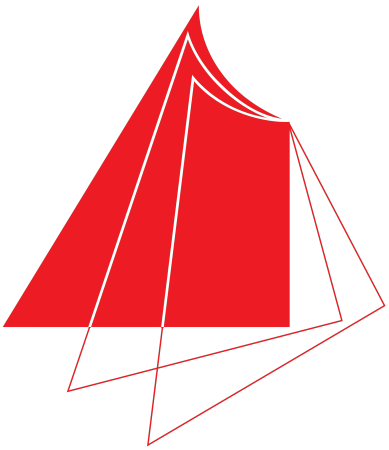
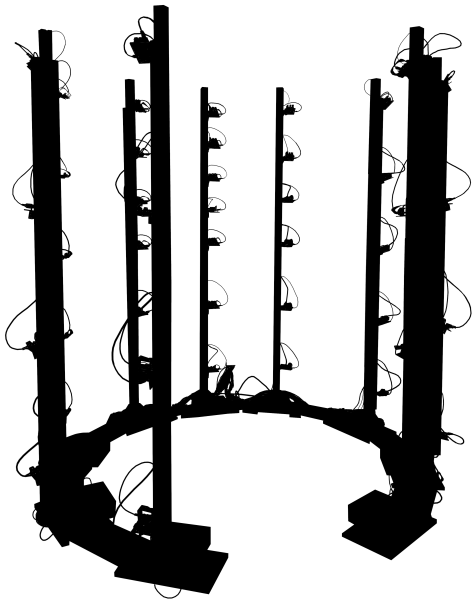


HOCHSCHULE KARLSRUHE - TECHNIK UND WIRTSCHAFT

Documentation of HSkanner3D Development

Benedikt Reinberger #62484



Contents

1 About This Document

This document covers the development of HSkanner3D, from the idea to the finished product. It can be read to understand why some of the decisions in the process were made, or what individual parts of hardware and software the project consists of. It is not listed in chronological order, but instead in a top-down way, starting with the general structure, then explaining the indiviual parts in greater detail. The images and graphics in this document are created by the author of this document, if not specified otherwise.

2 What Is HSkanner3D?

HSkanner3D is a 3D scanner. 3D scanners can scan three-dimensional shapes, gathering information about structure, proportions and sometimes color. The result is a pointcloud (as seen in Figure 1), from which a 3D model can be derived. But it is also the name of this project, which encompasses the design process of such a scanner as well as the hardware and software required. Figure 33 shows the scanning array of HSkanner3D, where the subject to be scanned stands.

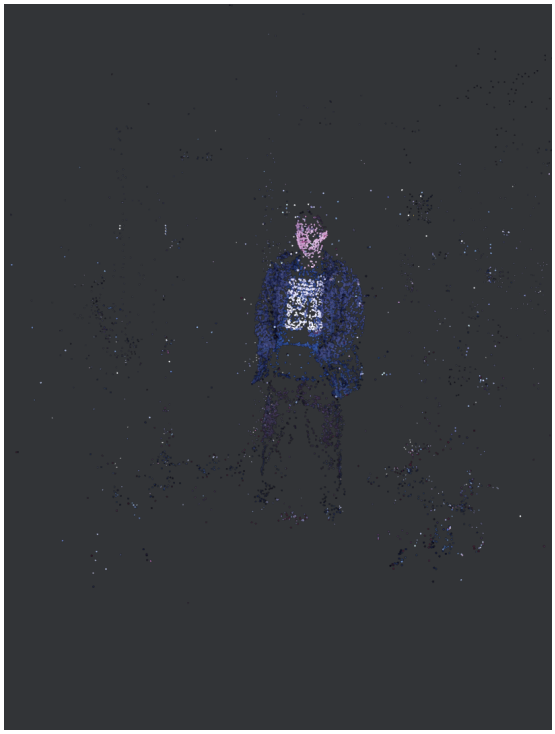


Figure 1: A coarse pointcloud of a person generated by HSkanner3D

Figure 2: The scanning array of HSkanner3D

3 Goals Of The Project

Since Hochschule Karlsruhe does currently not have a 3D scanner, it is missing out on the core features of such a system:

- Being able to recreate complex shapes in 3D space without much effort
- Creating 3D video data for use with e.g. motion tracking (not implemented, but very possible)

Using the scanner to create 3D models of visitors, the world of electrical engineering can be made more accessible to a younger audience interested in going to university. Further, the scanner has be able to be used by just about anyone without in-depth training on the subject of 3D scanning.

For some uses, purchase of a 3D scanner is the wiser choice. However, for us at Hochschule Karlsruhe, that really isn't the case, since there is a plethora of arguments for building your own scanning system over buying a finished product:

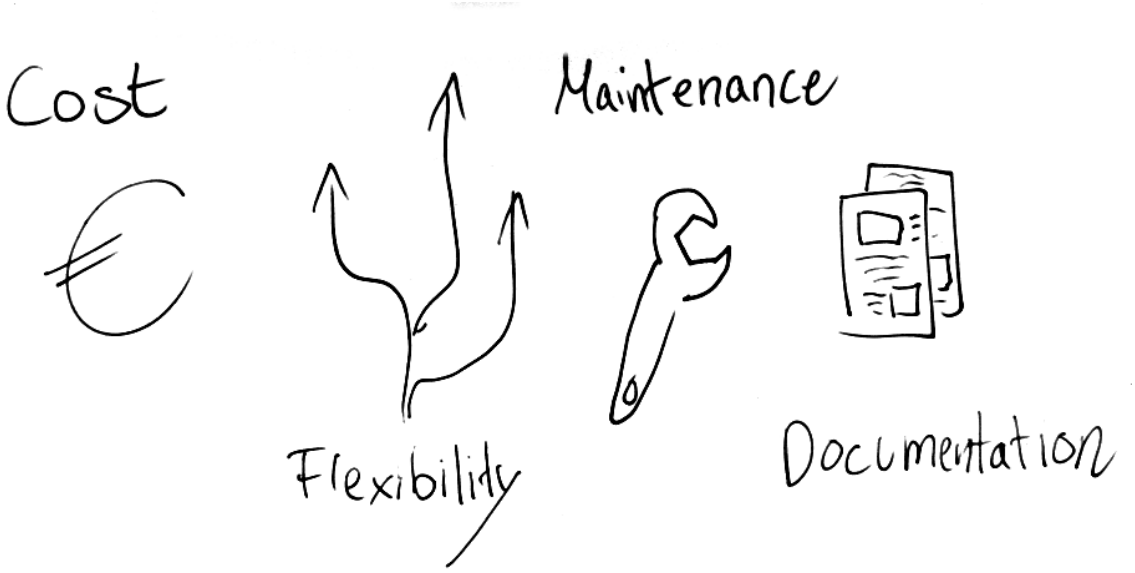


Figure 3: The four pillars of HSkanner3D design

1. Cost
   Commercial solutions like the one MakeShape [1] is offering cost large amounts of money. Their scanner uses 100 cameras, but costs around 43000€. HSkanner3D has a total project budget of 5000€ (including research & development cost, excluding mounting hardware and working cost).

2. Flexibility
   Since all parts of the system, be it hardware or software, are designed to be modular, interchangable and adjustable, the system can be altered at will to suit different needs in the future.

3. Easy maintenance
   The modularity helps make the system be more easily maintainable. Hardware and software are written and assembled with repairabilty in mind.

4. Well documented
   As the project is developed in-house, there's no secret sauce. All code and design assets are available for future improvements and enhancements by students of the Hochschule (and the world, by means of Github).

4 System Architecture

4.1 Components Of A 3D Scanner

There are multiple different ways to obtain 3D data. There is laser-based distance measuring with LIDAR and Time-of-flight systems, as well as image-based reconstruction with Feature Extraction (shown in Figure 4), Feature Matching and Structure from Motion and Structured Light (see Figure 5), among others. Our goal of scanning visitors in particular has several parameters that need to be met:

- Capture of all data around the subject at the same time to avoid artifacts from movement (even when standing still, humans move)
- Capture of color with flat light (for use as 3D model)
- Cheap scanner cost

Figure 4: RGB image taken by a Kinect 1.0 with features extraced by the SIFT algorithm

Figure 5: Infrared image with structured light pattern projected by a Kinect 1.0

The capture of color and the cheap scanner cost excludes exclusively laser-based methods, while the capture with flat lighting excludes the exclusively structured light approach. Mixing techniques is possible, especially the structured light and SfM processes, but was omitted due to cost and time constraints. We are left with SfM, which uses images captured from different angles and matches features between them to calculate depth and color information along with camera position in 3D space. A SfM-based scanner has two basic components: one (or more) camera(s), and a computer to calculate the 3D data from captured images. Since we want to capture all the data at once to avoid motion artifacts,

we need multiple cameras that all capture an image at the same time. Generally, more cameras is always better for SfM, since there is more data and bigger overlap between pictures (60% overlap and more is recommended by Meshroom documentation, whereas taking multiple images from the same position is bad). However, computational load and camera cost rises significantly with the use of more (or higher pixel count) images.

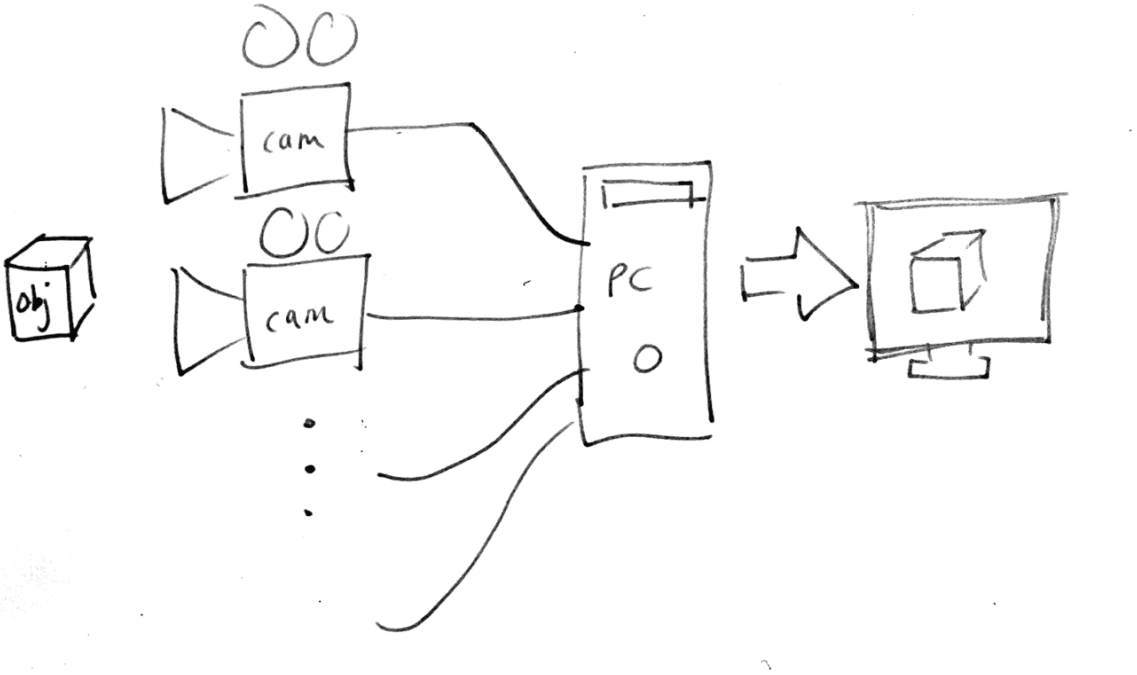


Figure 6: Schematic representation of basic scanner components

Now that we know we need many cameras and a computer to get us a 3D model, we need to know what cameras and what kind of a computer we require. There is a vast amount of digital cameras available, from DSLR/DSLM over USB (Web)cams to CCTV surveillance systems with wired (ethernet) and wireless (wifi) network interfaces. Let's go over what features the cameras need to be a good choice for our application.

- Higher resolution → more detail in image
- Cheap cost → more cameras → more overlap and detail
- Decent SNR → lower noise → more detail
- Fairly wide angle lens → more overlap → less cameras needed to cover all surfaces (but also less detail at the same image sensor resolution)
- Synchronized capture → no motion artifacts
- High reliability even with large quantities of cameras connected to the computer

USB-based solutions are possible, but not recommended - at least not connected directly to the PC. USB can struggle with many devices (we will be using about 50 cameras).

Furthermore, USB camera drivers are not designed to work with that big a number of connected cameras. Network connected cameras are a much better tool for the job - ethernet is built to support an immense amount of devices. Wireless networking is possible, but unreliable with large amounts of participants in such a dense application. Besides that, it is usually slower than its wired counterpart. CCTV cameras with ethernet connection are expensive, and commonly have bad SNR and sensor resolution. DSLR/DSLM cameras are flexible in lens choice and have good SNR due to their large sensors, yet they are expensive and rarely network-controllable outside the very expensive professional models. As a means to circumvent all the problems with USB or similar interfaces that are not built for massive numbers of clients, we can use additonal computers as an intermediary, since we can connect a small amount of USB cameras to each of those computers, then network them with our computation workhorse.

The intermediary computers should provide

- Small formfactor
- Cheap price
- Interface(s) for camera(s)
- Fast network interface for transmission of control signals and images
- Good software support and documentation

One such computer is the Raspberry Pi 4, Model B (see Figure 7), sporting a CSI interface for camera modules, two USB 3.0 and two USB 2.0 ports, Gigabit Ethernet, and a small footprint. Since the cheaper USB webcams usually don't provide good image quality or controls, the Camera Module V2.1 for the Raspberry Pi was chosen as a camera. It has a $1/4$" $8MP$ sensor, a lens with a diagonal FOV of about $80°$ and attaches to the Pi via the CSI interface. It's worth mentioning that there are splitter cables for these CSI ports, making it possible to connect more than one camera module to the Pi. Unfortunately, those do not seem very reliable from the reviews that were gathered from multiple different sources. Additionally, since they can't have long ribbon cables due to signal integrity, the attached modules can not be apart very far. Although this results in more image detail in a specific spot, it captures little depth information.

The combination of camera and intermediary computer is called sensor node (SN) within this project.

Figure 7: Raspberry Pi 4 Model B single board computer

Now that we have figured out what kind of camera we need, we need to determine what aspects a computer needs to have to be a good machine to run photogrammetry software (which will use SfM amongst other algorithms). HSkanner3D uses Meshroom to create 3D data. As to why that particular software was chosen, see Section 4.3.5. The computation workhorse should provide

- Gigabit Ethernet (almost all modern motherboards come with GbE)
- Good CPU single thread performance for meshing in Meshroom
- Good CPU multi thread performance for feature extraction and image filtering among others in Meshroom (it is possible to do this on the GPU, which is faster, but sometimes hangs Meshroom 2019.2 with the GTX 1660Ti - also more inconsistent, with alignment failing on GPU when it passes on CPU - GPU extraction can only do SIFT features, but not AKAZE)
- A GPU with CUDA (NVidia only) since Meshroom needs it for depthmap calculation (VRAM use is very mininal in Meshroom, lower than 1GB for 50 $8MP$ images)
- Depending on the resolution of the 3D model, a fitting amount of RAM - 16GB is plenty for decent quality

When using other photogrammetry software, such as Agisoft's Metashape, the hardware requirements may be different.

The computer used for 3D reconstruction is called compute node (CN) within this project. It was purpose-built for this task and features

- AMD Ryzen 5 3600 CPU
- NVidia GTX 1660 Ti 6GB GPU
- 16GB DDR4 RAM

which is a compromise between cost and performance.

To validate the capability of the camera module to be used as a camera for 3D scanning, a small development setup was built (See Figure 8). It has nine SN which are roughly positioned in a half-circle. This rig was not only used for hardware validation, but also as a testbed for software development. Notable is the use of Power over Ethernet (PoE) hats for the SN. This, combined with a PoE capable network switch, eliminates the need of additional wires for power. While this approach seems lucrative at first, it has some downfalls that make it unviable in this particular setup.

- cost of around 20€ per SN for the PoE hat
- blocking of all the GPIO pins
- need for PoE capable switch (which is always more expensive than the model without the capability)

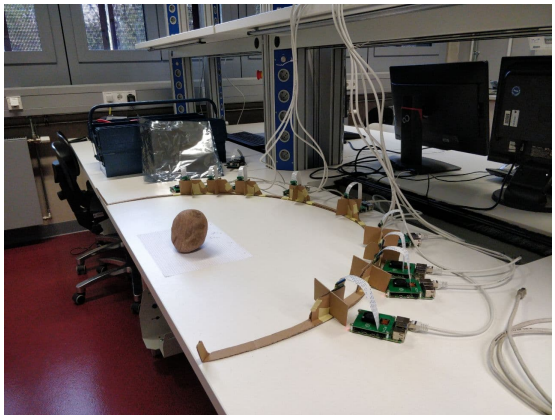


Figure 8: The development setup, side view

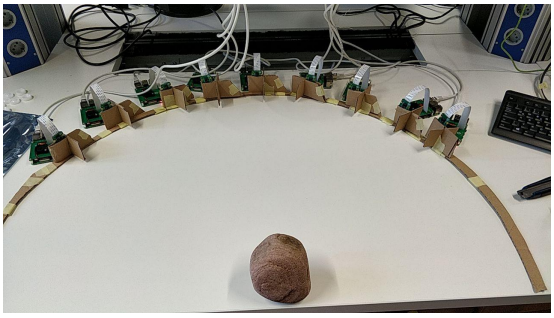


Figure 9: The development setup, front view

Figure 10 shows the setup in more detail. It consists of Raspberry Pies with Camera Modules, a GbE Switch and PC.

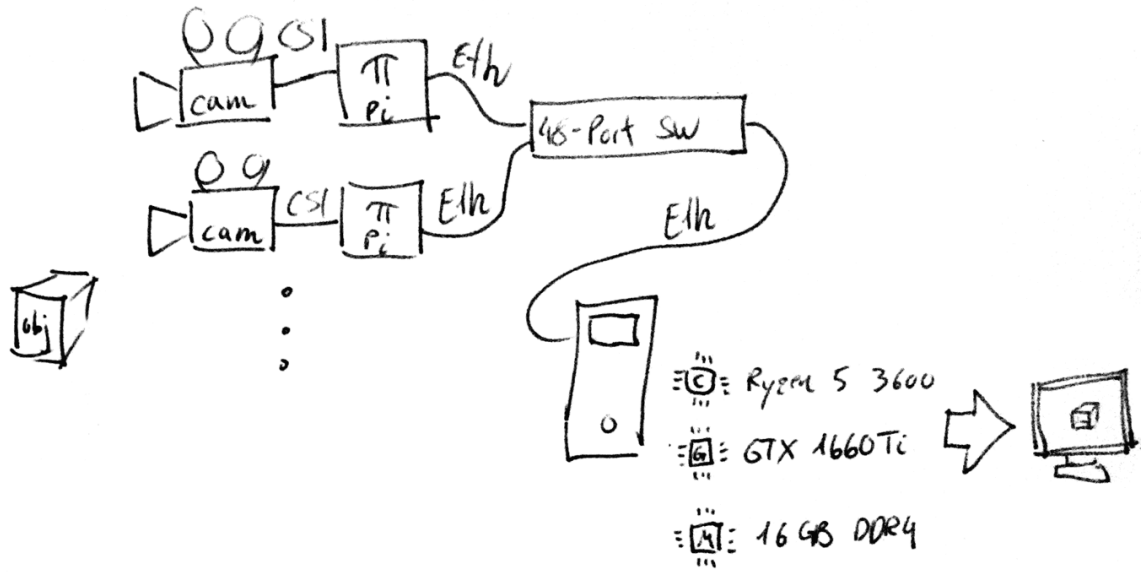


Figure 10: Schematic representation of the scanner in greater detail

The small sensor size of the camera modules as well as the requirement for flat lighting results in a need for a custom lighting setup. For its high flexibility and easy integration without additional hardware, individually addressable RGBW led strips were chosen as light source. Those can be controlled with the Raspberry Pi via the PWM capable GPIO pins (HSkanner3D uses GPIO pin 18). An additional advantage of these strips is the voltage requirement of $5V$ DC, which is the same as the Raspberries and allows for the use of a shared $5V$ only power supply. Since the lighting is controlled by some of the Pies (there are only nine $2m$ led strips in the final design, so not all Raspies are connected to a strip), those can also be referred to as lighting nodes (LN). Figure 11 shows this setup schematically.

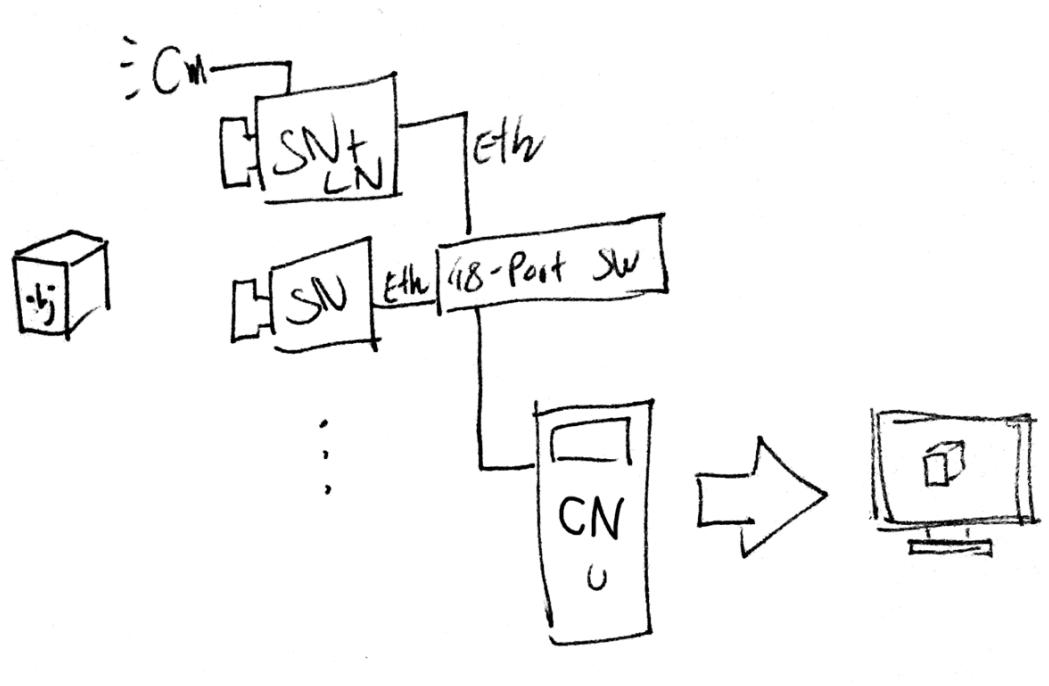


Figure 11: Schematic of the scanner with added lighting and component naming

4.2 Structure Of The Physical Scanner

4.2.1 Simulation With Blender And Meshroom

NOTE All the project files from this section can be found within the `design` folder of the repo.

Now that the basic components of the scanner are figured out, it is time to assess where and how to position and rotate the cameras and lighting sources in 3D space. The budget allowed for 45 SN, so the arrangement was chosen to get the most out of that number of cameras. To simulate the scanner, the 3D software Blender was used. A 3D model of a human was created with the MB Lab plugin which was chosen for its capability to create a realistic represenation of the setup without much knowledge in 3D modelling. The model was then dressed using the demo assets (which are less realistic) and posed using default poses from the plugin. The size of this model was kept close to $1.8m$, since that is the maximum average height over all ages for german men [2], who should be able to be scanned by HSkanner3D. Cameras and lights were positioned around the subject, the field of vision of the cameras was matched to the Raspberry Pi cameras (see Figure 12). The upwards and downwards angle of each camera was tuned so that each camera saw the most possible of the model. Then, each camera was rendered at $8MP$ (this took a long time for each run, even though light bounces were minimized and the fast OptiX API was used with an RTX 2070). The resulting images were imported into Meshroom, which then recalculated a Mesh (see Figure 13). This process had to be repeated for each camera and lighting configuration validation.

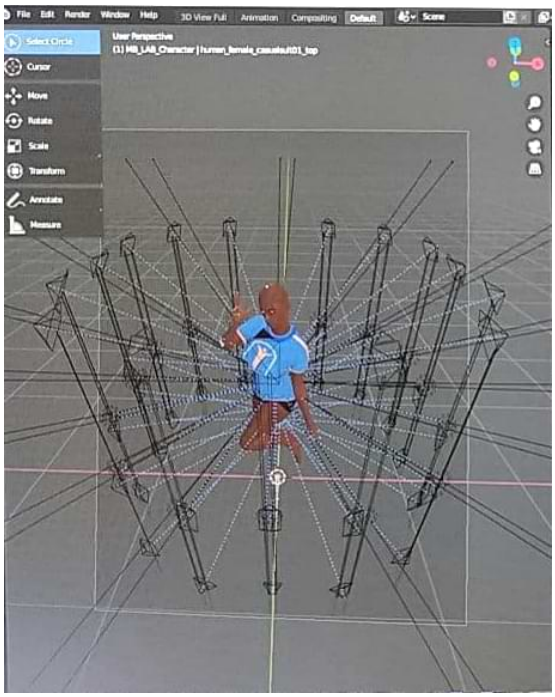


Figure 12: Blender simulation with 15 towers (3 cameras each)

Figure 13: Reconstruction of data using Meshroom

To capture data from all angles, it would be ideal for the cameras to be aligned on a sphere around the center of the subject. However, since the scanning array should fit within a $2m$ x $2m$ x $2.5m$ (LxWxH) space, and be easily and cheaply manufactured, a cylindric arrangement was chosen. This also makes it easier to create an exit and entry point to the array without much hassle. There are many ways to spread the cameras around the surface of a cylinder. In principle, there are two axes to consider: the vertical axis and horizontal (along the cylinder wall) axis (see Figure 14).

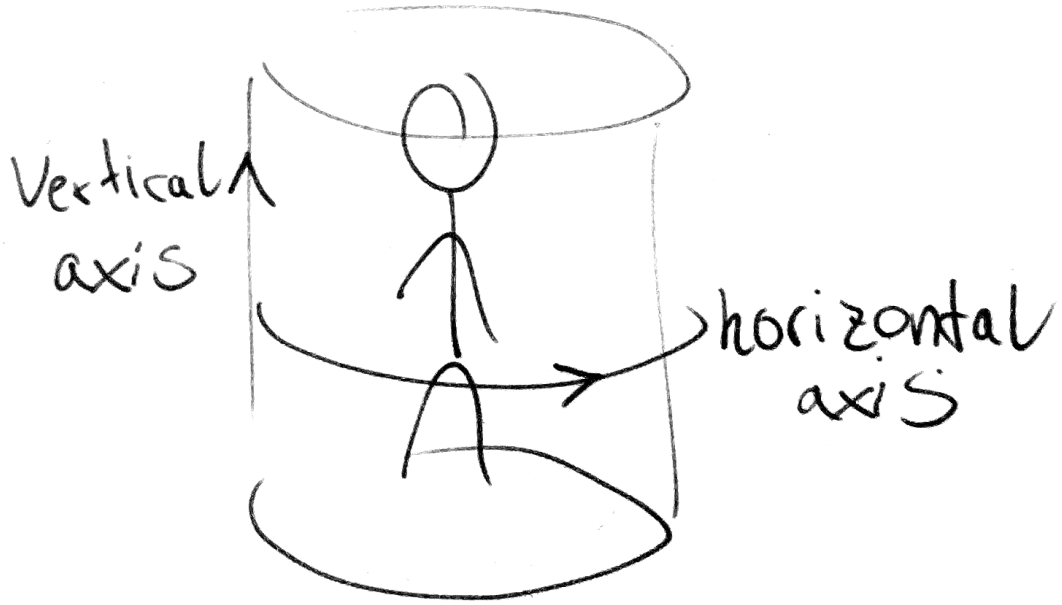


Figure 14: The axes of the scanning array cylinder

The photogrammetry software needs significant overlap (above 60% is a good value) between neighboring pictures. That means that the cameras can't be spread too thin in either axis. If the cameras are mounted on what is effectively a pole, getting a higher vertical camera density is trivial - just adding more cameras along the length of the pole will suffice. Yet to get the cameras closer in the horizontal axis, more poles have to be set up, which increases cost and manufacture time. Thus, the minimum amount of poles one could get away with had to be determined. After five major iterations, a good middle ground for camera density in both axes was found. The design uses nine poles, so-called "towers", with mixed camera density. A higher amount of cameras was positioned in the facial area, since capturing detail there was considered most important. Around back is the lowest camera density as well as the array entry and exit. The resulting camera arrangement is shown in Figure 15. The left side presents the placement of the towers along the horizontal axis, the right side depicts the camera arrangement on the vertical axis. Most towers are spread apart $40°$ for even distribution around the circumference, since $\frac{360°}{9towers} = \frac{40°}{tower}$. T2 and T9 are closer to T1 to boost camera concentration around the face in the horizontal axis by using only $30°$ spacing. This also opens up the back of the array for an exit between T5 and T6. To ensure the high camera density does not go to waste, the person standing in the focal point f should face towards tower 1 (T1).

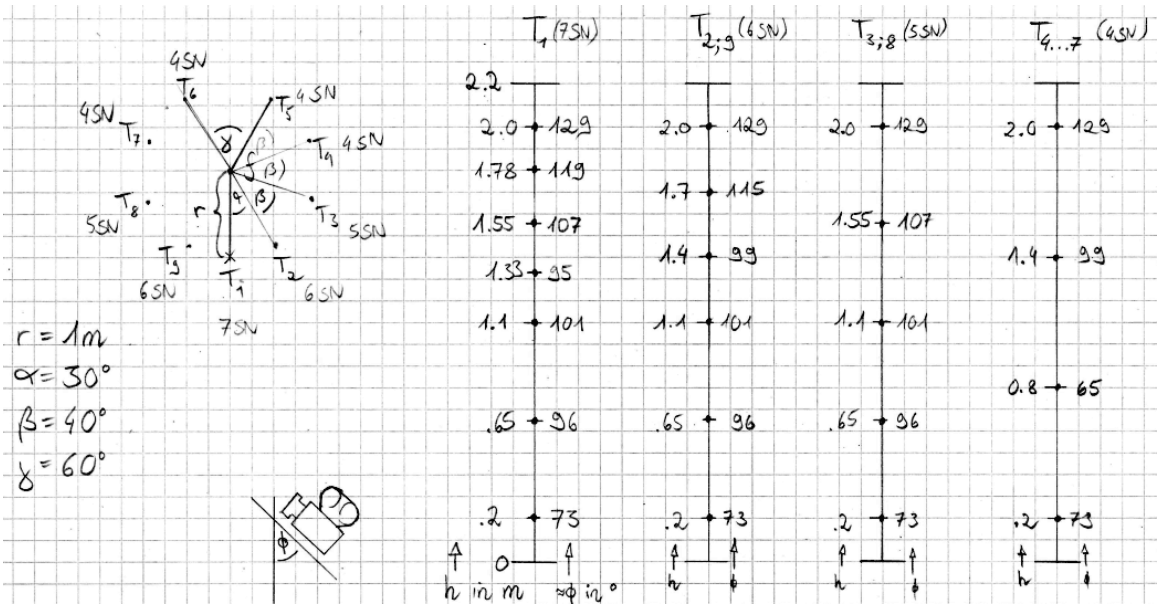


Figure 15: Camera and tower position layout and orientation angles

4.2.2 Part Design

At this point, we know where to put the cameras - and the use of LED stripes along the tower length as diffuse light source was validated as well by using LED-stripe-sized area lights in Blender. Therefore, some contraption had to be made to allow for the cameras to be mounted at their respective height and angle (see Figure 16).

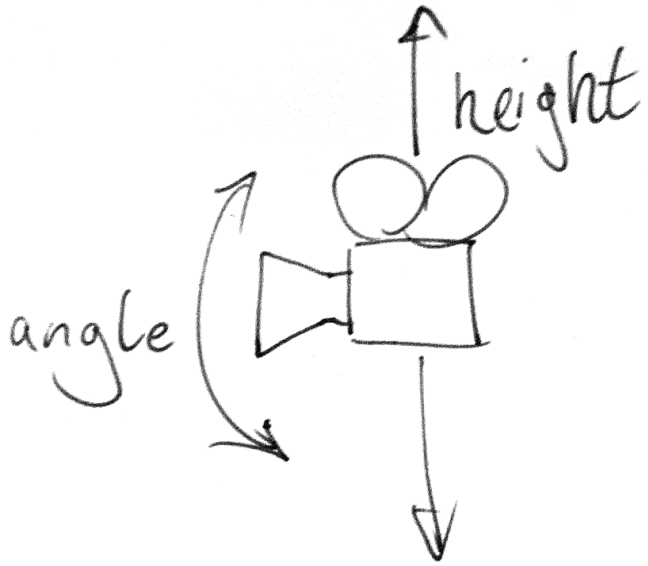


Figure 16: The two degrees of freedom that the camera mounts need to have

First Concept The first construction idea was wood-based and had a lot of parts. Figure 17 displays a full tower based on this idea with one of the walls removed.

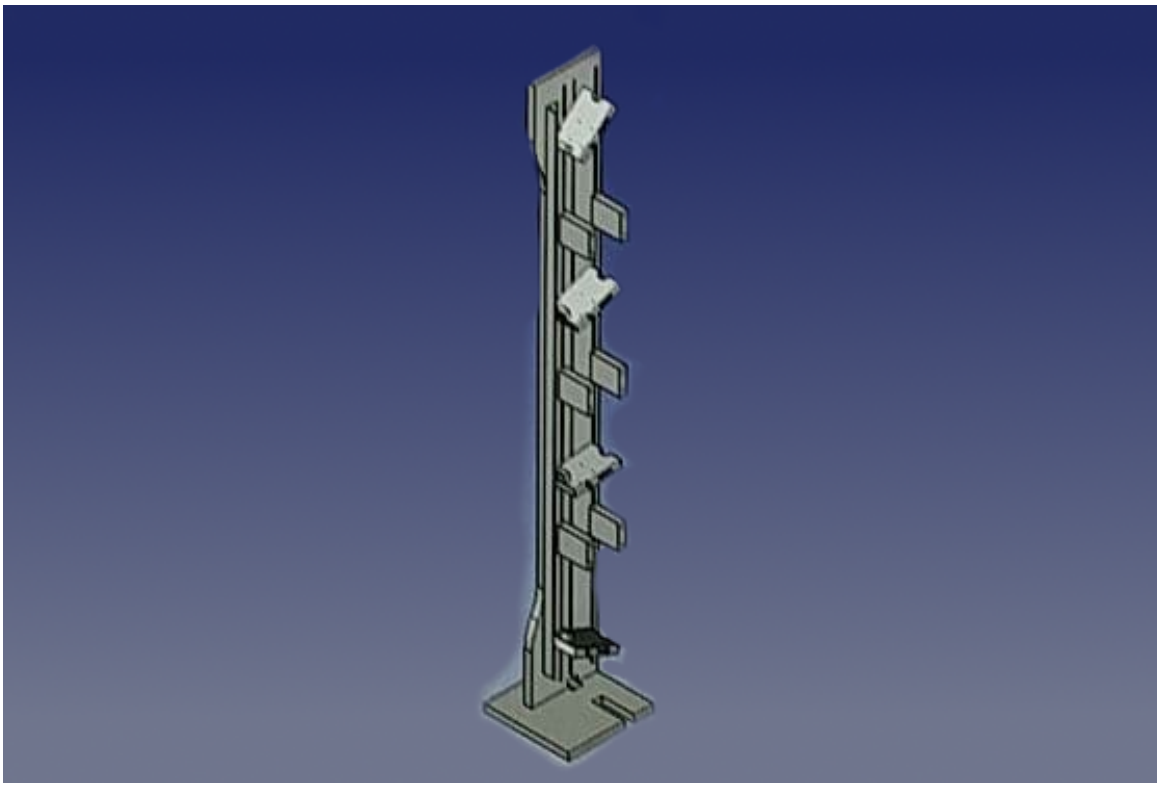


Figure 17: The first concept: a wooden tower

In between the two walls, there were carriers for SN as well as spacers for stability. All of those rode in long slots cut into the walls. This results in extremely high flexibility in use. However, it has severe downfalls that make it useless:

- slots that span two meters in length are incredibly hard and costly to manufacure
- wood is more expensive than the solution of design two
- the tolerances of the spacers had to be relatively tight, making them more expensive
- the SN were very inaccessible in the center of the tower

The first concept was thus canned and a new and improved design was created keeping in mind the key concepts along with the recently learned principle ”keep it as simple as possible”.

Second Concept - Baseplate First, the base was conceived. Since the towers will be aligned in a circle and bases can be joined together, the base for single tower can be pretty small without the hazard of the construction tipping over. The baseplate is made out of PVC and is presented in Figure 18. The center of the plate features six holes to bolt down angled brackets, the right and left side have two holes each to fit connecting plates later.

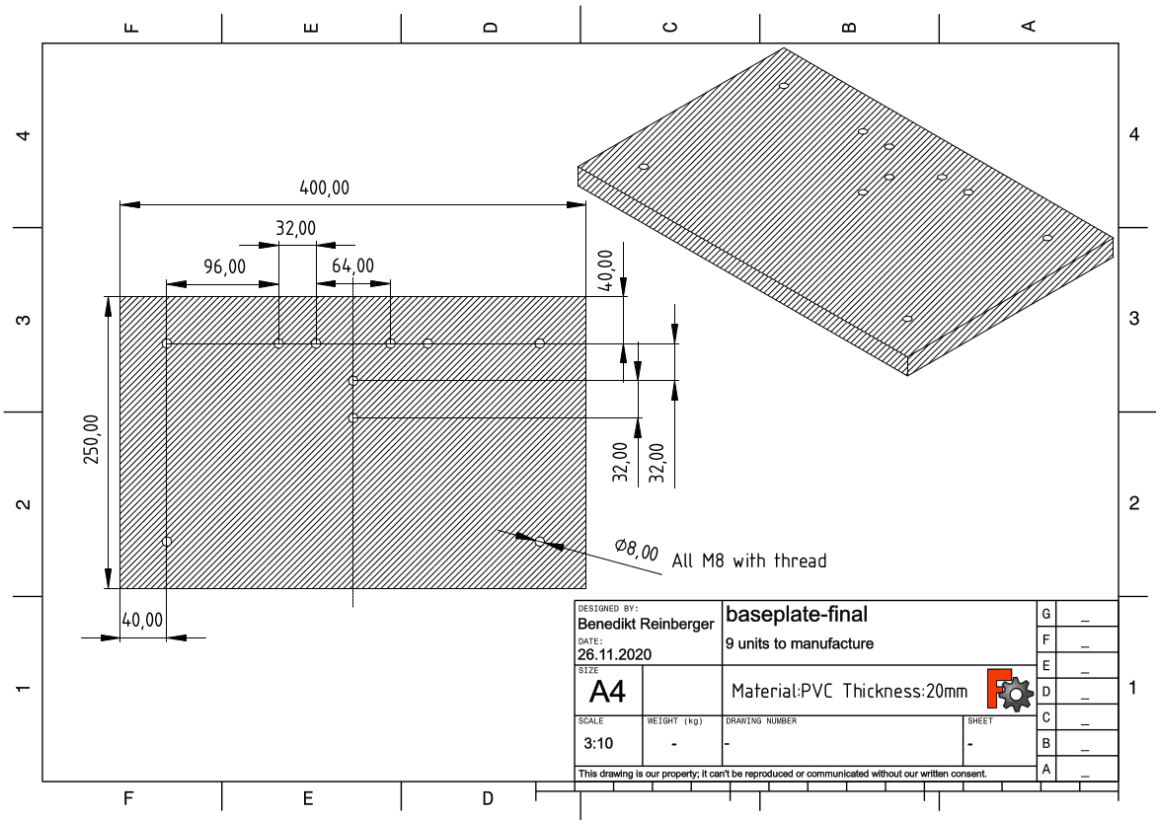


Figure 18: The baseplate from a tower from the second major hardware concept

Second Concept - Aluminium Profile And Angles This baseplate holds an aluminium profile of $2.2m$ length. It is fastened to the base using angled brackets and the center holes. Both parts can be purchased off the shelf and are readily available. The assembly of baseplate, profile and three angles is called

`asm-tower-0c.FCStd`

(assembly-tower-zero-cameras) and displayed in Figure 19.

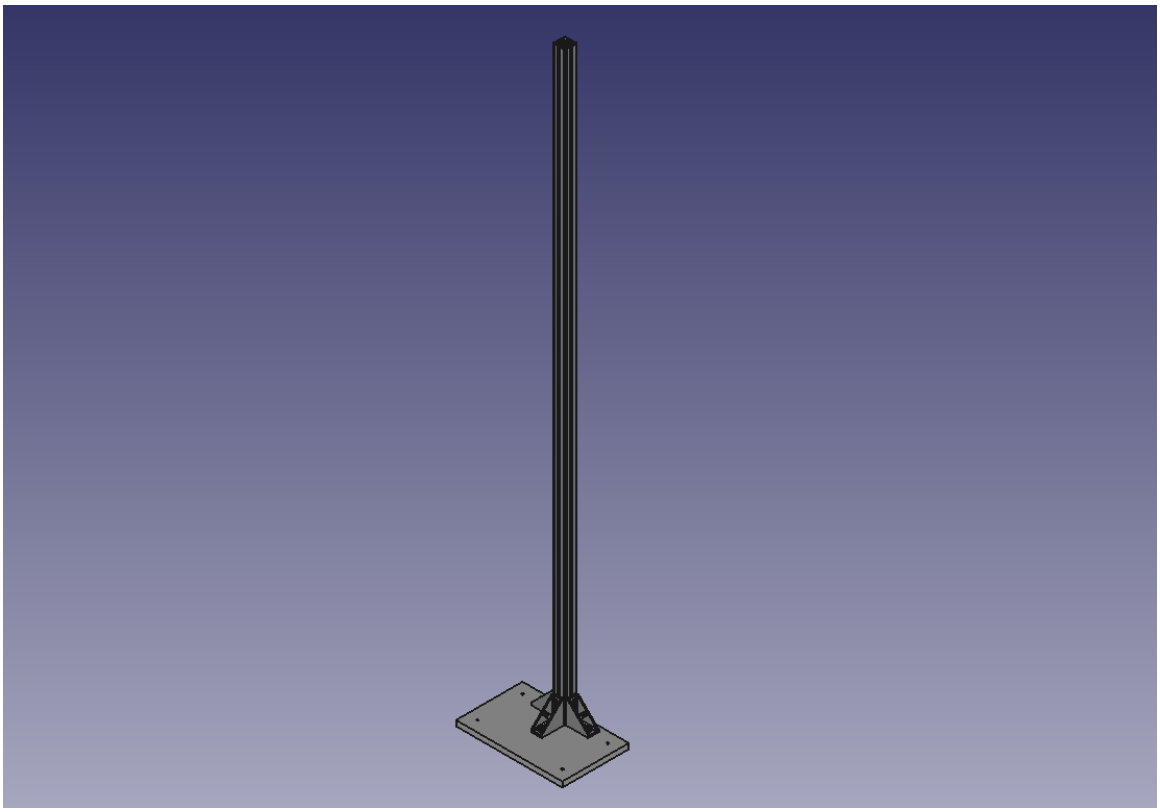


Figure 19: The tower without anything attached

NOTE The center holes in the baseplate are offset wrongly - each pair of two should be $64mm$ from the common center, not $32mm$. This design error was not found in prototyping (one tower was built as a prototype before the rest of parts was ordered), since the hole distance was corrected only in prototype production, but never communicated back. The final run was manufactured to specification (which had the wrong hole distance still). Therefore the the profiles are being secured using the outer holes only - by three, not six bolts.

The profile has high flexibility in its use, since it has four rails that can be used to mount hardware. One is used for SN carriers, another for a cable management duct. Being made from aluminium, heat conductivity of the profile is generally good. This makes the inwards-facing (north) side of it the ideal place to adhere the lighting strips to, as it doubles as a heatsink. The rail next to the LED strips is occupied by their power wiring, the remaining (east) rail is free for future use.

NOTE If there is need to open the Freecad project files (.FCStd), use Freecad 0.19 with the Sheetmetal and Assembly 4 workbenches installed and open them in order:

1. tower-parts.FCStd (contains all individual parts but the connecting plates)
2. asm-tower-0c.FCStd

3. any of

    - asm-tower-4c.FCStd
    - asm-tower-5c.FCStd
    - asm-tower-6c.FCStd
    - asm-tower-7c.FCStd
    - asm-array.FCStd (will open all the tower assemblies with cameras on them)

This is neccessary due to the way Freecad 0.19 handles dependencies between project files, which is inconsistent.

Second Concept - SN Carrier Module Next, we need a way to mount the SN on any height in the profile, at arbitrary angles. The part that does this duty is called SN carrier. It needs have the following perks:

- cheap and fast to manufacure
- good accessibility of Raspberry Pi and camera module
- easy to adjust and maintain

The shape that fulfills these requirements is revealed in Figure 20.

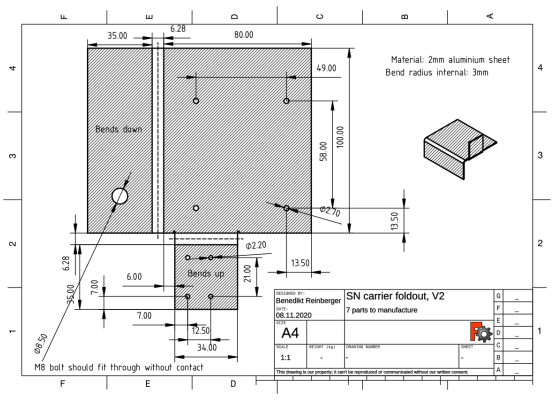


Figure 20: The final design of the SN-carrier after many small iterations

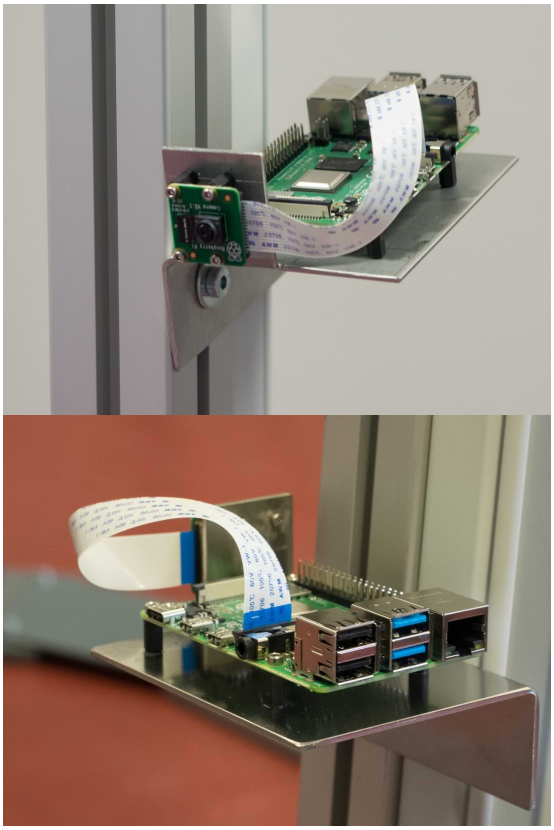


Figure 21: Raspberry Pi and camera module mounted on the carrier

For ease of manufacture, a sheet metal construction was chosen. It also fits nicely with the metal theme of the second concept. The outline of the shape and the holes can be (laser-)cut from a larger 2mm sheet. Since the holes for mounting Pi and camera are through-holes, no additional tooling is needed for them. To archive the required geometry, only two 90° bends are needed, and tolerances on those can be quite lenient - since the carrier only has one contact surface with the profile. The hole to mount the carrier on the profile is oversized as well to allow for fine angle and height adjustment without loosening the bolt. The small part that bends up holds the camera module, the large flat area holds a Raspberry Pi. Both use plastic spacers to avoid contact of components on their PCBs with the surface of the sheet metal below (see Figure 21). The carrier is laid out in a way that makes access to the I/O of the Pi easier, since the ports face outwards or upwards, and not towards the profile. The camera takes photos in the portrait orientation to capture more data of the subject, which is more vertical in nature than it is horizontal.

Second Concept - Baseplate Connection Plates Finally, to connect the bases together for much needed stability, connecting plates are needed. There are two types of connecting plates: the ones for the 30° spacing and for the 40° one. The towers were placed in Freecad according to Figure 15, then the hole distance from center to center for each connecting plate was measured. Due to the geometry of this array, the spacing is very specific. To combat this issue, the holes were oversized. This allows for wiggle room both in manufacture and assembly. The resulting parts can be observed in Figures 22 and 23.

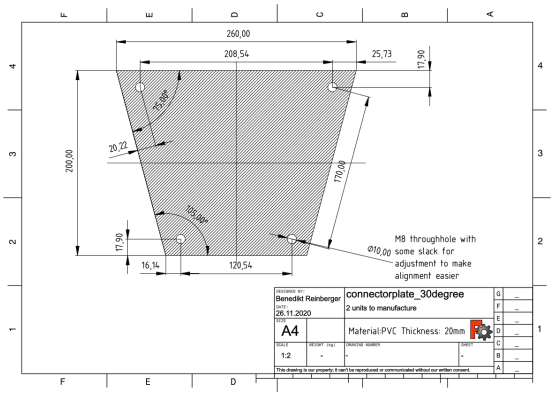


Figure 22: Connector plate for 30° spacing

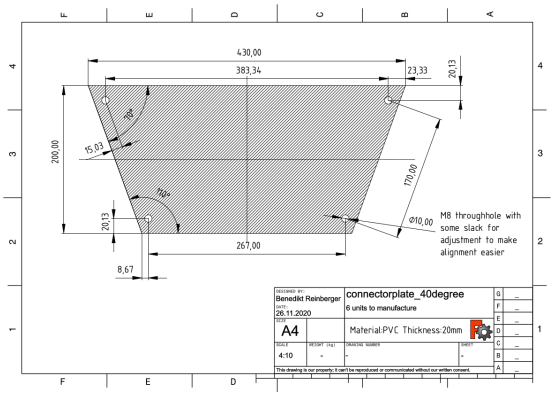


Figure 23: Connector plate for 40° spacing

Second Concept - Array Assembly At this point, we know most of the individual parts, but not what they look like when put together. The array structure is shown in Figure 24.

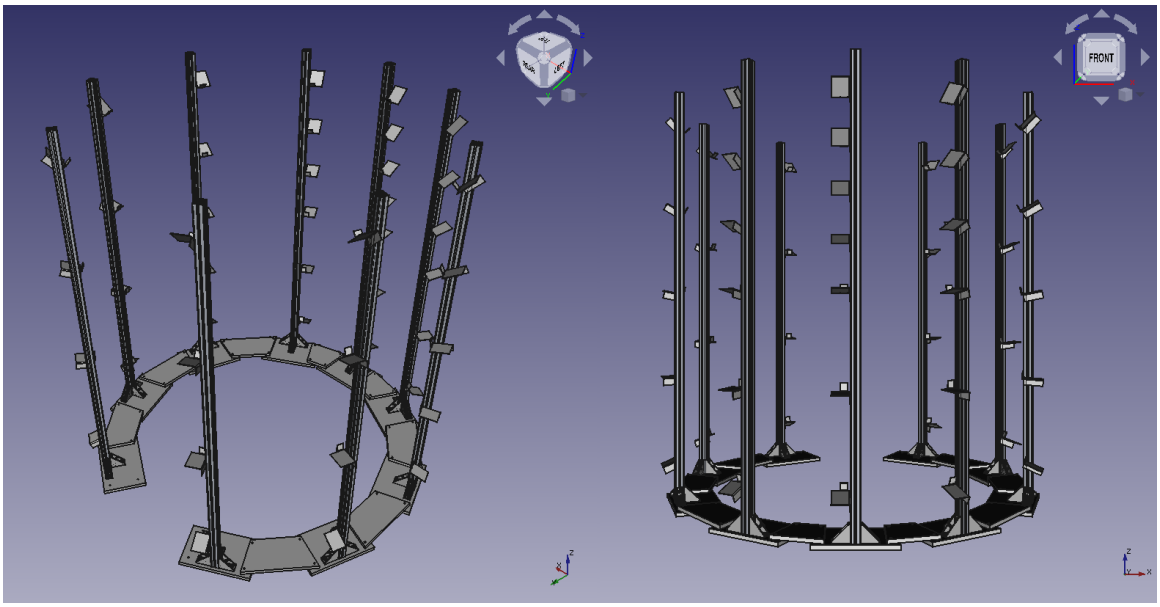


Figure 24: The array with most components installed

Second Concept - Lighting And Power Although these pieces make up most of the scanner, there are still important parts missing. For one, the LED strips, which have to be cut down into multiple $2m$ parts (one for each tower). In addition to that, both the SN and the LEDs require $5V$ DC. To reduce power loss over wires, higher voltage is better (since less current has to flow to deliver the same amount of power). Thus, each tower gets its own $5V$ power supply that is fed with mains voltage. On each tower baseplate, there is a power distribution box that contains the power supply (upper half), terminal blocks for power distribution (lower half) and a passthrough for mains voltage. This passthrough minimizes cable clutter. The first prototype of the central power distribution box is shown in Figures 25 and 26.

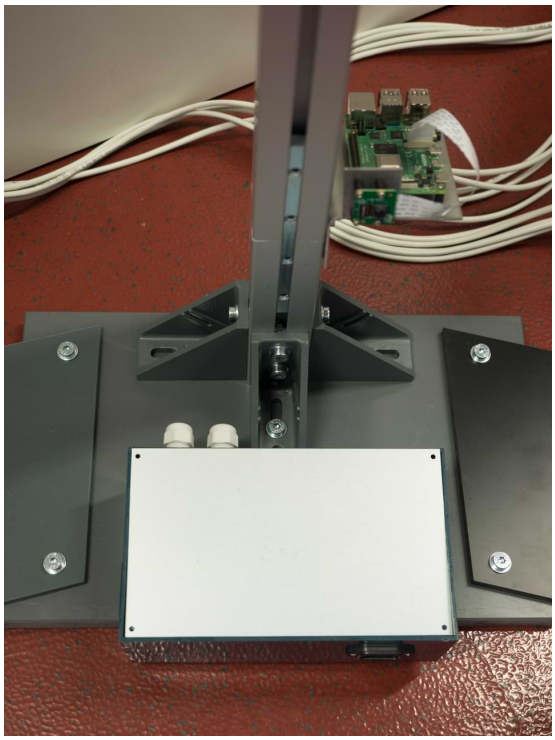


Figure 25: The T-junction power distribution box (PDB) at the tower baseplate

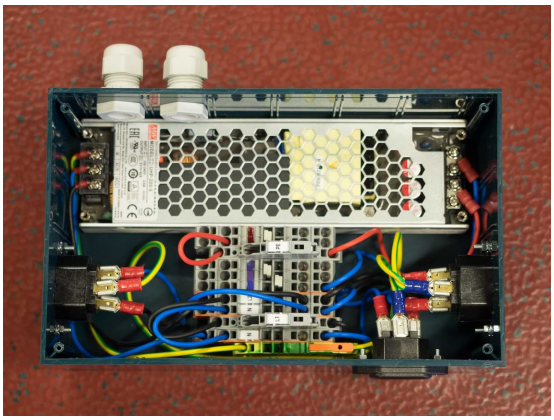


Figure 26: The insides of the T-junction PDB

The power budget for each tower includes:

$$
\begin{aligned}
P\_SN &= 15\frac{W}{SN} \cdot 10SN && = 150W \\
P\_LED &= 0.3\frac{W}{LED} \cdot 60\frac{LED}{m} \cdot 2m && = 36W \\
P\_TOT &= P\_SN + P\_LED && = 186W
\end{aligned}
$$

The power of $15W$ for each SN is the value for a synthetic 100% CPU load. Idle power consumption is around $3W$ for the Pi 4 Model B. The 10 SN is an estimate for how many SN one might want to put on one side of the tower. If significantly more SN are put on a single tower, it might be better to put an ethernet switch at the base of each tower - sacrificing bandwith for less cable clutter. Note that with the current implementation of image transfer via the network, which uses Compound Pi and is sequential, a switch would NOT sacrifice bandwidth, but a better implementation or a different use case might be bottlenecked (see Figures 27 and 28). Of course, one can obtain switches with at least two high speed interfaces to compensate. However, the cost seems pretty high considering a single switch can do the same but with more cables. Therefore, a switch at the base of each tower was not chosen, at the expense of pretty big cable overhead. The $300mW$ per LED is the power consumption figure for maximum brightness. Since the power supply that was chosen is rated $40A@5V$ continuously, it can supply $200W$ and thus an additional $42W$ of headroom to the already very generous power budget.

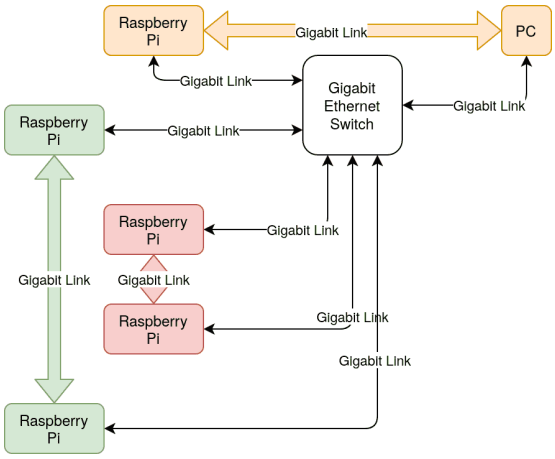


Figure 27: Setup with a single gigabit switch. Every pair of connections has gigabit speed

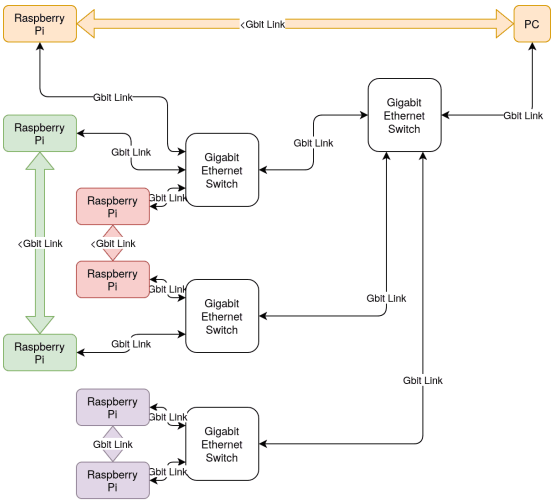


Figure 28: Setup with multiple single gigabit switches. Every pair of connections has gigabit speed only if it is connected to the same switch. Otherwise it has to share bandwidth

Second Concept - Network Wiring But the scanner still has to be wired up. For the networking, Cat5e S/FTP cables were chosen. Those support gigabit speeds - further, each cable is shielded to avoid crosstalk between different cables (but not between pairs). This might be neccessary since the cables will are bundled together tight for cable management. Velcro straps are being used, since these make it trivially easy to add more cables to a strand later. Besides that, they don't grip as tight as zip ties, reducing possibilities for cable damage. Figures 29 and 30 show bundles of ethernet cables, managed with velco straps. All ethernet cables start in either a SN or CN and end in a network switch. The networking cables from all the individual towers come together at T1 to make a giant bundle of 45 cables.

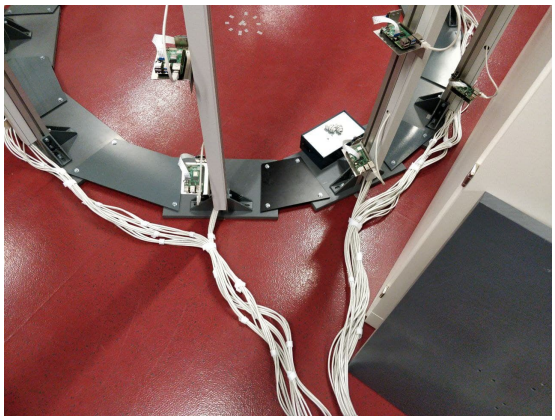


Figure 29: The cables at the array

Figure 30: The cables at the switch

Second Concept - 5V Power Wiring The cross-sectional area of the $5V$ power cables is $1mm^2$. In theory, this allows a bit over $2A$ of current to be pulled from a SN with 2m cable length while dropping less than 3% the voltage, as recommended. The $230V$ jumper cables in between PDBs can support up to $10A$. For the termination, the order of wires is as follows: Left to right, up to down. The LED lighting is always wired first, then the cameras are wired bottom to top (C1 at bottom, C9 at top). Figures 31 and 32 show a straight through PDB with and without wiring order.

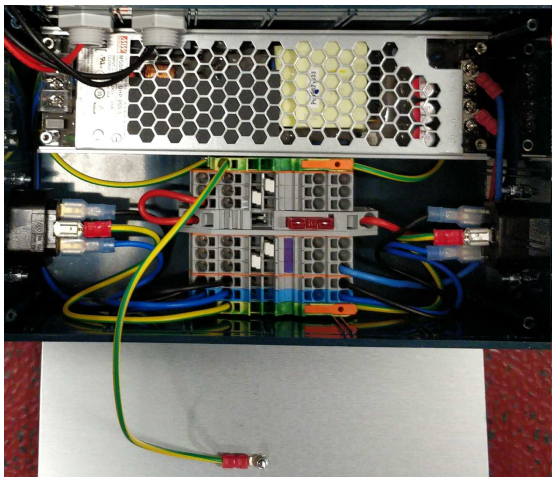


Figure 31: A PDB without wires terminated

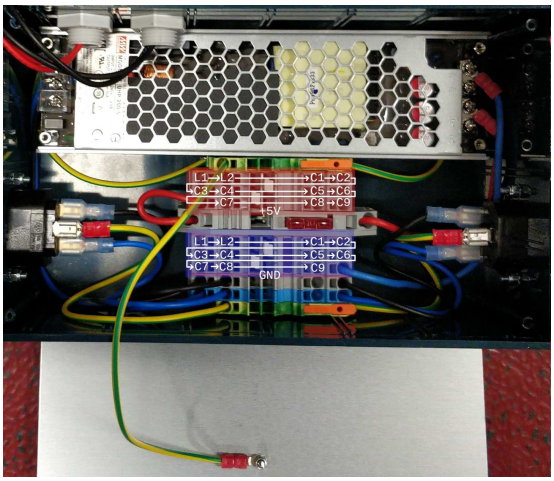


Figure 32: The order in which the power wires are to be terminated

For the sake of everyone involved with this project's power wiring, please follow the

same pattern. This means that if you insert a camera between C2 and C3, move the wiring of C3 and higher existing cameras up in the chain, starting with the topmost one. Both network and $5V$ power are routed through cable management channels that run along the back of the profile. This profile is of the closed type. While it looks good, it is also highly inflexible compared to a more open type with 'fingers' and a lid:

- the connector at the end has to fit through the drilled hole, rather than just the cable (part of the ethernet cable retainer had to be cut off to fit through)
- holes have to be drilled into it to route cables through
- hole drilling makes a giant mess of plastic 'spaghetti' and small parts that get everywhere
- holes can't be drilled if the cable management channel is full with cables
- if a SN is moved, the holes don't move - since the cable lengths are fixed, new holes have to be drilled

Thus, if there are plans to add any more SN to the towers, please change the cable management channels to the 'finger' type. The closed type is only used since a communication error resulted in them being ordered, and time constraints made it unfeasable to order and install replacements of the better type.

4.2.3 Creation And Assembly Of Parts

The base- and connecting plates, the electrical distribution boxes as well as seven of the carriers were manufactured locally. The 38 remaining carriers were outsourced to a laser cutting facility. The fastening and spacer hardware along with the aluminum profiles and brackets are off-the-shelf parts. Figure 33 shows the assembled scanning array.

Figure 33: The scanning array of HSkanner3D

For the assembly, the bases were laid out into their respective positions. Then, the profiles were attached to their bases. Afterwards, the bases were connected to each other using the connection plates. With this 'skeleton' in place, the carrier modules were outfitted with the SN, then slid into the west rail (north faces inwards to the subject). Further, cable management channels were installed in the south slot of each tower. For the north side, LED stips were cut down to nine $2m$ strips, outfit with two power connections as well as the 'data in' wire, then stuck to it using the supplied adhesive tape. With everything attached to the towers, it was time to wire everything up. Holes were drilled into the cable management channels in between the SN positions. The ethernet cables were cut down at one of their ends to fit the holes, then plugged in and packed into bundles. For power, USB

type C 2.0 cables were utilized. Those are cut up, then the positive lead is connected to the $5V$ wire (pink, because red was sold out) and the negative lead is connected to the GND wire (black, whereas the USB data wires are joined together (see Figure 34), since most USB power supplies are shorting them as well. The power wires end in clamp terminals in the power distribution box. Figures 34 and 36 show some of the wiring. Figure 35 shows the voltage at the GPIO of the topmost SN (longest wires) running the LED strip. It is quite a bit lower than the $5.04V$ the power supply puts out, most likely due to the use of the USB cable for about $40cm$, which has very thin power wires. Adding to that are contact resistances in the clamps, USB connector and resistance in the solder joints.

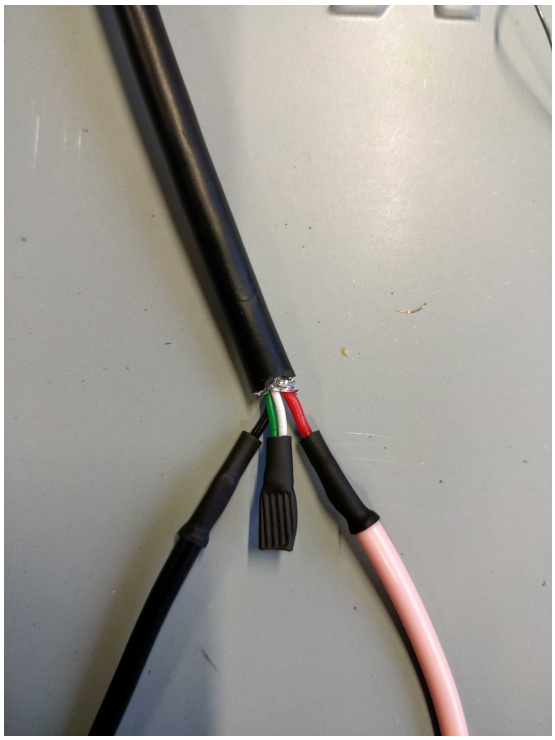


Figure 34: USB cable wired up

Figure 35: The voltage at the topmost SN under HSkanner3D idle conditions - quite low

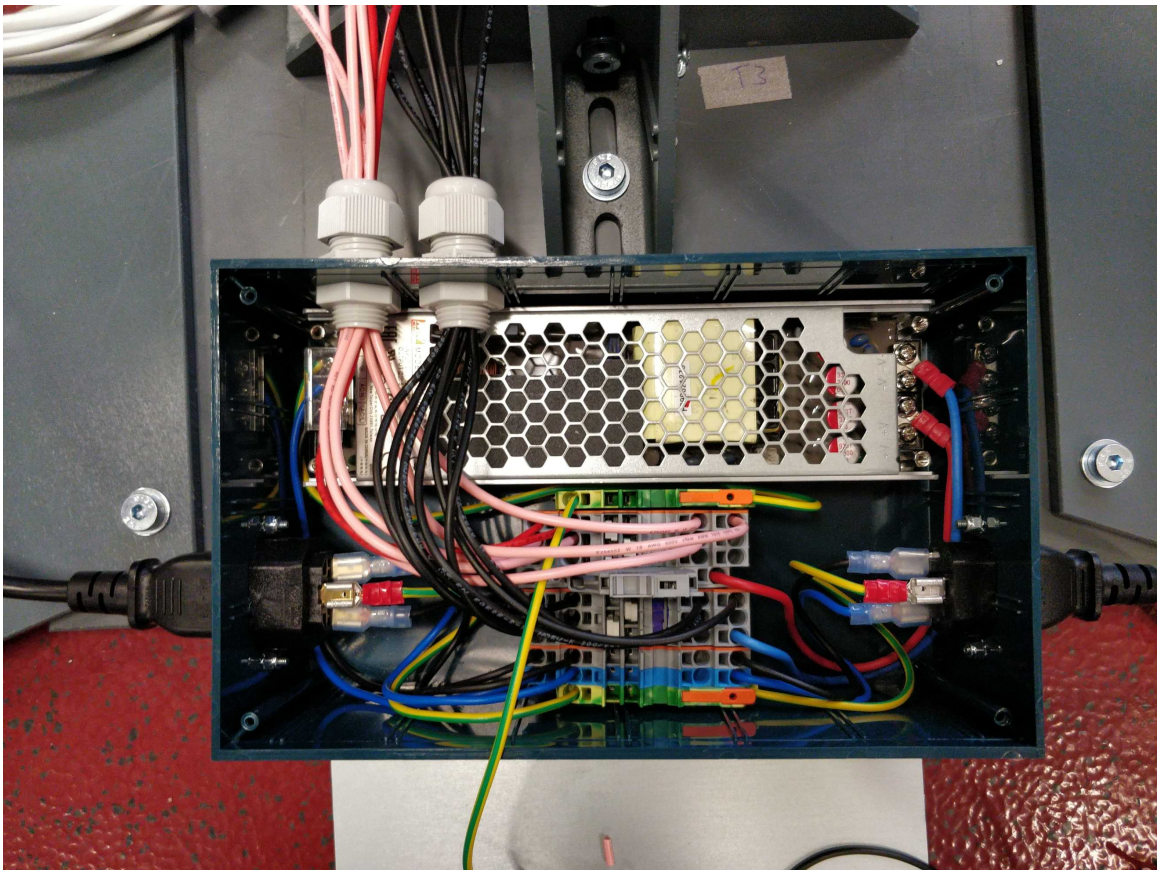


Figure 36: PDB of a tower with five cameras

To summarize, over $100m$ of each network and power leads was put down during the project. Figures 37 to 39 show some details of the finished scanner.

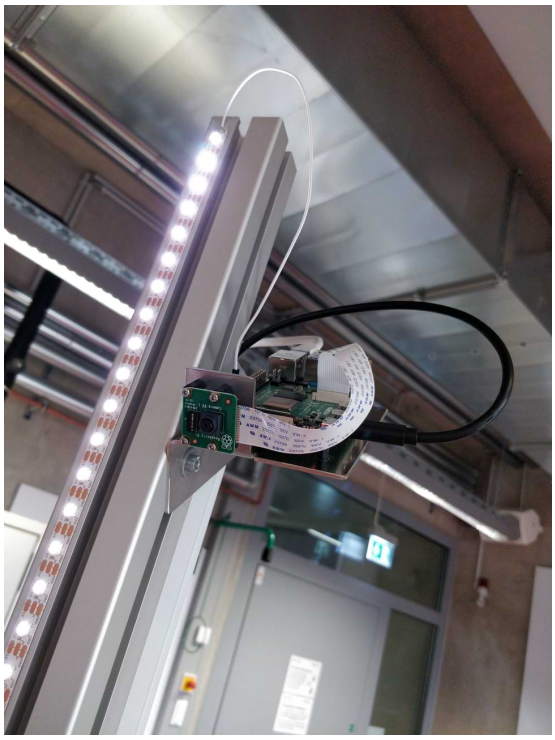


Figure 37: The topmost camera of T5 - topmost cameras control the LED strips

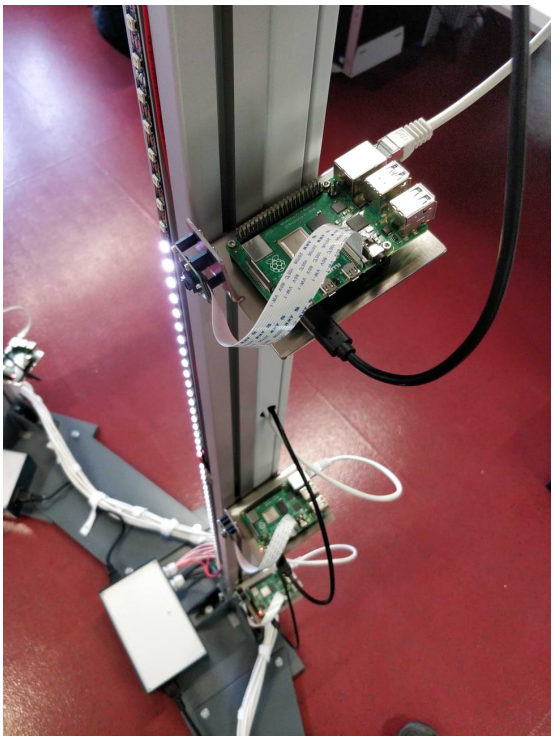


Figure 38: The bottom three cameras of T7

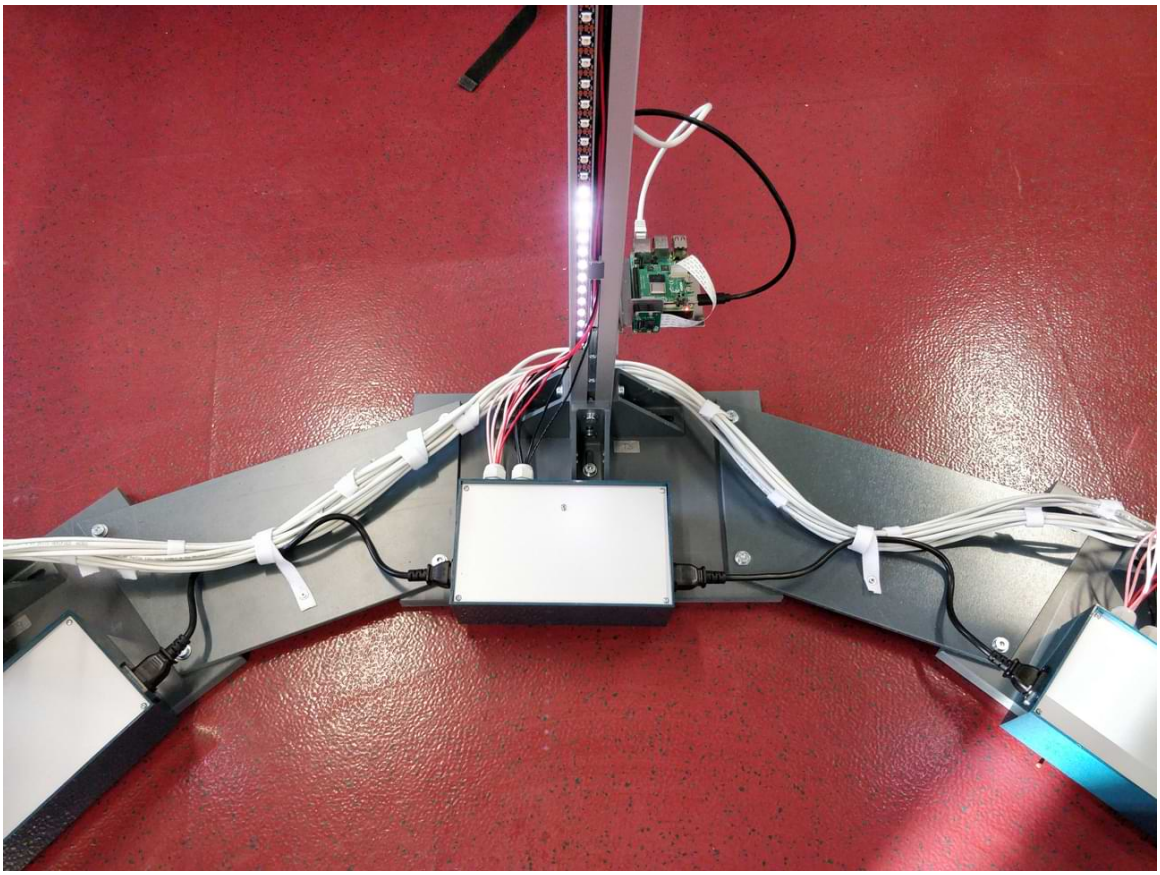


Figure 39: The lowest camera of T7, a PDB and cable management

4.2.4 (Dis-)Assembly For Moving The Scanner

HSkanner3D was designed in a way that makes it possible (although somewhat cumbersome) to be moved to a different location. First, to disassemble the scanner, make sure the system does NOT have power. Then, remove the connecting plates, being careful not to tip the towers over. After a tower has been seperated from the array, the networking cables can be freed and the $230V$ jumper cable can be removed. It is not recommended removing the networking cables from the tower side. Instead, disconnect them from the switch side, then wrap the bundle around the tower base for transport. When a tower is disconnected completely, it can be (carefully!) tipped over and carried by two people.

For reassembly, first lay out the positions of the towers according to figure 15, wherein the position of the towers is the center of the aluminium profile (for reference: the short side of the plate ends $40mm$ behind the tower center). Then, position the towers and lay down the connecting plates, not completely fastening the screws. Carefully nudge the towers around until the holes line up well enough, then fasten the bolts.

4.2.5 Ethernet Switch Modification

In addition to the construction of the array, the fans in the ethernet switch had to be swapped due to their high operation noise (it sounded like a whole server room all on

its own). Even though the replacement fans have sigificantly less throughput and pressure, they do the job just fine, since there is little resistance to airflow inside the case of the switch. Further, there is not much cooling to do, since PoE is not in use. The temperatures were monitored under load conditions (scanner operation) and never exceeded $34°C$ at about $22°C$ ambient. Note that the fan error LED will light on this particular model (Dell PowerConnect 6248P), since the tachometer wire of the replacement fans was not connected. Figure 40 shows the switch without the top cover, with the fans already replaced.

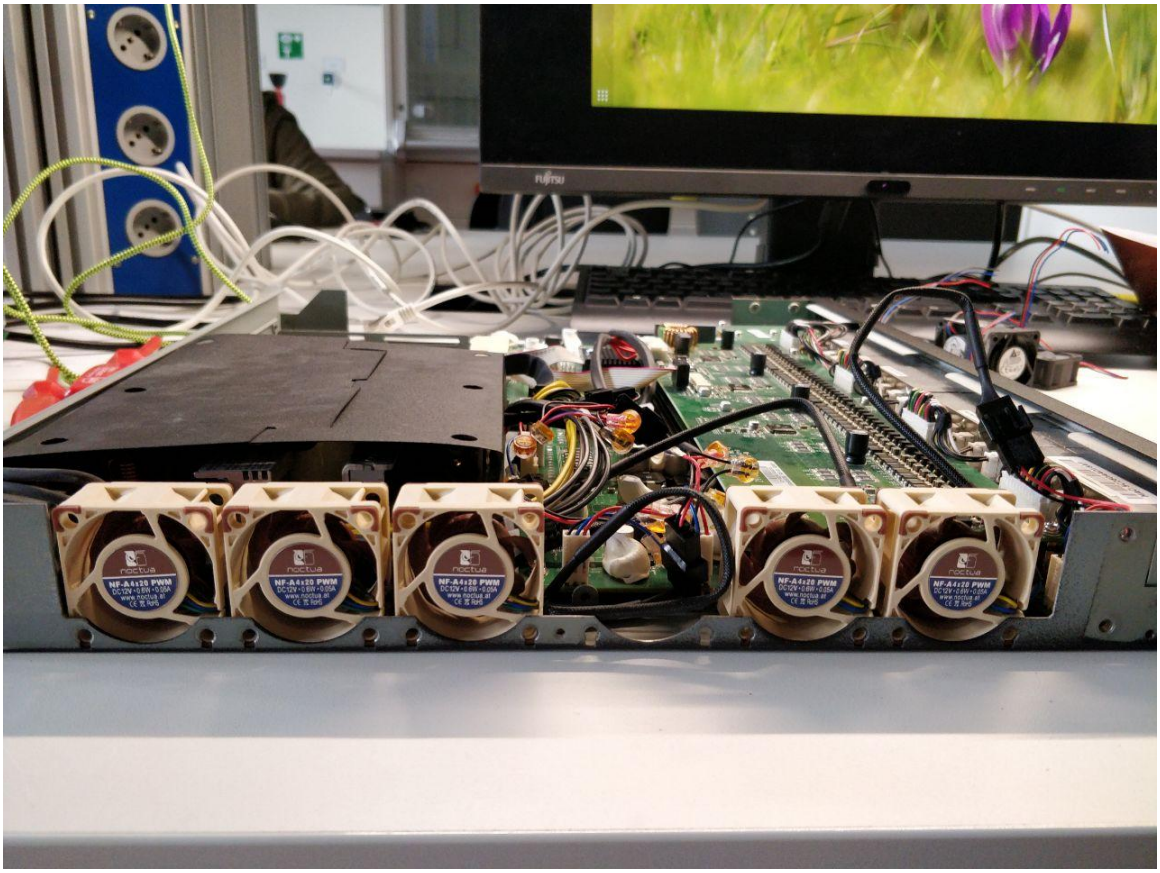


Figure 40: Dell PowerConnect 6248P with swapped fans

4.3 Software: Automation And User Interface

NOTE All the project files from this section can be found within the `installation_setup` and `processing_chain` folder of the repo.

To control the scanning functions easily and effortlessly, a software concept was conceived and executed. It is a modular design in logical units, and is made to be very maintainable and enhanceable. It consists of multiple basic blocks, which in turn are split into several scripts for each basic function. Figure 41 makes the overall concept more accessible, needed components with current progress are shown in Figure 42.

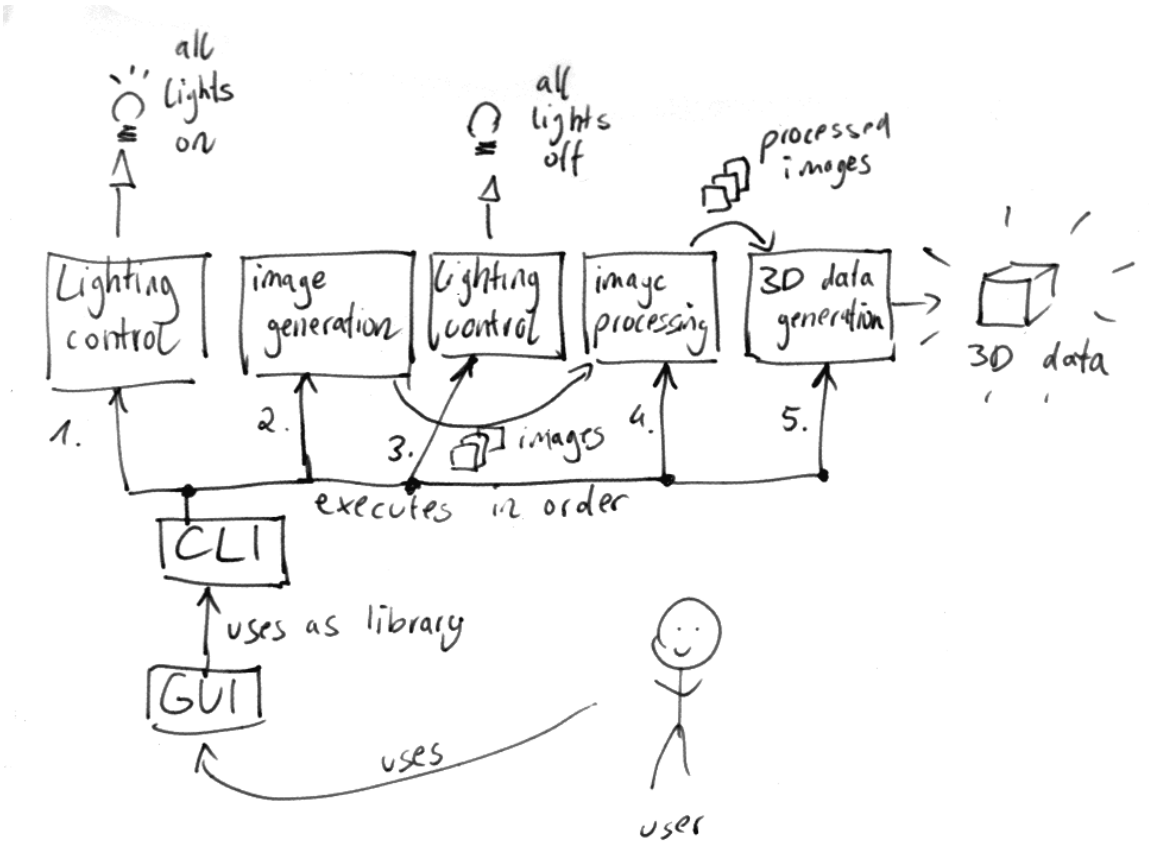


Figure 41: The processing chain structure (start at the user to follow the chain)

Figure 42: The needed blocks, currently none completed

4.3.1 Network Booting Of Raspberry Pies

But before we can address any of those building blocks, there is something else that needs to be automated. Since were are using computers as intermediary for lighting and camera control, those need an operating system. Raspberry Pies traditionally boot from (micro) SD cards, a type of flash storage. This has multiple disadvantages:

- SD cards are costly, since one would be neccessary for each SN
- SD cards are an additional part that can fail, making smooth operation harder
- if the software on the Pi has to be changed - either for a different use case, or because

one of the processing modules is replaced, every OS installation has to be changed manually → extremely time-intensive and tedious process

Thankfully, the Raspberry Pi 4B has the option to boot from the network (which is one of the main reasons why it was used). This network booting is done via PXE (preboot execution environment) and has to be set up on the CN, which will act as PXE server. Further, the bootloader on the Pi does not support PXE out of the box - it has to be flashed to a different bootloader. This flashing process of multiple Pies can be automated with the contents of

`installation_setup/bootloader-flash-utils`

and a cron job that executes

`flash_bootloader.sh`

at reboot (more details in the README). In addition to the much improved maintainability, ommision of the SD card means that the filesystem of the Pies is available locally on the PXE server, which makes editing and moving files from and to the SN easier. The steps required to make PXE work between CN and SN are roughly:

1. flashing a PXE capable bootloader to the SN
2. installing DHCP and PXE server software on the CN
3. configuring the DHCP and PXE servers to work as intended
4. preparing a single OS install for the SN (on the SN)
5. formatting said OS install in a way that makes it network-bootable and copying that over to the CN

The exact process is detailed in the README and involves the two automation scripts

`pxe_setup_script_static.sh`

and

`pxe_refresh_OS.sh`

from which the latter can be used to easily boot a whole network of SN into a different preconfigured OS while backing up the previously used SN OS at the same time.

4.3.2 Lighting Control Over The Network

The CN needs to be able to control the lighting to increase the quality of the images produced by the SN by increasing the SNR. On the other hand, it is difficult to directly attach lights that are able to be regulated to the CN. Usually, a microcontroller is used to bridge the gap between flexible lighting solutions and a general-purpose computer.

Since we already use Raspberry Pies as intermediaries for the cameras, and those provide microcontroller-like functionality via their GPIO (general purpose input/output) headers, the lighting is governed by them instead of using additional hardware. Because the SN are already networked to the CN, software control of the lighting is also much simpler.

To reduce the amount of components needed and at the same time increasing flexibility of the lighting, individually addressable RGBW LED strips were chosen as a light source. Those can produce light in many different colors and brightness levels. Since they are individually addressable, there is quite the variety of effects that can be rendered on the LED strips. The only limiting factor here is the software: each tower will display the same LED pattern due to the simplistic way the software is set up.

The software is a very straightforward transmitter-receiver combination. The transmitter is executed on the CN, which will transmit a user-definable string at the broadcast IP of the subnet (x.y.z.255) at port 5353. On the SN's filesystem, there are python scripts around the `/home/pi` directory which contain code that can control RGBW strips connected to GPIO pin 18 (and, by altering a line within the code, all the other PWM pins). The receiver is running in the background on the Raspberry Pies - it tries to execute the string sent by the transmitter under Python3.7. The idea is to send the name of an LED control script with the transmitter - the receiver will then execute said script under root permission, taking control of the GPIO and setting up the LED strip.

Be that as it may, some effects should loop until different lighting is needed. This poses a problem, since only one script can control the GPIO at the same time. Nevertheless, a solution was found. If the scripts are run in a known process using

```
process = subprocess.Popen(..., shell=True, preexec_fn=os.setsid)
```

then this process can be killed by

```
os.killpg(os.getpgid(process.pid), signal.SIGTERM)
```

and access to the GPIO is freed again. So whenever the receiver receives a message, it frees the GPIO by killing the old process, then runs the received message. Figure 43 is a visualization of the lighting control process. The dashed line marks seperation between CN and all of the SN. The left side is the CN part of the software, the right side is the SN part.

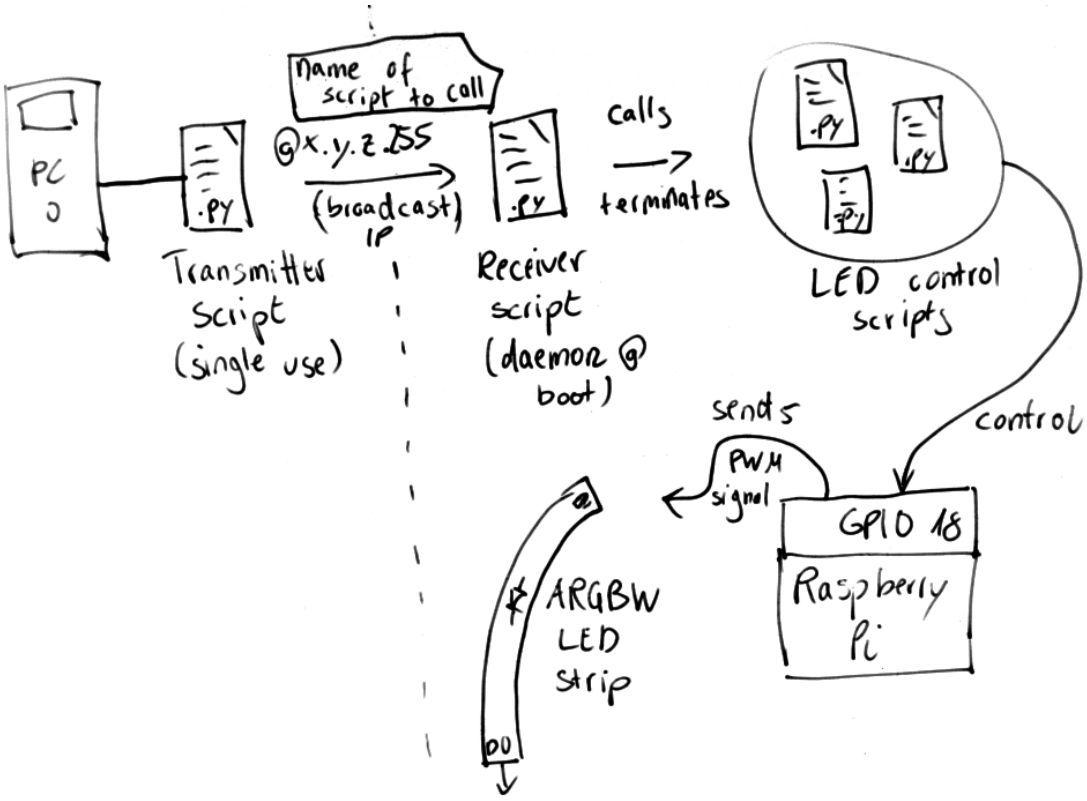


Figure 43: Schematic representation of the lighting control software

Figure 44: The needed blocks, currently one completed

4.3.3 Synchonized Image Capture

To achieve the best results with a full body scanner, the images need to be taken at the exact same time. If they are delayed and the subject moves, there are motion artifacts which manifest as duplicate 3D structures (see Figure 45). Additionally, the photos need to be taken at a higher shutter speed (about $\frac{1}{50}s$ for humans trying to stand still) to avoid motion blur, which would otherwise diminish the detail in the images significantly.

Figure 45: A duplicate 3D structure to the top left of the stone, caused by wrong camera alignment in this particular case

Luckily, a complete solution to the problem of synced capture with multiple Raspberry Pies exists: Compound Pi [3] (also referred to as CPi). It provides an API (application programmer interface) to use with Python2 and encompasses camera settings control as well as capture and image download to the array controller (which is the CN in our case). It consists of a client part, which runs on the CN and a server part, running on the SN.

Though to enable synchronized capture, time between the systems needs to be in sync for the reason that Compound Pi uses timestamps as capture trigger. For instance: if you give it a sync time of $2.0s$ and the time on the CN is $12:34:56$, it will set a timestamp $2.0s$ into the future, at $12:34:58$. Then, all the SN capture an image when this timestamp is reached within their own time state.

This has a major disadvantage if the times of the participants are not in sync: the timestamp is set in reference to the CN time, but triggered on the SN time. In case of a mismatch, it will either trigger never (if SN time is behind the set timestamp already), or late (if SN time is one hour behind and you set a sync time of $2.0s$, it will take $1:00:02$ until a photo is captured). In both cases, the SN side of the program locks up and the SN has to be rebooted. Unfortunately, CPi does not provide methods to check for synchronization between client and server, and NTP does not provide a way to force synchonization from the server side. It would be possible to force a synchronization on the client side, but that has not been implemented. The idea behind that is if additional software has to be written for the capture anyway, one might very well just replace CPi altogether. Another disadvantage of Compound Pi is its age: it is only compatible with Python2, which is obsolete and slowly but surely being rotated out of many of the main linux distribution's packages - meaning it will become harder and harder to install its dependencies and execute it in the future. In addition to this issue, the downloading of images is both sequential and very slow at the same time. To increase responsiveness of the array control, this is the module

to upgrade.

HSkanner3D uses the API and infrastructure provided by Compound Pi to control the cameras and capture images. Figure 46 shows the software infrastrucure and interactions required for synced capture with Compound Pi.

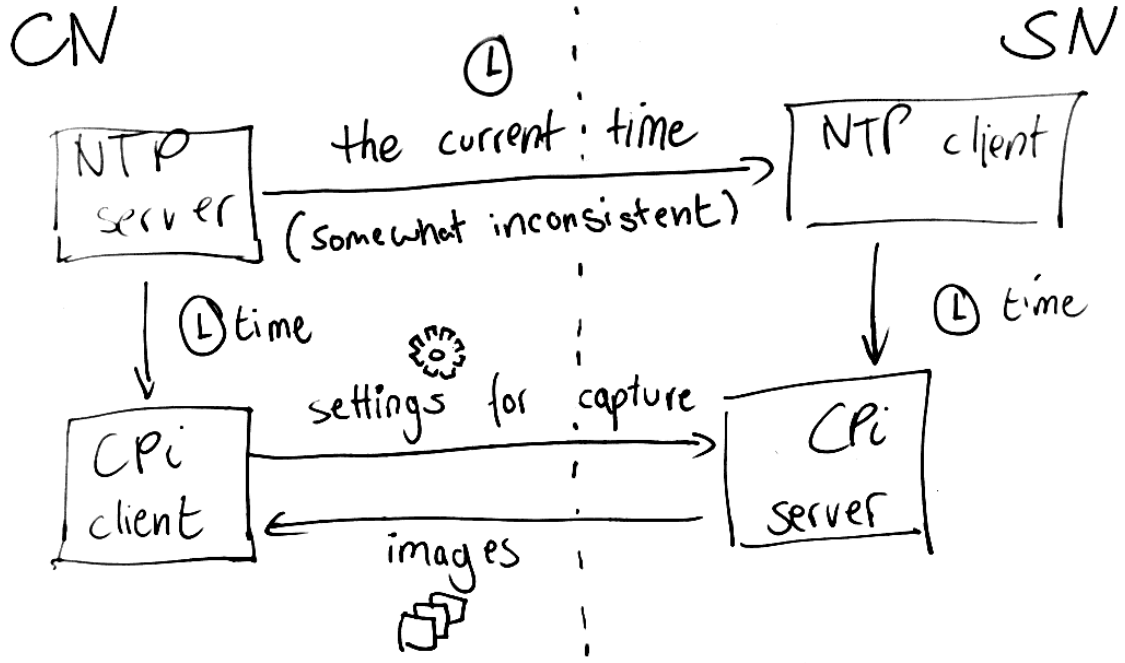


Figure 46: Software architecture for synchonized capture

Figure 47: The needed blocks, currently two completed

4.3.4 Filtering 2D Images

The images as they come from the cameras might not be perfectly suited for processing into 3D data. To create more reliablility or higher quality models, one might

- denoise images
- segment the image into subject and background, then remove the background to avoid computation of unnecessary data
- enhance detail in the image (sharpening, clarity, ...)
- correct exposure

Yet most of those things can be managed by choosing decent camera settings in the first place and are thus not strictly required. So, as of now, all the 2D filtering does is rotate and scale the images so that the viewer does not have to turn their head to look at them the right way up. Note that image rotation does not matter for the 2D $\rightarrow$ 3D conversion. Setting rotation to $0°$ and scaling to 100 (percent) turns the filtering into a simple copy-paste, making it faster.

Figure 48: The needed blocks, currently three completed

4.3.5 Generating 3D Data From The Images

Even though there is software for lighting and image generation now, we still don't have 3D data. There are multiple ways to extract depth information from 2D images, most under the umbrella of photogrammetry software. The software of choice should have several key features:

- a way to automate it easily
- low cost
- good processing speed
- the ability to run on the platform (Linux)

This list of requirements limits the options severely. Noteable are two pieces of software in particular. First, there is Agisoft Metashape [4] with its defining characteristics:

- has a Python API to automate
- Standard Edition node-locked license for education is $59\$US$ (which does not have the Python API), Professional Edition is $549\$US$ (which is needed for Python API) and thus very expensive
- is blazing fast compared to Meshroom
- runs natively on Linux (Bonus: it does not use CUDA and so it does not require the use of an NVidia GPU)

Secondly, there is Alicevision Meshroom [5], which brings its featureset to the table:

- has the option be automated via CLI

- is free open source software
- is **very** slow compared to Metashape (it takes the same time from images to finished model in Metashape as it takes Meshroom to align all the images and do a sparse reconstruction)
- runs nativly on Linux (but needs a CUDA capable GPU for dense reconstruction)
- has more features for debugging and changing the 3D processing chain than Metashape (ability to view extracted features, matched features, lots of settings and nodes)

For the purpose of use in the scanner, Meshroom seems like the better fit for now, as one of the main goals is accessibility and easy maintenance. It is also free, which helps with the tight budget. The Meshroom GUI is shown in Figure 49 and the workflow is as follows:

1. images are loaded into the program, for example by dragging them onto the left side of the 2D Viewer
2. processing nodes of Meshroom are added/removed as needed within the Graph Editor
3. settings of the nodes are adjusted by the user on the right side of the graph editor
4. user rightclicks on a node within the editor and selects the `Compute` option
5. all the required nodes to compute that node are executed in order
6. output can be observed in the 3D Viewer (Figure 49 features a coarse reconstruction with 11000 points and camera alignment - output of the StructureFromMotion node)
7. the output files can be obtained from their automatically generated folders

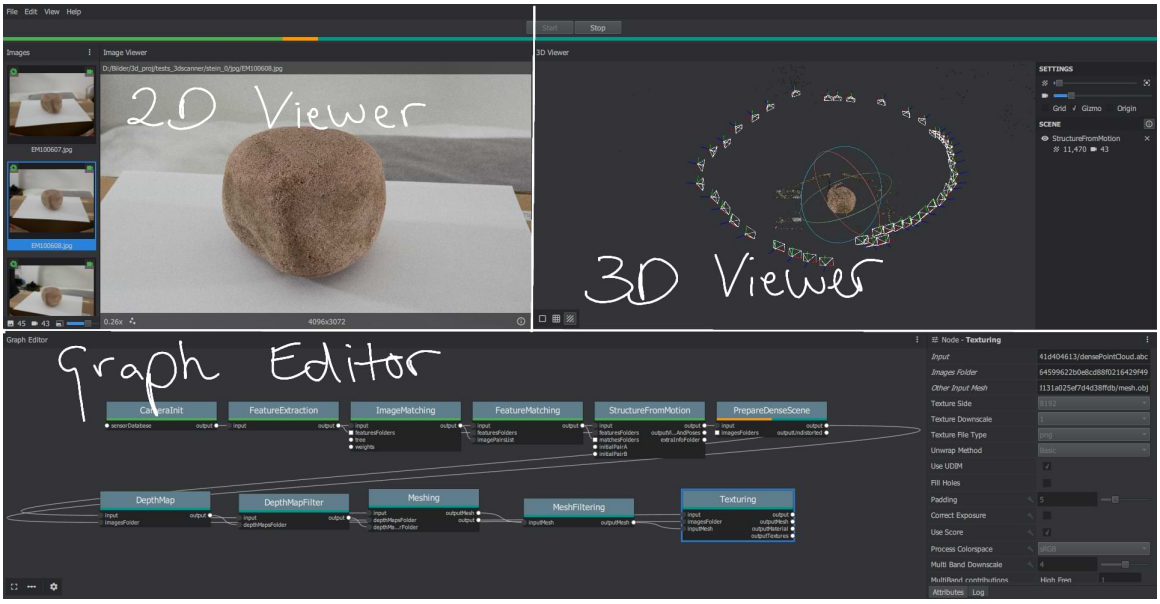


Figure 49: Overview of the Meshroom GUI with the default node graph

To automate this process, Meshroom provides a CLI. The Meshroom CLI only needs three arguments:

1. input directory (where the images are)
2. output directory (where the output data should be put)
3. pipeline file directory and name (path)

The pipeline file has the extension of `.mg`. It contains the processing nodes and their wiring as well as some other information about Meshroom settings and imported images. It can be generated by opening the Meshroom GUI, mapping out a series of processing nodes in the Graph Editor for it to compute, then simply saving. The pipeline file is similar to a project file in other programs. For it to work properly with the CLI, there are some requirements:

- only one input node (CameraInit)
- only one output node (the Publish node - the output of processing nodes can be connected to this node to be saved to the output directory)
- it is better not to have any images imported - start with a fresh project

All the nodes have to complete before publishing. This causes the limitation that during a dense reconstruction, no intermediate steps like the coarse reconstruction can be observed throughout the computation. The limitation of only one input node means that there is currently no way to have a set of 'alignment rough shots' and 'detail shots for enhancement' using the CLI. You are at the mercy of Meshroom managing all images at once into a decent model. Figures 50 and 51 show the graphs for HSkanner3D 'faster processing' and 'higher quality' settings respectively (at the time of writing). To speed up the 3D generation process by a lot, this is the software to replace.

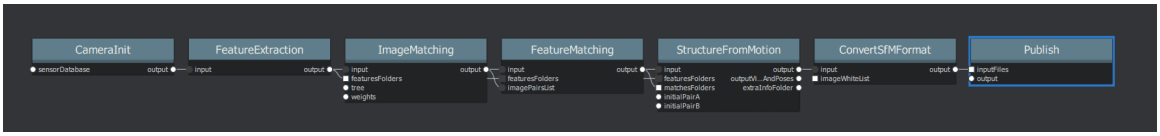


Figure 50: The node graph for the 'faster processing' preset in the HSkanner3D GUI: only coarse reconstruction

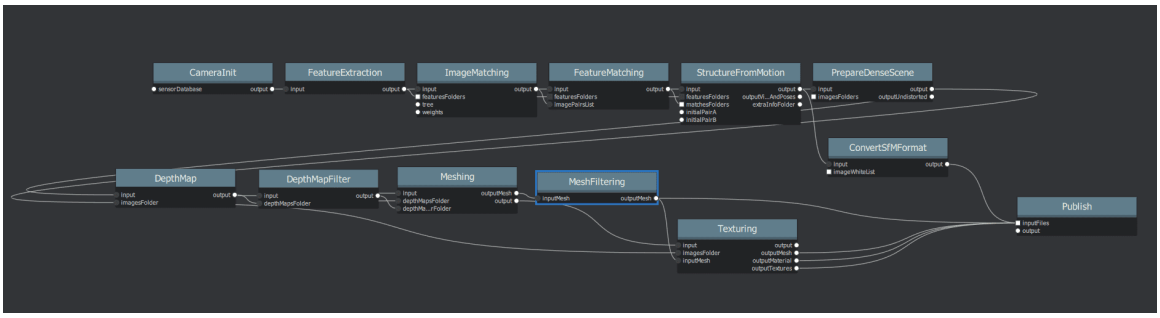


Figure 51: The node graph for the 'higher quality' preset in the HSkanner3D GUI: dense reconstruction, meshing and texturing

To obtain excellent and repeatable results with the software, it is important to know how it works. The official Meshroom documentation can be found here [6]. Additionally, one might look at section 5.1.

Figure 52: The needed blocks, currently four completed

4.3.6 User Interface

At this point, it is possible to control cameras, lighting and computation of 3D data by executing several scripts. While that is much more convenient than doing it manually, it is still relatively hard for someone who did not read up on the scanner to use it. To overcome this last hurdle, a GUI was made. The job of the GUI is to provide simple buttons for control over the scanner as well as its settings with minimal effort from the user.

To keep with the theme of software that is simple to use and maintain, Tkinter was chosen as a GUI framework. It can be loaded in Python as a library and provides much of the basic functionality needed to create a GUI to our liking. Since the settings have to persist between sessions, some kind of nonvolatile save state had to be made. The simplest way to do this was to create a configuration file which can be read from and written to by the GUI. There exist many different formats for configuration files, such as `.xml`, `.json` and `.ini`. The `.ini` style of configuration was chosen because there is a Python library readily available that can handle writing to and reading from such files with ease. Additionally, it is also very readable by humans.

I split the user interface into two tabs: the 'Tasks' tab and the 'Settings' tab. The 'Tasks' tab (see Figure 53) contains everything around basic scanner control:

- image capture and coarse reconstruction

- image capture and dense reconstruction
- image capture
- closing the application

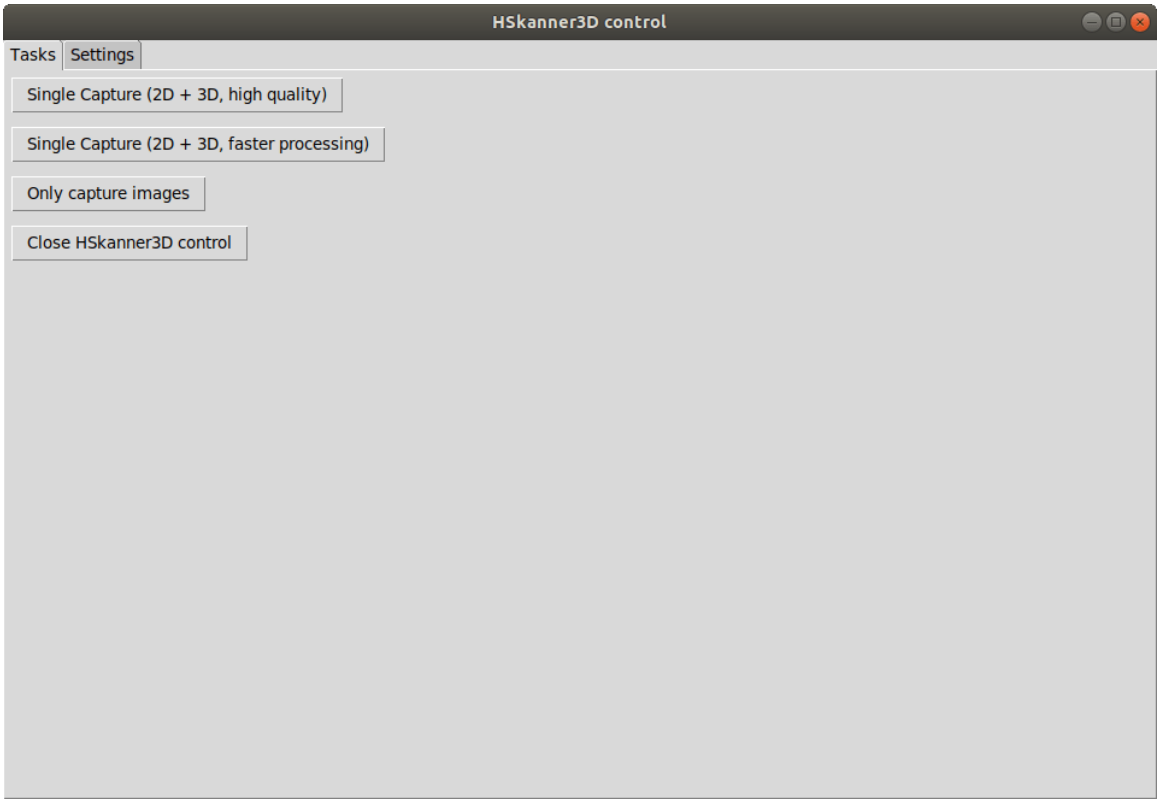


Figure 53: The 'Tasks' tab of the GUI

The 'Settings' tab (see Figure 54) is intended for advanced users and developers to tweak or set up the scanner. It contains all the variables from the configuration file, which are loaded dynamically - so adding of variables does not require manual modification of the GUI layout. To add a variable, add it in the configuration file in both default and custom sections. Then, define it in the HSkanner3D script file in the constructor of the hska3d class, keeping the same syntax as the rest of the loaded variables. Finally, add the variable to the script call in its respective position to pass it.

The 'Load defaults' button writes the default values from the config file to the text fields, then saves them as user configuration. The 'Check and Save Settings' checks the text fields for data type first. If the data type of the input is not equivalent to the previous user configuration, it will not save it - rather, it clears the field in question until all errors are resolved. It does not check ranges or content - just data type. If you want to change the data type of a variable, you have to edit the config file manually, then close and reopen the GUI. Function of the individual variables is explained in the README.

Note that one does not have to use the GUI to control or debug the scanner. Due to the modularity of the software, each aspect of the scanner can be used and tested individually

by using the respective script for the job.

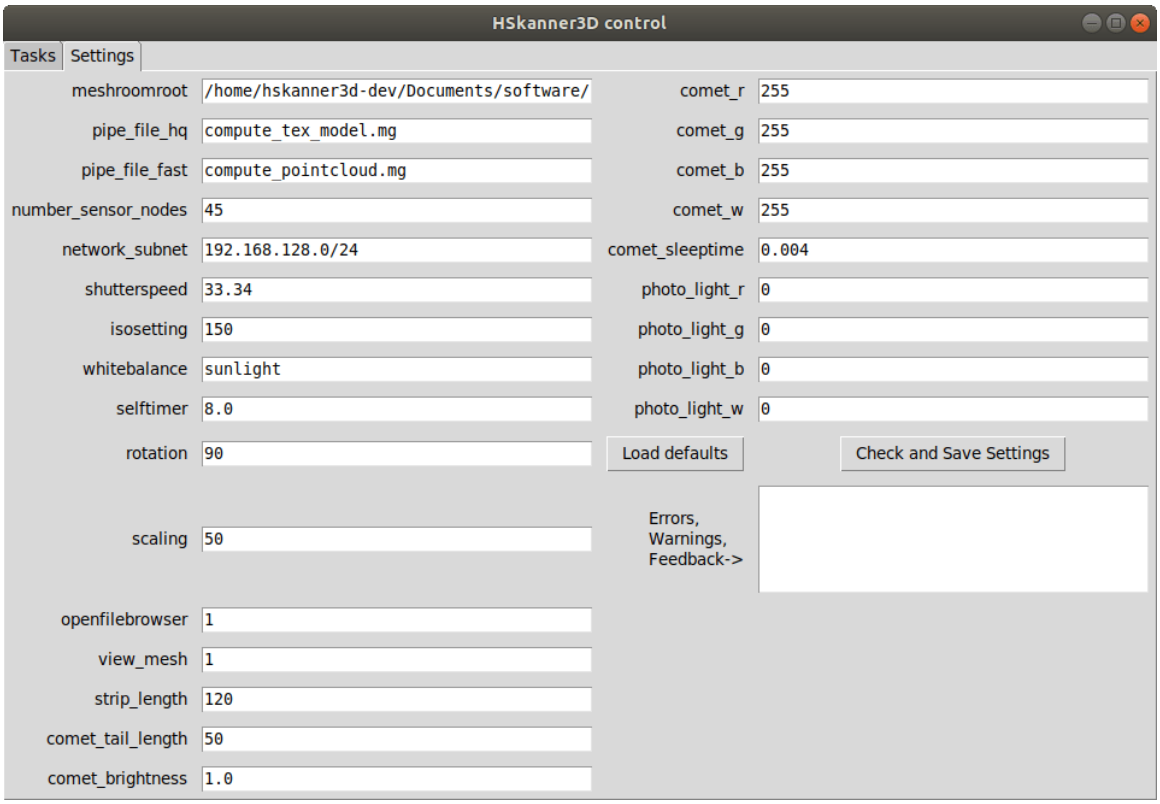


Figure 54: The 'Settings' tab of the GUI

Figure 55: The needed blocks, all completed

4.4 Overview

At this point, all the required software and hardware components are developed and implemented. The resulting overall system architecture of the scanner is shown in Figure 56.

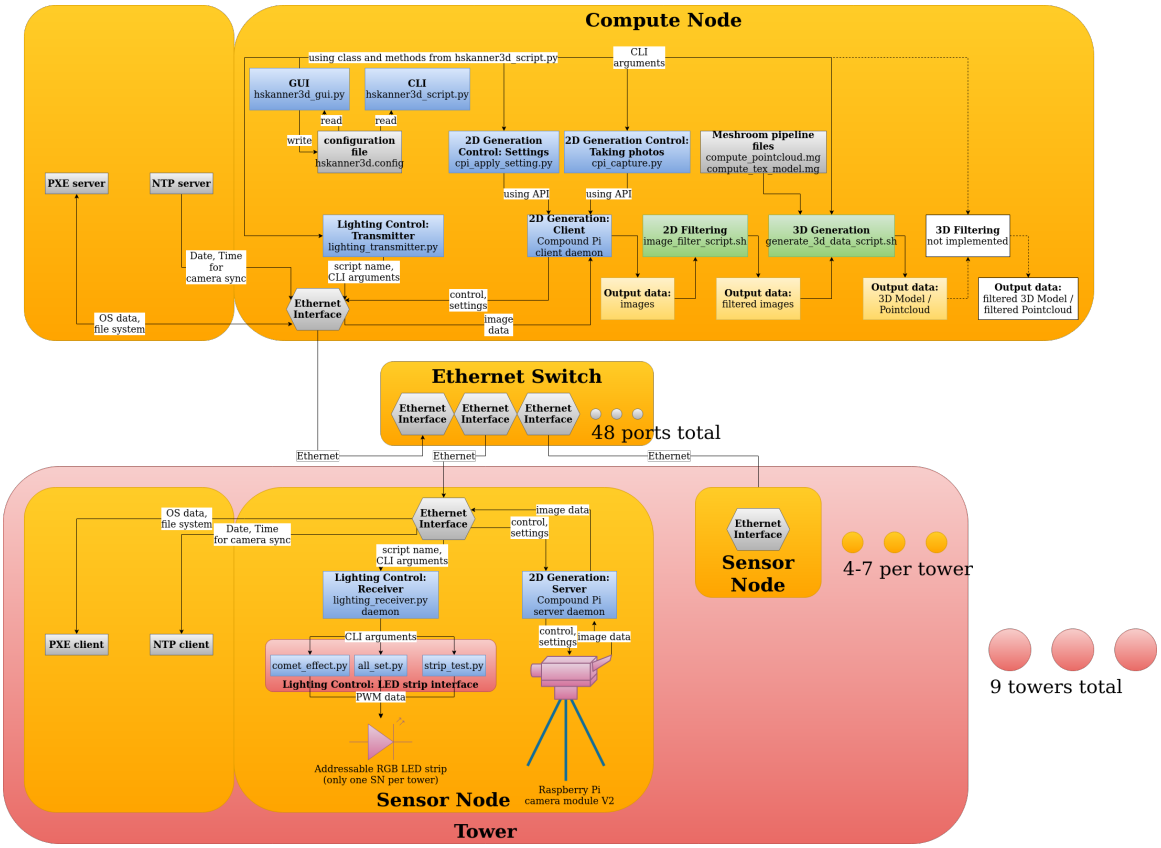


Figure 56: Overall system architecture of HSkanner3D

To make understanding of this rather complex image easier, we will take a look at an example use case: An operator wants to scan somebody in high quality. The process is as follows:

4.5 From cold boot to 3D model - an example

1. scanning array and CN are powered
   - CN boots
   - ethernet switch boots
   - all SN boot over the network using PXE
2. all SN receive current time from NTP server on CN
3. operator logs into the OS
4. before starting the GUI, the operator either waits about 10 minutes or monitors the system state. Figure 57 shows the network traffic during normal PXE booting. The operator makes sure that there is no low network traffic for about a minute after the prolonged load. They can also check if all the green status LEDs on the Pies are flashing twice rapidly. If they are lit continously instead, they have to wait a little while longer.

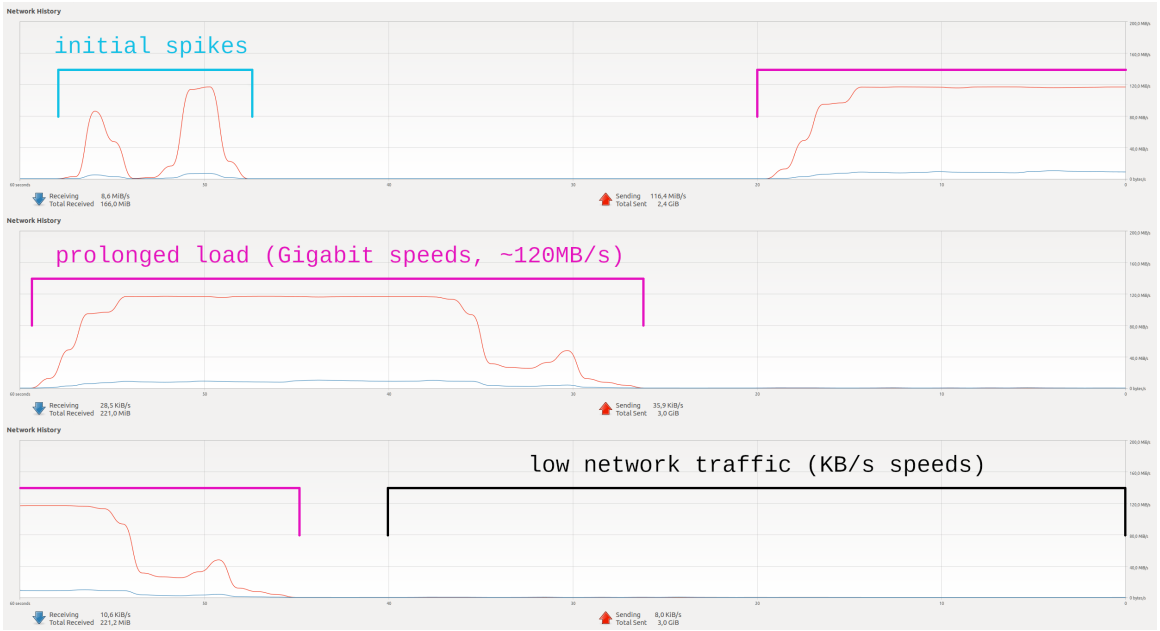


Figure 57: Ethernet traffic during PXE boot process monitored using System Monitor

5. operator starts up the GUI by executing `hskanner3d_gui.py`
6. GUI initializes scanner and itself
   - loading `hskanner3d_script.py` as library
   - reading `hskanner3d.config` file to get all settings
   - using `cpi_apply_settings.py` → applies capture settings to SN → uses CPi client to send capture settings to CPi servers
   - using `lighting_transmitter.py` → sends command over network → SN receives command using `lighting_receiver.py` → SN executes `comet_effect.py` with custom settings for idle animation of LEDs
7. operator checks if time on SN and CN are in sync by using SSH and executing `timedatectl` on one SN. One can also utilize the `watch -n 1` command for convenience. Figures 58 to 60 show the process. In Figure 60 one can also obeserve the time zone and correct it using raspi-config if it is not the same as on the CN. Ignore that it says 'System clock synchronized: no' - the clock is synchronized. Time synchronization with NTP usually takes a couple minutes, since the polling interval is at the default 64 seconds to keep network traffic low. If one does not want to wait, the selftimer can be set to 0.0, avoiding synchronized capture altogether and instead firing as fast as possible after receiving the command.

```
hskanner3d-dev@compute-node:~$ cat /var/lib/misc/dnsmasq.leases
1613422497 dc:a6:32:e3:96:80 192.168.128.120 raspberrypi 01:dc:a6:32:e3:96:80
1613422454 dc:a6:32:b3:f7:a9 192.168.128.179 * 01:dc:a6:32:b3:f7:a9
1613422449 dc:a6:32:b3:f3:a1 192.168.128.173 * 01:dc:a6:32:b3:f3:a1
1613422454 dc:a6:32:e3:96:9e 192.168.128.144 * 01:dc:a6:32:e3:96:9e
1613422454 dc:a6:32:e3:8a:d9 192.168.128.108 * 01:dc:a6:32:e3:8a:d9
1613422449 dc:a6:32:e3:98:65 192.168.128.190 * 01:dc:a6:32:e3:98:65
1613422444 dc:a6:32:e3:97:b3 192.168.128.117 * 01:dc:a6:32:e3:97:b3
1613422449 dc:a6:32:e3:98:9c 192.168.128.140 * 01:dc:a6:32:e3:98:9c
1613422444 dc:a6:32:e3:8b:9d 192.168.128.198 * 01:dc:a6:32:e3:8b:9d
1613422454 dc:a6:32:e3:9a:d9 192.168.128.200 * 01:dc:a6:32:e3:9a:d9
1613422449 dc:a6:32:e3:98:33 192.168.128.136 * 01:dc:a6:32:e3:98:33
1613422444 dc:a6:32:e3:95:fc 192.168.128.187 * 01:dc:a6:32:e3:95:fc
1613422449 dc:a6:32:e3:96:db 192.168.128.103 * 01:dc:a6:32:e3:96:db
1613422454 dc:a6:32:e3:98:36 192.168.128.139 * 01:dc:a6:32:e3:98:36
1613422454 dc:a6:32:e3:97:d9 192.168.128.151 * 01:dc:a6:32:e3:97:d9
1613422454 dc:a6:32:e3:9b:36 192.168.128.189 * 01:dc:a6:32:e3:9b:36
1613422454 dc:a6:32:e3:98:1e 192.168.128.116 * 01:dc:a6:32:e3:98:1e
1613422454 dc:a6:32:e3:98:57 192.168.128.172 * 01:dc:a6:32:e3:98:57
1613422454 dc:a6:32:e3:97:88 192.168.128.171 * 01:dc:a6:32:e3:97:88
1613422454 dc:a6:32:e3:96:ae 192.168.128.159 * 01:dc:a6:32:e3:96:ae
1613422449 dc:a6:32:e3:97:c7 192.168.128.133 * 01:dc:a6:32:e3:97:c7
1613422454 dc:a6:32:e3:97:d5 192.168.128.147 * 01:dc:a6:32:e3:97:d5
1613422493 dc:a6:32:e3:97:97 192.168.128.186 * 01:dc:a6:32:e3:97:97
1613422493 dc:a6:32:e3:98:7b 192.168.128.107 * 01:dc:a6:32:e3:98:7b
1613422449 dc:a6:32:b3:f7:70 192.168.128.119 * 01:dc:a6:32:b3:f7:70
1613422444 dc:a6:32:e3:97:e5 192.168.128.163 * 01:dc:a6:32:e3:97:e5
1613422454 dc:a6:32:e3:95:36 192.168.128.191 * 01:dc:a6:32:e3:95:36
1613422449 dc:a6:32:b3:f7:97 192.168.128.158 * 01:dc:a6:32:b3:f7:97
1613422454 dc:a6:32:b3:f7:5b 192.168.128.199 * 01:dc:a6:32:b3:f7:5b
1613422454 dc:a6:32:b3:f5:f9 192.168.128.156 * 01:dc:a6:32:b3:f5:f9
1613422454 dc:a6:32:b3:f6:d4 192.168.128.169 * 01:dc:a6:32:b3:f6:d4
1613422392 dc:a6:32:e3:98:81 192.168.128.114 * *
1613422449 dc:a6:32:e3:98:3f 192.168.128.148 * 01:dc:a6:32:e3:98:3f
1613422449 dc:a6:32:e3:97:2a 192.168.128.178 * 01:dc:a6:32:e3:97:2a
1613422454 dc:a6:32:e3:96:20 192.168.128.118 * 01:dc:a6:32:e3:96:20
1613422489 dc:a6:32:e3:95:06 192.168.128.143 * 01:dc:a6:32:e3:95:06
1613422440 dc:a6:32:e3:82:76 192.168.128.113 * 01:dc:a6:32:e3:82:76
1613422425 dc:a6:32:b3:f7:76 192.168.128.125 * 01:dc:a6:32:b3:f7:76
1613422418 dc:a6:32:c6:76:2b 192.168.128.115 * 01:dc:a6:32:c6:76:2b
1613422420 dc:a6:32:e3:97:34 192.168.128.188 * 01:dc:a6:32:e3:97:34
hskanner3d-dev@compute-node:~$ ssh pi@192.168.128.120
```

Figure 58: Looking up DHCP leases for the IP of a Pi to SSH into and connecting via SSH

```
Every 1.0s: timedatectl

               Local time: Fri 2021-02-12 15:18:02 GMT
           Universal time: Fri 2021-02-12 15:18:02 UTC
                 RTC time: n/a
                Time zone: Europe/London (GMT, +0000)
System clock synchronized: no
              NTP service: inactive
          RTC in local TZ: no
```

Figure 59: Output of timedatectl, not in sync yet

```
Every 1.0s: timedatectl

               Local time: Mon 2021-02-15 08:58:34 GMT
           Universal time: Mon 2021-02-15 08:58:34 UTC
                 RTC time: n/a
                Time zone: Europe/London (GMT, +0000)
System clock synchronized: no
              NTP service: inactive
          RTC in local TZ: no
```

Figure 60: Output of timedatectl, in sync

8. person to be scanned walks into scanner and faces tower 1
9. operator clicks on 'Single Capture (2D + 3D, high quality)' button
10. GUI window shows that it is busy by freezing (intended behaviour, to avoid accidentally running two scans at once or changing settings mid capture)
11. GUI turns all LED lights on using `lighting_transmitter.py` → sends command over network → SN receives command using `lighting_receiver.py` → SN uses `all_set.py` to turn all LEDs to the user defined color and brightness
12. GUI generates image data using `cpi_capture.py` → uses CPi client to send capture timestamp to CPi servers → uses CPi client to download captured images from CPi servers
13. GUI turns all LED lights back to idle animation using `lighting_transmitter.py` → sends command over network → SN receives command using `lighting_receiver.py` → SN executes `comet_effect.py` with custom settings for idle animation of LEDs
14. GUI filters images using `image_filter_script.sh` → copies files over to output folder → filters images in output folder

15. GUI generates 3D data from the filtered images using `generate_3d_data_script.sh` → interfaces with Meshroom, passing `compute_tex_model.mg` as pipeline file → Meshroom calculates 3D model with texture

Checking the state of the scanner requires training and is not simple, contrary to our goals. This is due to how Compound Pi works (or doesn't, in this case). This issue can be resolved by using or writing better software, or by just waiting a long time to ensure everything has booted and synchronized. If CPi is run before all the SN are booted, it will sometimes hang the SN side of the program on some SN. In this case, reboot both CN and SN.

5 Results

5.1 Output Data

Now that the scanner has matured in software and hardware, it is high time to scan some people.

`...or is it?`

Figure 61 shows the Meshroom GUI. In the left third, there are the input images. In the middle, an image with extracted SIFT features (green) and matches that made it into the SfM (red) is displayed. To the right, the 3D viewer shows the coarse point cloud generated from the input data as well as camera positions.

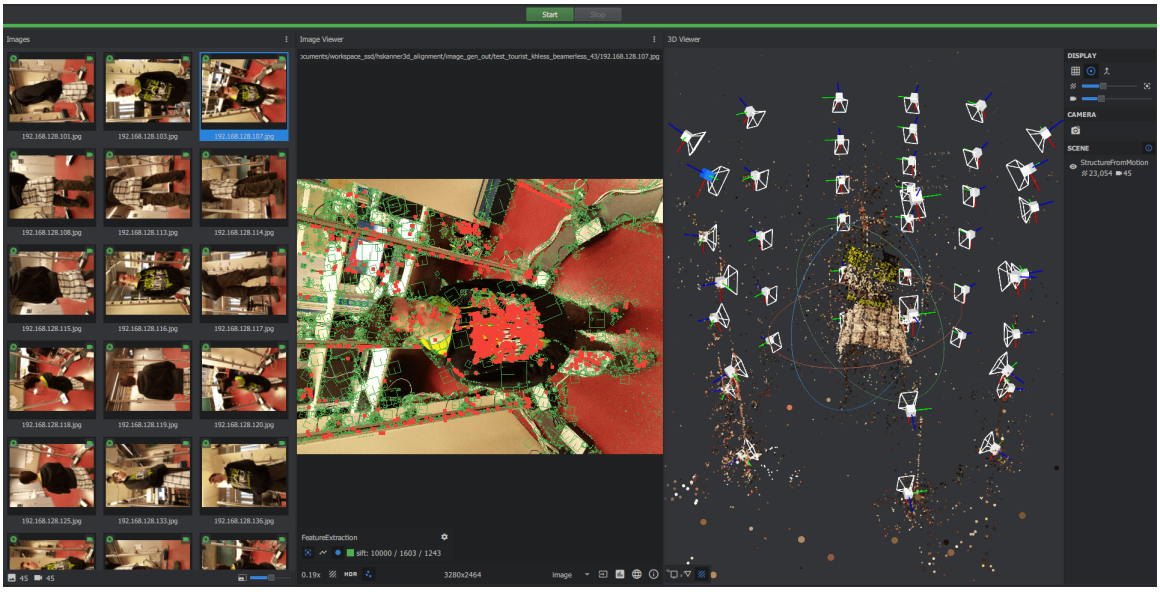


Figure 61: Meshroom GUI with some initial scanning results

For basic 3D data generation, Meshroom has to execute some steps.

1. the images are initialized using their metadata and the Meshroom camera database

2. features are extracted using algorithms of choice (SIFT was used here, since it is most consistent and accurate - green squares in the middle of the GUI)
3. images are put into multiple pairs
4. features are matched between image pairs
5. feature matches are used to calculate camera positions and feature depth to generate a coarse point cloud (results on right side of the GUI, number of aligned cameras in the far bottom left behind the number of images)

For those initial steps, there is a hurdle to overcome:

- if there are not enough good matches between image pairs, the camera can not be aligned in 3D space
- if the camera is not aligned in 3D space, the matched features of that image are also not in the dense point cloud
- if the camera is not aligned in 3D space, later processing steps can not generate data from that view - effectively, it is like the shot was never taken

This means that for a successful alignment, all cameras need enough matches between each other to align. Let's start with the input images. There are 45, one for each SN. The subject is wearing a black hoodie, brown cargo pants and a plaid shirt around the waist. They have brown hair. In the background of the selected image in the preview (middle), a table can be seen. That table is white, and it is clipping with the current exposure settings (meaning that there is no detail in the surface, it is pure white, `0xffffff` in hex triplets). Most of the hoodie, and parts of the hair, on the other hand, are pure black (`0x000000`). This is due to dynamic range restrictions since we are working with JPEG files and Compound Pi does not support taking RAW captures (which is yet another reason to replace it). Thus, the chosen exposure was a compromise between just barely clipping bright objects while losing some of the shadows. Looking closer at the image viewer, we can observe that there are barely any SIFT features (green) extracted from the black parts of the hoodie in this view. Luckily, the hoodie has a pattern in the front that is just covered with very reliable details (details that stay the same from angle to angle and camera to camera), as can be seen by the number of matches in that area. Similarly, the plaid shirt in the back provides plenty detail for the features in the images to be matched reliably. This can also be observed on the right side of the GUI in the SfM result, which shows mostly the hoodie front and plaid shirt. In this particular scan, every camera can see either the shirt or hoodie details. Thus, all cameras can be and have been aligned.

While this is an example that works (at least in the alignment), many cases don't. That is because the subject needs to have sufficient detail in their clothing for all the cameras to align. One possible solution to this would be precise alignment using an ideal subject and using this alignment in futher computation. If the cameras don't have to align, the subject can lack features to a degree.

In further steps, Meshroom computes a dense point cloud using the known camera positions and gathering depth maps from them and the image content. If the image content barely has any features or only weak ones - in this particular case, the trousers, the software has great difficulty deriving depth data from it. Figures 62 and 63 show a mesh generated from this particular image set.

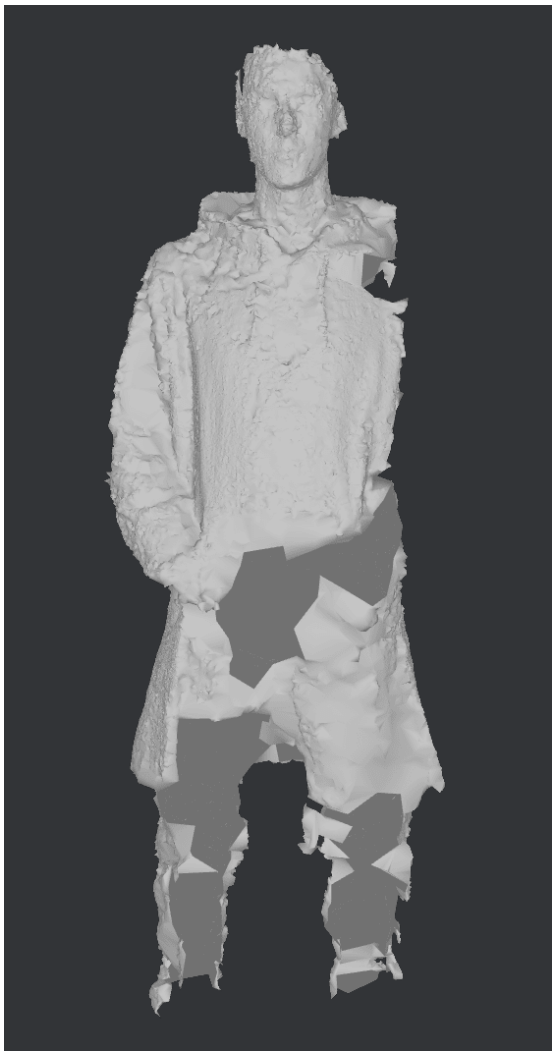


Figure 62: Frontal overview of the mesh

Figure 63: Back overview of the mesh

As you can see, there is a severe lack of geometry. The hair had too little detail and was partly lost to the shadows, as was part of the hoodie and trousers. The parts that came out great were the ones that had high detail in the first place: the front of the hoodie and the plaid shirt (see Figure 64). Note that even though the face was not seen from that many cameras and didn't have that many feature matches, it still has some respectable representation in the 3D model (see Figure 65). Texturing also works pretty well, as can be

seen in Figure 66.

Figure 64: The plaid shirt in the 3D reconstruction

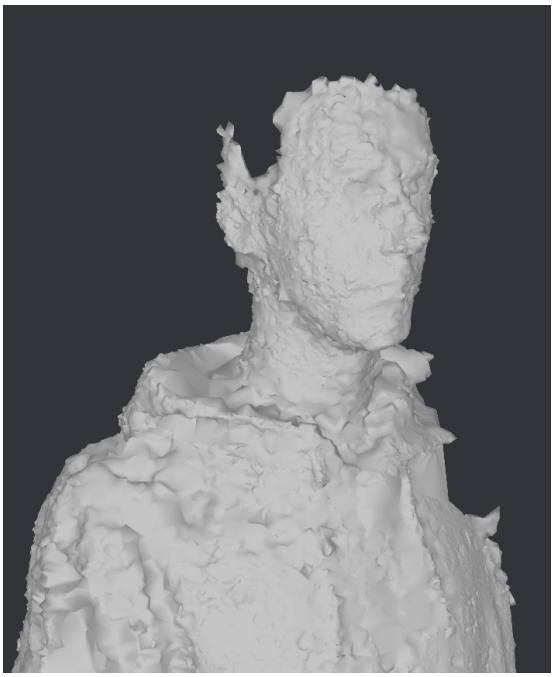


Figure 65: The face in the 3D reconstruction

Figure 66: The plaid shirt, textured

5.2 Future Improvements And Enhancements

There exist a plethora of possible improvements and enhancements that can be made to the system. To name a few:

- Up the accuracy and reliability of the scanning array
  As discussed in Section 5.1, the scanner still has some problems regarding the resulting model and point clouds. To combat this, an individual calibration of all cameras (correcting lens distortion, color inaccuracies, ...) can be made to make sure each camera is as similar to the others as possible.

  Additionally, since the cameras don't move, if a set of images with those calibrated cameras is taken, the camera positions can be estimated much more precisely using a photogrammetrically ideal subject (random pattern with lots of contrast on a pillar), then locked. This also speeds up the computation, as the camera positions don't have to be estimated every time. To know exactly which image corresponds to which camera, it would be good to rewrite the image capture software to allow for embedding of the ethernet MAC address of the Pi or Raspberry Pi / camera module serial number into the image metadata as serial number.

- Improve scanner responsiveness
  By replacing Compound Pi, scanner reaction time can be improved: Images will be captured faster after pressing the button and transferred faster to the CN. Ideally, a communication concept will be used that allows for information exchange from both parties. If the image capture software is rewritten, time synchronization can also be forced and confirmed from the SN side and RAW data can be obtained.

- Boost 3D model calculation speed
  If Meshroom is replaced by a faster solution like Metashape, the calculation of 3D meshes can be sped up significantly (by about a factor of 3...6).

- Increase scanner resolution
  Utilizing more cameras yields better detail in the final model. Additionally, camera types can be mixed, such as wide angle cameras for overall propotions and surfaces, and telephoto for detailed shots of e.g. the face.

  Similarly, if a surface has little detail (no weave or pattern is visible on the cloth, it has a single color, ...), it is hard for the photogrammetry software to estimate depth. With this in mind, we can project a pattern onto the surface, increasing detail. Since we don't want to show those patterns - since they are not part of the subject to be scanned - we have to shoot an additional series of pictures patternless that will provide the textures for the model.

- Getting more from the hardware
  The scanning array of HSkanner3D contains 45 Rasperry Pies with camera modules, arranged in a cylinder. This setup can be used for many purposes other than 3D scanning, such as

  - using the Raspberry Pies as a compute cluster
  - recording video on all cameras for 3D video, motion capture, ...

- Optimizing the output
  As with all photogrammetry scans, the output is a dense point cloud or model with some degree of surface noise, holes and a lot of datapoints. For use as a 3D avatar, this mesh would have to be optimized and filtered. Usually, fine details and reflective surfaces such as glasses are not represented in the output in 3D but only as a texture. Such detail would have to be added.

- Benefit more from the output
  To do more with the 3D model other than looking at it, one could make or import a Blender 3D scene into which the model can be put. This scene can then be rendered, adding lighting to the model and making it fit into the scene better. This could be used to render the face of a subject onto Mount Rushmore to name an example.

5.3 Conclusion

Considering the scope of the project, the number of people involved in the development and their previous skillset, an astounding amount of work has been done in the six months that were the Wintersemester 2020/2021.

- design and automation of complete software solution
- design, assembly and wiring of array structure

On the other hand, the time could have been distributed more efficiently. The software was written and about 80 percent finished before the array design had even been started. This resulted in long wait times from part ordering, delivery and manufacture, which could have been used for writing software. Unfortunately, the goal of scanning people has only been completed partially due to the complexity of the matter (clashing with the simplicity of the Blender model) and the time constraints.

References

[1] `https://youtu.be/bxOtIgJ_5OA?t=12`. Visited 19th September 2020.

[2] `https://www.gbe-bund.de/gbe/!pkg_olap_tables.prc_set_page?p_uid=gast&p_aid=66171153&p_sprache=E&p_help=2&p_indnr=223&p_ansnr=32664914&p_version=2&D.000=3739&D.003=42`. Visited 30th January 2021.

[3] `https://compoundpi.readthedocs.io/en/latest/`. Visited 25th January 2021.

[4] `https://www.agisoft.com/`. Visited 27th January 2021.

[5] https://alicevision.org/#meshroom. Visited 27th January 2021.

[6] https://meshroom-manual.readthedocs.io/en/latest/. Visited 27th January 2021.