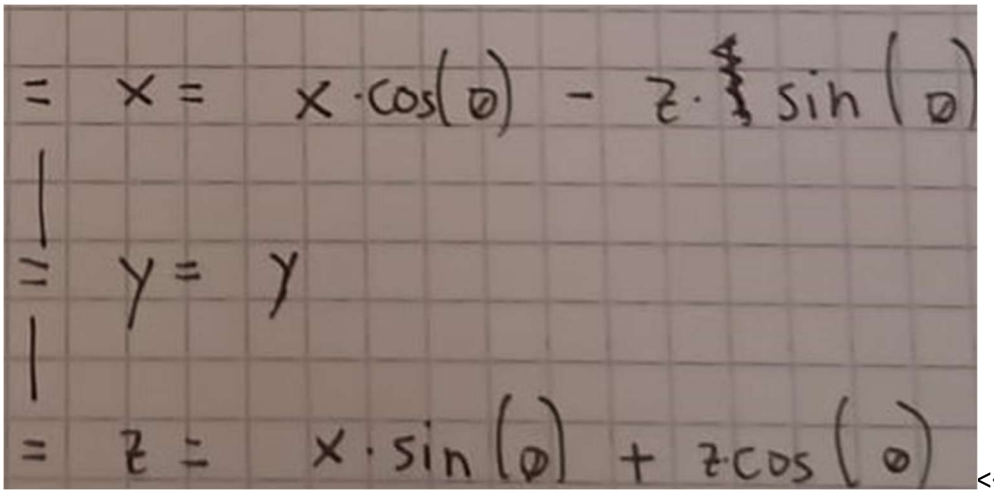


<<< IM.5

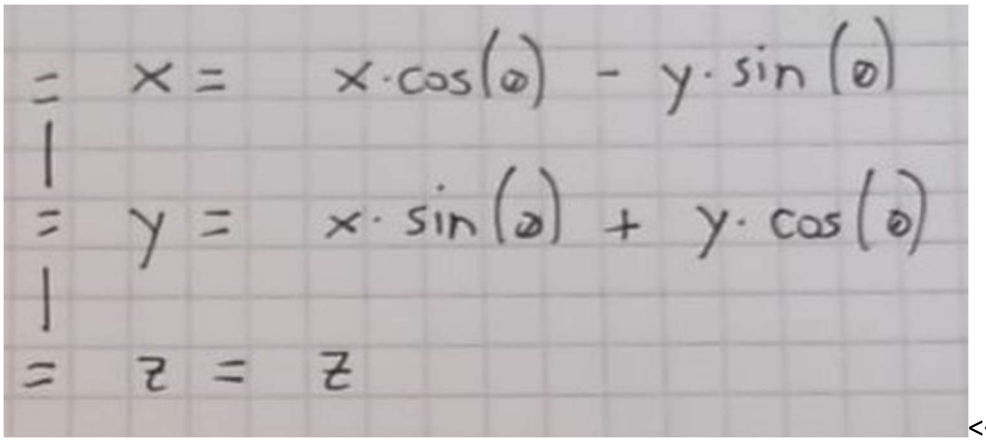
In particular, I obtained the three formulas necessary for the rotation on the X axis (IM.6), the three formulas for the rotation on the Y axis (IM.7) and the three formulas for the rotation on the Z axis (IM. 8).



<<< IM.6

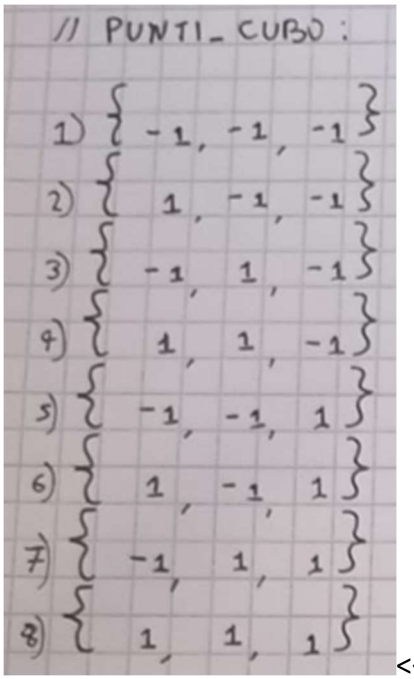


<<< IM.7

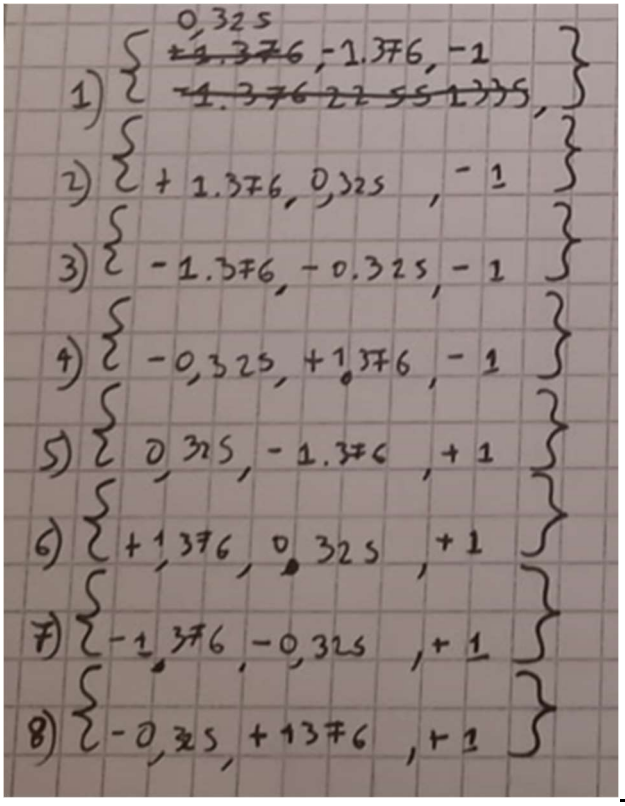


<<< IM.8

Once I had managed to obtain the formulas, all that remained was to verify them, which I did by simulating on paper what I should have made the program do: first we start from the three-dimensional points of the figure we have chosen (in our case a cube) (IM.9) and then apply the formula relating to the axis on which you want to obtain the rotation. At the end of the calculations, the coordinates of the rotated three-dimensional points are obtained, which describe the figure, rotated by a certain angle θ (IM.10).



<<< IM.9



<<< IM.10

As for the rotations, that's all I needed to know, but, of course, I consulted other sources as well as Wikipedia. I leave the complete bibliography of the resources I used below. (Only related to rotations)

https://www.youtube.com/watch?v=hFRlnNci3Rs&t=1227s <<< RO.2

https://www.youtube.com/watch?v=vfPGuUDuwmo <<< RO.3

https://www.google.com/search?q=arduino+3d+renderer&rlz=1C1CHBF_itIT1011IT1011&oq=arduino+3d+renderer&gs_lcrp=EgZjaHJvbWUyBggAEEUYOTIICAEQABgWGB4yBggCEEUYQNIBCTQzNjVqMGoxNagCALACAA&sourceid=chrome&ie=UTF-8#fpstate=ive&vld=cid:cc6e2993,vid:kBAcaA7NAlA,st:0 <<<RO.4

}

Phase 2 >>> projection: {

After managing to rotate the points, I immediately found myself with another, albeit simpler, problem. I didn't know how I would be able to project the three-dimensional points I had obtained onto the screen. Once again Wikipedia was of great help to me, and this time I managed to find exactly what I needed, without any need for modification (RO.5).

https://en.wikipedia.org/wiki/3D_projection <<< RO.5

In the current version of the renderer there are, as I already said in the introductory part of this topic, two projection algorithms: one capable of performing a perspective projection, and one capable of performing an orthogonal projection. As for the Orthogonal projection, I already knew how to implement it, so I didn't have to carry out any research on it, it is sufficient to completely ignore the Z coordinate and consider only the X and Y coordinates, which are precisely the coordinates of the two-dimensional points that will then have to be shown on the screen. As regards perspective projection, however, I had no idea how I could implement it, and therefore I had to carry out a short research, which made me discover the formula for "weak perspective projection", a simplification of the more complex formula for perspective projection.

This simplification, unfortunately, also introduces a compromise: the focal length cannot be modified, and is equal to 1, which in any case I don't think is a huge problem, given that, originally, I had decided to introduce perspective projection in the renderer just to use it as a supplement to orthogonal projection, which sometimes produced results that were a bit difficult to interpret, due to the fact that there is no depth information.

}

Step 3 >>> Drawing: {

After finally managing to obtain consistent results from all the previous phases of the rendering pipeline , all I had to do was proceed to "draw" the points and lines obtained on the screen. This last phase did not require me to apply complicated mathematical formulas for its realization, given that thanks to the help of a special library which I will talk about in the SOFTWARE section, I was able to use a ready-made algorithm for calculating the lines,

which otherwise It would have required a lot of time and effort. The only thing I needed to do, from a mathematical point of view, was to use simple additions and subtractions to align all the points and the different graphic elements within the quadrant that had been assigned to them.

}

PART 3: OPERATION AND HARDWARE COMPONENTS

If you hadn't already understood, this project requires considerable computational power: it is necessary to calculate the points and draw them on the screen, dozens of times every second, repeating the process for each of the three quadrants. These types of requirements are not achievable with just any processor, but something modern, fast, updated and, above all, not too expensive is needed. The world of microcontrollers is very diverse and complex: on the one hand there are the simplest to use processors, which have been used to build world-class hardware such as the Arduino UNO, while on the other there are industrial grade microcontrollers, those capable of providing good performance, security and a greater amount of memory. Before jumping into this project, I had already had a great deal of experience with the ATMEGA 328P, an iconic, world-famous processor, which, while simply fantastic, was not very suitable for the difficult requirements for this project. Just to have a point of comparison, the ATMEGA 328P has 32 kilobytes of flash memory (the one in which the program is stored), 2 kilobytes of RAM memory (used to keep track of the different variables used by the program) and a speed of clock of 16 megahertz (MHZ): specifications that can be considered pathetic compared to the processor I decided to use for this project: the ESP 32 S3. The ESP32 processor series, is a very modern processor series, designed with industrial development and automation in mind. This series of processors allows you to use modern connectivity standards, such as WIFI and BLUETHOOT, coupling this versatility with respectable performance, more than suitable for what we intend to do. The ESP 32 S3 is a special variant of the ESP 32, and is the most powerful. By checking the datasheet (RO.6) you can easily discover that this variant offers a clock speed of 240 MHZ, 15 times greater than that of the ATMEGA 328p, 540 kilobytes of SRAM and up to a maximum of 16 megabytes of flash, a simply overkill amount of memory for most of what you could think of doing on a microcontroller. The greatest peculiarity of this processor is the presence of not one, but two " Xtensa ® LX 6" (IM.12) cores, which allows you to implement much more complicated programs, dividing the work between the two processors.

https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

RO.6 ^^

Without going into other hardware details, you can easily understand that the ESP 32 is practically the perfect choice for this project: it has excellent performance, great availability, a negligible price, and a large community of developers always ready to help resolve compatibility issues with different software. Despite this, if you wanted, you could also aim even higher, opting for a real microprocessor, perhaps an ARM architecture processor like the one integrated inside the Arduino giga (RO.7), a dual core monster , which has even higher clock speeds (480 megahertz on one core and 240 on the other [they are of different types]), a choice which however would be thwarted by the greater difficulty in development, given that being a new product, literally released this year, it could does not yet offer good compatibility with certain libraries.

https://store.arduino.cc/products/giga-r1-wifi <<< RO.7

However, if you do not want to make compromises, there is also the "Arduino PRO" (RO.8) series of industrial processors and boards, which I would sincerely advise against, due to their very high price, which, although justifiable in given the added features, it's not justifiable to make lines move on a screen, since, at that point, it would literally be easier to use a real video card.

https://www.arduino.cc/pro/ <<< RO.8

In addition to the processor, it is obviously necessary to interface with the screen on which you want to view the orthogonal projection: to do this, you can use the VGA protocol, which, in addition to being relatively simple and well supported by a large number of libraries, also allows to interface with the most modern screens, given that a VGA to HDMI converter can be found for a few dozen euros. From a practical point of view, in any case, microcontrollers practically never offer a video interface already set up "by the factory", which makes it necessary to obtain a VGA cable, strip it, exposing the individual wire strands and then solder the other wires between the cable and the processor. A simpler but more expensive way of doing this exact thing is to buy a specific printed circuit board (RO.9) that has a VGA port on one side and wire connectors on the other, which would allow you to avoid the sacrifice of an innocent cable.

https://www.amazon.it/AAOTOKK-Morsettiera-Adattatore-Terminale-Connettore/dp/B08155NQRV/ref=sr_1_2?crid=E0CZ6DJD1B3A&keywords=vga%2Bbreakout&qid=1697815081&sprefix=vga%2Bbrea%2Caps%2C86&sr=8-2 &th= 1

Ro.9 ^^

The connections between the cable and the processor may vary depending on the library and the processor itself, here I will only indicate the connections to be used for the ESP32 S3 with the “ESP32LIb” library created by the user “ bitluni ” (RO.10)

I will go into more detail about how the library works in the software section.

https://github.com/bitluni/ESP32Lib <<< RO.10

The VGA protocol is an analog protocol, which means that information is not transferred in the form of bits, but in the form of signals, which are then reinterpreted by the screen to understand which color to assign to which pixel. From a practical point of view we are interested in 5 different cables: the signal for red, green and blue, as well as vertical and horizontal synchronization. The three red, green and blue cables transfer the information for the color, which is composed by combining the three signals, while the vertical and horizontal synchronization is used to select the pixel to which the color will be assigned. Using only these cables it is possible to make a screen work without using very complicated circuits, just follow the wiring in the diagram alongside. The only compromise for this simplicity is the fact that it is possible to use only a very limited quantity of colors, given that we are using digital signals to control the three analog color signals (VSync and HSync instead are driven by real analog signals generated by specific hardware parts present in the processor). To overcome this problem, we can increase the number of digital signals we use to generate the color signals, which however requires a specific circuit, which must be able to modulate the voltage depending on which bit is being used for " command" the analog signal, which is easily done with a " resistor ladder ”, which uses resistors of different values to compose the signal starting from the different bits (further clarifications available on the project's GitHub page [RO.9]). The resulting circuit is a bit difficult to build but allows you to achieve good results, in case you want to add a wider range of colors, which has not been implemented in the current version of the renderer.

Bitluni , in addition to having done an excellent job in creating its library, has also done an excellent job in documenting it through a long series of videos, in which it explains in detail how to prepare the hardware for the 3-bit version as well as the 14 bits. I recommend looking at them before making any decision regarding the hardware components to purchase, given that compatibility with this library is a fundamental aspect for the success of the project.

https://www.youtube.com/watch?v=muuhgrige5Q&t=34s <<< RO.11

https://www.youtube.com/watch?v=qJ68fRff5_k&t=149s <<< RO.12

https://www.youtube.com/watch?v=G70CZLPjsXU&t=279s <<< RO.13

}

PART 4: SOFTWARE OPERATION AND LIBRARIES USED

The software part, in this particular project, in addition to taking on a central role for the operation of the device, also represents a connection point between mathematics and the hardware which must actually perform all the necessary calculations. Here we won't go into much detail about the syntax of the programming language I used, but we will outline the different steps that, from a software point of view, must be implemented.

>>> Calculate the projected points and how to store them:

To memorize the points, two-dimensional arrays can be used (one array inside another) (IM.15), which allows us to independently access a point and each single coordinate of the same independently, which helps us a lot in the calculations. This approach also allows us, if desired, to change the object whose orthogonal projection is being created during Runtime, without having to restart the device, given that by changing the origin points, all the data present will also be changed in the pipeline. I would like to point out that the two-dimensional arrays must be used, in addition to storing the coordinates of the origin points, also to store all the results of the different steps of the pipeline (ES: rotated points, projected points...), which should not be a big problem, given that the ESP32 offers a large amount of memory.

```
float punti_oggetto_origine[32][3] =  {{-1,-1,1},{1,-1,1},{1,1,1},{-1,1,1},{-1,-1,-1},{1,-1,-1},{1,1,-1},{-1,1,-1}};
```

IM.15

As regards calculating the points, it is not very difficult, it is only necessary to follow the formula in the "mathematics" section, without ever forgetting that this, being a pipeline, requires that the output of the previous step is inserted directly into the 'output of the next step , without any exceptions.

>>> Draw the points on the screen:

To draw the points on the screen I used, as I have already mentioned in the hardware section, the Bitluni VGA library , capable of working with different resolutions, colors and

different buffering levels , which allows you to customize the library depending on the processor that you use and the resources you are willing to use. In this iteration of the renderer, I set the resolution to 640 * 480 and did not activate buffering , which was a forced choice due to memory issues. Since there is no type of buffering , to reach acceptable levels of frames per second, one is forced to draw the points, calculate the new points, delete the old points and draw the new ones, a process that is carried out for every single frame, since clearing the entire screen and redrawing it all over again would take too long, which can be avoided by only updating the parts of the screen that actually change from frame to frame.

>>> Input / output:

In the current version of the renderer, to manage the different input / output operations, I used the second of the two cores of the ESP32, which allows me to have one core dedicated entirely to rendering and one dedicated to interaction with the user. Currently I have used simple buttons, which change the values of numerous global variables that I use for communication between the two cores. For more information on how to independently manage the two cores of the ESP32, I recommend going to the following link: (RO.14).

https://www.youtube.com/watch?v=k_D_Qu0cgu8&t=225s <<< RO.14

PART 5: ADDITIONAL RESOURCES

Below are some additional resources that could be useful for the success of the project.

https://www.youtube.com/watch?v=e0d-N3t7ccU <<< RO.15

https://www.youtube.com/watch?v=p8Lyb-oBCDk <<< RO.16