TU WIEN Informatics

Integrierung von Azure Kinect Daten in eine Echtzeit Soll-Ist Vergleichspipeline

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Jonas Prohaska, BSc
Matrikelnummer 01449302

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Mag.rer.nat. Dr.techn. Hannes Kaufmann
Mitwirkung: Projektass.in Dr.in techn. Iana Podkosova
Projektass. Dipl.-Ing. Dr.techn. Christian Schönauer

Wien, 8. Mai 2024

Jonas Prohaska	Hannes Kaufmann

Informatics

Integrating Azure Kinect data into a real-time planned-vs-built comparison pipeline

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Jonas Prohaska, BSc
Registration Number 01449302

to the Faculty of Informatics

at the TU Wien

Advisor:	Univ.Prof. Mag.rer.nat. Dr.techn. Hannes Kaufmann
Assistance:	Projektass.in Dr.in techn. Iana Podkosova
	Projektass. Dipl.-Ing. Dr.techn. Christian Schönauer

Vienna, May 8, 2024

Jonas Prohaska　　Hannes Kaufmann

Erklärung zur Verfassung der Arbeit

Jonas Prohaska, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 8. Mai 2024

Jonas Prohaska

Danksagung

Zunächst einmal möchte ich mich für die freundliche Betreuung innerhalb der gesamten VR Research Unit an der TU Wien bedanken, allen voran Iana Podkosova, mit der ich seit der ersten E-Mail eine wunderbare Kommunikation gepflegt habe und die mir immer einen neuen Anstoßpunkt für meine Arbeit gab. Christian Schönauer danke ich für seine technische Expertise und im Besonderem dem Zusammenbasteln unseres Hardware-Setups. Nicht zuletzt möchte ich mich auch noch einmal explizit bei Hannes Kaufmann bedanken, der mir schon bei meinem ersten Praktikum genug Vertrauen schenkte, um mich mit einem Spot Roboter sowie einem gigantischen terrestrisches Laserscanner experimentieren zu lassen.

Auch wenn das Schaffen die Tätigkeit ist, bei der ich meinen Geist am Liebsten bilde, sind es doch die Abende unter Freunden, für die sich dies Bilden erst lohnt. Mein guter Freund Samuel proklamierte einmal, in seinem nicht ganz akzentfreien englisch: "I don't really think in terms of *who's my best friend*"; und so möchte ich auch diese Liste möglichst frei von Rangfolge und Favoriten verzeichnen. Denn euch umgibt alle etwas Besonderes, das ich sonst nirgends finden kann. Weder in Wien, Transdanubien noch dem Rest der Welt. Samuel, Aaron, Matthias, Gabschi, Vinzi, Emil, Niki, Fabi, Sasith, Lukas, Jeany & Paula, sowie allen Studienkolleg||innen, die mich auf dem Weg unterstützt haben. Sogar Jürgen. Ansonsten möchte ich Martin Polaschek und Helmut Wanek von der Uni Wien danken, die mit ihrer C++ Einführung die beste Vorlesung gaben, die ich je erlebt habe.

Am Wichtigsten ist es mir aber meinen Eltern zu danken. Nicht nur, weil sie wirklich äußerst geduldig mit mir waren, sondern mir auch sonst im Leben immer das Gefühl gaben, dass ich nicht nur alles erreichen kann, was ich mir in den Kopf setze, sondern auch auf die Weise, wie ich sie für richtig halte. Nicht zu selten müssen Kinder zum Erlangen eines solchen Freiheitsgefühls erst Grenzen zwischen sich und ihren Eltern setzen. Es wäre, denke ich, auch nicht zu weit aus dem Fenster gelehnt zu behaupten, dass ich ohne die finanzielle Unterstüzung derer wohl kaum genügend Zeit oder Kraft gehabt hätte, dieses Masterstudium durchzuziehen. Thomas Bernhard mag ich auch.

Acknowledgements

First of all, I would like to thank everyone in the VR Research Unit at TU Wien for their kind support, particularly Iana Podkosova, with whom I have had pleasant communication ever since the first email. She should also be credited for always keeping this thesis' greater goal in mind and never failing to provide me with new impulses. I also thank Christian Schönauer for his technical expertise and in particular for the assembly of our hardware setup. Last but not least, I would once again like to explicitly thank Hannes Kaufmann, whose trust in me goes back to my first internship in the VR Research Unit, with which I was allowed to experiment with a Spot robot and a gigantic terrestrial laser scanner.

Even though creating is my preferred activity to cultivate my mind, it is nevertheless the evenings spent with friends that make cultivating it worthwhile in the first place. My good friend Samuel once proclaimed in his not quite accent-free English: "I don't really think in terms of *who's my best friend*". Thus, I would also like to record this list as free of hierarchy and favorites as possible. Know that each of you enchants me in a way I cannot find anywhere else. Neither in Vienna, Transdanubia, nor the rest of the world. Samuel, Aaron, Matthias, Gabschi, Vinzi, Emil, Niki, Fabi, Sasith, Lukas, Jeany & Paula, as well as all fellow students who have supported me along the way. Even Jürgen. I would also like to thank Martin Polaschek and Helmut Wanek from the University of Vienna, whose C++ introductory course was the best lecture I have ever attended.

Most importantly though, I must thank my parents. Not only because they were extremely patient with me but also because they not only made me feel like I could achieve anything I set my mind to, but also in the way I deem right. It seems to me that in order to gain this luxury of freedom many other children first have to set up boundaries between themselves and their parents. I also think it would not be too far-fetched to claim that without their financial support, I would hardly have had enough time or energy to complete this master's degree. I also like Jean-Luc Godard.

Kurzfassung

Mit der fortschreitenden Digitalisierung der Bauindustrie findet BIM (Building Information Modeling) immer mehr Anwendungsfälle. Der Vergleich eines BIM-Modells mit seinem realen Gegenstück war bisher eine manuelle und oft zeitaufwändige Aufgabe. Um dies zu automatisieren, wurde das BIMCheck-Projekt initiiert. Bisher können mittels LiDAR (Light Detection and Ranging) und SLAM (Simultaneous Localization and Mapping) ein Gebäude und seine verschiedenen Räume grob mit seinem BIM-Modell verglichen werden. Aufgrund der geringen Dichte der LiDAR-Punktwolken können jedoch kleine Objekte von Interesse wie Steckdosen, Lichtschalter oder Notfallinstallationen nicht erfasst und damit auch nicht mit ihrem BIM-Modell verglichen werden.

Um diese kleineren Objekte zu erfassen, führt diese Arbeit eine Azure Kinect-Komponente in die modulare BIMCheck-Pipeline ein. Diese Komponente erlaubt die Aufzeichung, Registrierung und Verarbeitung mehrerer aus Azure Kinect Bildern extrahierten kolorierten Punktwolken in Echtzeit. Während der Großteil der Registrierung durch die LiDAR SLAM Komponente erreicht wird, wird ebenfalls eine zusätzliche Registrierung durch Perspective-n-Point (PnP) eingesetzt. Aufgrund der großen Menge an Daten, die von der Azure Kinect generiert werden, verfügt die Komponente auch über mehrere Datenreduktionstechniken, um die Punktwolken auf ihre relevantesten Daten zu filtern.

Die Leistung der Komponente wird in Bezug auf Registrierungsqualität, Datenreduktionsrate und Geschwindigkeit bewertet. Zwei Arten von Umgebungen werden verwendet, um die Registrierungsgenauigkeit zu bewerten: Ein simpler aber detailreicher leerer Flur und ein größeres Bürogebäude unter normalen Arbeitsbedingungen. Unsere Komponente zeigt robuste Ergebnisse in beiden Umgebungen, mit der Fähigkeit, Parameter je nach Umgebung und Aufnahmeverfahren noch extra fein abzustimmen.

Die Komponente überwindet viele der Einschränkungen, die mit der Registrierung und Kombination von Azure Kinect-Punktwolkendaten einhergehen. Sie wurde in C++ unter Verwendung von PCL und OpenCV geschrieben. Zusammen mit der LiDAR SLAM-Komponente kann das System auch außerhalb der Bauindustrie eingesetzt werden, etwa für die Erfassung archäologischer Stätten oder zur Erstellung von 3D-Modellen realer Objekte für den Einsatz in Virtual-Reality-Anwendungen.

Abstract

With the continued digitalization of the construction industry BIM (Building Information Modeling) lends itself to ever more use cases. Comparing a BIM model to its real-life twin has traditionally been a manual and often time-consuming task. In order to automate this, the BIMCheck project was initiated. So far, using LiDAR (Light Detection and Ranging) and SLAM (Simultaneous Localization and Mapping), a building and its various rooms can be roughly compared with its BIM Model. However, due to the sparse density of the LiDAR point clouds, small objects of interest such as power sockets, light switches or emergency installations cannot be sufficiently recognized and compared to their BIM model counterpart.

To capture these smaller objects, this thesis introduces an Azure Kinect component into the modular BIMCheck pipeline. This component records, registers and post-processes multiple unstructured colored point clouds extracted from the Azure Kinect frames simultaneously in real-time. While most of the registration is achieved by a good initial guess obtained from LiDAR SLAM, additional registration through Perspective-n-Point (PnP) is also employed. Due to the large amount of data generated by the Azure Kinect, the component also features multiple data reduction techniques in order to keep only the most relevant data while filtering away the rest.

The performance of the component is evaluated in terms of registration quality, data reduction rate and performance. Two types of environments are used to evaluate the registration accuracy: A simple but feature-rich empty hallway and a larger office complex under normal working conditions. Our component shows robust results in both environments, with the ability to fine-tune parameters depending on the environment and recording procedure.

The component overcomes many of the limitations that come with registering and combining Azure Kinect point cloud data. It was written in C++ using PCL and OpenCV. Together with the LiDAR SLAM component, the system may also be used outside of the construction industry, ranging from scanning archaeological sites to creating 3D models of real-world objects for use in virtual reality applications.

Contents

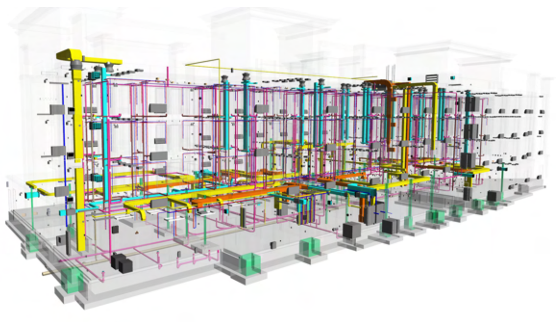
CHAPTER 1

Introduction

This thesis contributes a novel approach to integrate dense, colored point cloud data from a depth camera into a scanned-vs-built comparison pipeline in real-time. We integrate this approach in a software component that can be used as-is without any necessary training. It supports multiple environments and use-cases. Its accuracy was evaluated by comparing it both to ground truth data as well as data captured externally by a state-of-the-art scanner. An additional evaluation was undertaken to determine the biggest possible data compression rate without losing detection of objects of interest.

This work is a part of the multidisciplinary BIMCheck project. The software component proposed in this thesis is only a part of the whole project. Therefore, the following sections will provide an overview for the motivation, general goals of the BIMCheck project as well as this thesis' contribution to it.

1.1 Motivation


(a) Complex BIM model from [BIM24]

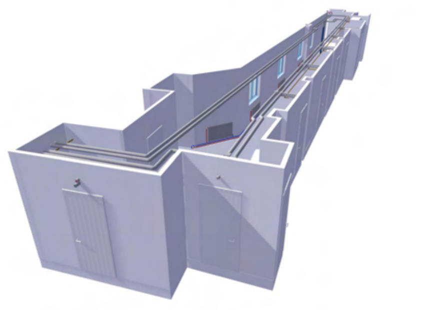

(b) TU Wien hallway BIM model

Figure 1.1: Visualization of BIM models

With the ever increasing demand for new houses and office complexes, the construction industry is looking to digitalize as much of the process as possible. The first standard for Building Information Modeling (BIM) was introduced in the early 2000s. With BIM a building's digital twin is created, which can be used in all stages of a building's life-cycle. This includes planning, construction, operation and maintenance. Furthermore, it creates a common ground and way of communication for all parties involved in the construction process. A client can easily communicate his wishes to the architect, and after receiving the plans, the construction company can easily estimate the costs and time needed for the project. Problems or misunderstandings can be detected early and the whole process can be streamlined. However, traditionally BIM has only played a significant role during the planning phase of a building. An example of BIM models can be seen in Figure 1.1.

Once a building is planned using the BIM standard, the construction company can use it as a guiding tool for constructing the actual building. Due to a variety of reasons, the final building may differ from the original plans though. This can be due to the construction company not following the plans, an architect making last-minute changes that are lost or the client changing his mind. To make sure that the changes are detected and documented, it is important to compare the actual building with the original plans. So far, this has been achieved by manually comparing the building with the plans. However, doing it this way is not only costly and time consuming but also error prone.

The described problems grow even bigger when it comes to offices, where a third party might depend on the correct installation of power sockets from day one, which would each have to be checked manually. Other small objects of high importance would include emergency installations like fire extinguishers and dampers, or the correct installation of an air conditioning system. To assist this process of verifying installations in an automated way, a system is required that can capture objects as small as a power socket and localize them in the building's digital twin, in real-time. This would allow a single person or even just an autonomous robot equipped with a scanning device to walk through a building once and automatically compare all the installations to the original plans.

The research project BIMCheck attempts to address the challenged described above. This thesis makes a part of this project. Particularly, it addresses the capturing and registration of small objects like light switches or power sockets. Proper classification of these objects is left to another part of the system. The next section will provide an overview of the current state of the BIMCheck project and how our component will be integrated into it.

1.2 BIMCheck Project

BIMCheck is a collaborative project between TU Wien, FH St. Pölten and rtech engineering GmbH. BIMCheck uses real-time sensors to produce a point cloud that captures the as-built state of the building. It then automatically compares it to the BIM plan used for the construction. This procedure includes all steps from data acquisition to presenting differences in all measurable details of a BIM model. Cost but also time savings are of high importance for the project. Scanning a single room should not take an unnecessary amount of time, i.e. be as fast as a human or mobile robot who passes through the room at a normal pace. The results of the comparison shall then be visualized to the user. The whole process shall be usable by and benefit all parties involved in the construction process.

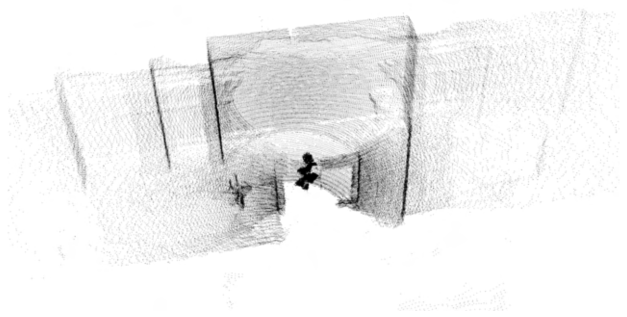
The conceptual pipeline of BIMCheck is as follows: 1. A sensor setup is carried through a part of the building, and a point cloud scan of it is created. 2. The point cloud is localized with respect to the BIM model. 3. Differences in large geometrical features are found automatically between the point cloud and the BIM model. 4. Smaller installations are detected in the point cloud and compared with the BIM model. 5. The results are measured and visualized to the user.

At the time of the start of this thesis, steps 1 and 2 have been developed, but only for a Light Detection and Ranging (LiDAR) sensor. The visualization of step 5 is implemented for the information currently extracted. We will now go into detail on how steps 1 and 2 are achieved and how we plan to expand them with this thesis, in order to accomplish step 4.

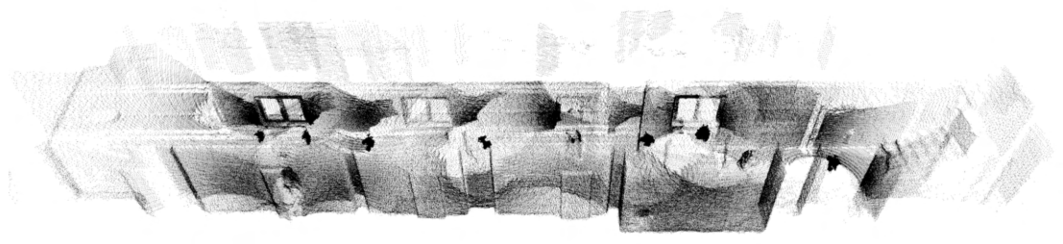
A mobile LiDAR is carried through the building either by a human or a robot. LiDARs are widely used in automobiles or robotics, as they provide an accurate 3D model of an objects surrounding in real-time. Aside from capturing their surroundings in 360 horizontal degrees, most LiDARs also measure their own movement and with that their pose inside the point cloud through the use of an Inertial Measurement Unit (IMU). For BIMCheck we decided on using an Ouster OS0 as our LiDAR (see Figure 1.2). In order for all the captured 3D points to create a cohesive whole, i.e. a 3D model of the building, the scans need to be aligned (registered) as the LiDAR is carried through a building. In theory, registering point clouds from different positions is as simple as keeping track of the LiDAR's position on each scan, which we should know from the IMU. In practice, the IMU is not as accurate as registration needs it to be, and outliers or false positives might still plague our reconstruction. In order for the registration to be more robust while still adhering to the real-time demands, a Simultaneous Localization and Mapping (SLAM) algorithm was added to the pipeline. SLAM uses as input not only IMU data but also the scanned point cloud data in order to provide a vastly more accurate position of the LiDAR in the environment. An example of the registration can be seen in Figure 1.2c.

The current work in the BIMCheck project already achieves automatic, real-time localization of LiDAR point clouds in a BIM model [Sch22]. However, those LiDAR point clouds are sparse, and while this property is beneficial or even crucial for real-time SLAM to work, small objects like power sockets, light switches or fire dampers are lost due to the lack of resolution. To capture such objects, we need to integrate a source of denser point cloud data into the pipeline for our object recognition component. We chose to use an Azure Kinect device (seen in Figure 1.3a), due to it's low price, real-time applicability and the fact that it also captures color, which may be beneficial for future use cases. Kinect-like devices also have a well-established research and development community,

(a) Ouster LiDAR OS0 device



(b) Ouster LiDAR OS0 point cloud



(c) Multiple LiDAR point clouds aligned and combined

Figure 1.2: Ouster LiDAR OS0 overview

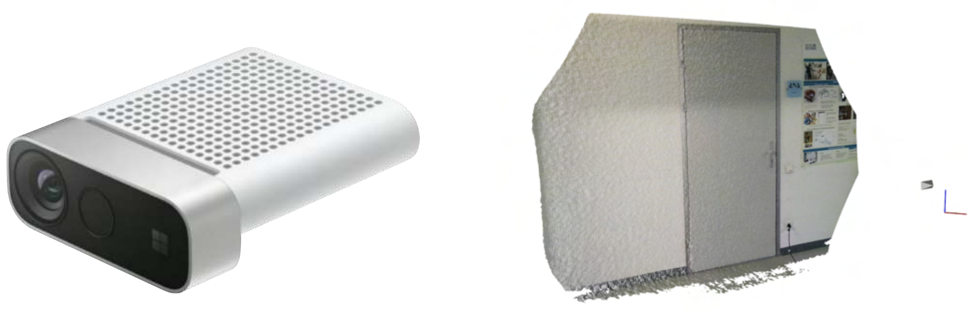
the experience of which can be used for our goals. An example of a point cloud captured by an Azure Kinect can be seen in Figure 1.3b.

1.3 Goals of the Thesis

The main research question of the thesis, related to the goal that should be achieved is the following:

The primary goal of this thesis is to develop a method for real-time capture and registration of dense depth data of Azure Kinect in order to use it for the recognition of small objects within the scanned space.

For this goal to be accomplished we need to correctly register every frame captured by the Azure Kinect device with our LiDAR point cloud. This will be achieved by first transforming frames into point clouds inside our LiDAR coordinate system. We can then synchronize each point cloud with a corresponding pose from the LiDAR SLAM algorithm. We also need to make sure that the point clouds are dense enough s.t. the



(a) Azure Kinect device (b) Azure Kinect point cloud

Figure 1.3: Azure Kinect overview

objects of interest can be distinguished, but not too dense for the memory of the system to be overloaded.

Achieving the above goal can be separated into several sub-goals:

First of all, the Azure Kinect and Ouster OS0 run at different frequencies. The LiDAR can run at 10 or 20 Hertz (Hz), while the Azure Kinect may run at 5, 15 or 30 frames per second (fps). Without reducing the LiDAR's Hz and therefore registration accuracy, the only framerate that could sync up with the LiDAR are 5 fps, at which point our depth camera would operate at only a 6th of its capabilities. Even then, there is no way to perfectly sync the two devices to capture data at the exact same time. Additionally, the SLAM system we are using runs on various IMU samples, with no way of knowing which exact point in time a LiDAR's pose represents. *We need to correctly match the poses received from our LiDAR SLAM component with our Azure Kinect frames.*

With the amount of data we are recording using the Azure Kinect, one also quickly runs into memory and storage limitations. A single Azure Kinect color frame may have a resolution of up to 3840x2160, leading to roughly 250 million points per second at full operational capacity, each requiring 3 32-bit floating point numbers for position and 3 8-bit integers for color, if stored naively. Handling and storing all of this data would exceed the memory of most computers in no time, especially if the system is scanning a whole building. *Therefore, we want to also introduce a novel approach to handle and save only the most necessary data, without losing any objects of interest.*

In order to reduce the amount of data stored, several strategies can be integrated into the system. A rigorous filter can throw away all points not belonging to one of the objects of interest defined based on image classification. But we also want to make use of the typical environment the system is devised for: empty, newly constructed buildings. Such environments typically feature a lot of empty, white walls. By using plane segmentation, we want to significantly reduce the amount of points extracted from each frame and, should the wall be needed, abstract it and store it using a simple plane equation. Another approach for reducing the data stored is to only extract or keep frames that feature a part of the building not previously captured. *We shall reduce the data stored as much as possible, without losing any objects of interest in the process.*

As will be shown in Chapter 2, accurately capturing and registering an environment using an Azure Kinect often requires schooled personnel, who are aware of the device's shortcomings. Due to the nature of cameras, every recording will feature at least some motion blur, but rotating the Azure Kinect too rapidly or moving it in an unconventional way may lead to problems in the registration as well. Since our system is devised to be used by non-technical personnel, we want to make sure that the system is as robust and generalizable as possible by pre-filtering frames by their sharpness. *Our system shall be robust enough and automatically correct the most common pitfalls and limitations of the Azure Kinect.*

While we do not see any immediate use for color in the BIMCheck project, we want to incorporate it into the system nonetheless. At the very least, it can provide a much nicer looking visualization, which is always good to have for a presentation. Additionally, future object detection or post-processing registration algorithms may benefit immensely from the color information.

1.4 Structure of the Thesis

WeWe will now provide an overview of the structure of the thesis.

In Chapter 2, we will give an overview of the history of research on depth cameras and their registration algorithms up until the current state-of-the-art. We will also show how its shortcomings were tackled over time, as well as its latest use-cases when combined with a BIM model.

In Chapter 3, we will go into the details of the existing BIMCheck pipeline, how our component fits into this modular system, as well as how we plan to design our component's pipeline in order to achieve the goals set in the previous section.

Chapter 4 will go into detail on the technical implementation and the algorithms we came up with for our component. It will also describe how said algorithms are interlinked and run simultaneously in order to achieve real-time performance.

Our system will be evaluated in Chapter 5. The system was tested in 2 different environments against their BIM counterpart, one of which has additional ground truth data from a Leica LiDAR scanner much better than the one used in our project. Aside from accuracy measurements, it will also show the achieved compression rate and runtime statistics.

Chapter 6 provides a usage guide for users of the system, as well as describe every parameter that can be changed in the configuration. As stated before, we think to have already found a general set of parameters that works well in the tested environments without the need of any prior training for its users. However, domain experts may still achieve better results by tweaking the parameters or system using our guide.

Finally, Chapter 7 will conclude the thesis and give an outlook on future work that could further improve the BIMCheck project or the field of computer vision and robotics in general.

CHAPTER 2

Related Work

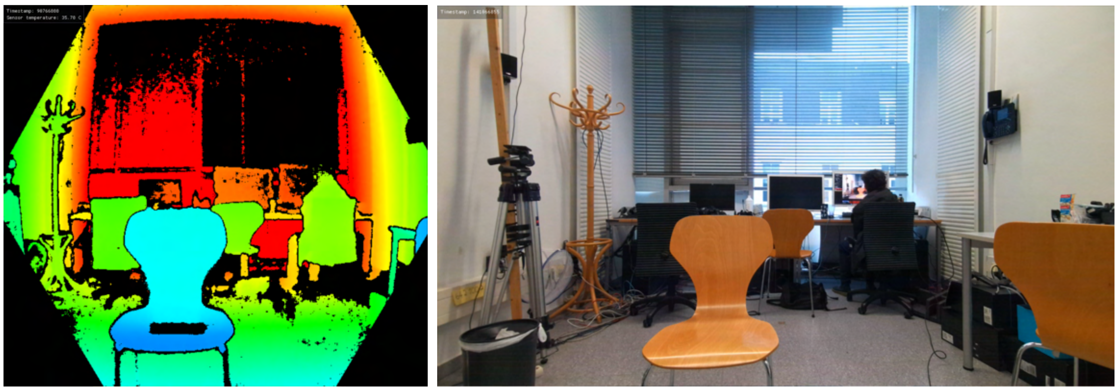
A lot of research has been undertaken in the field of depth camera (RGB-D) based point cloud registration. Many papers since 2010 are particularly focused on the Microsoft Kinect and its various successors. This was due to its unmatched price-performance ratio, making RGB-D cameras accessible for all kinds of research or application budgets. While a more expensive camera may produce better results, research for Kinect is sprawling particularly because people want to find ways to mitigate its shortcomings, as the next best RGB-D camera is still an order of magnitude more expensive.

This chapter will first start with a brief overview of the Azure Kinect device, its and its predecessor's history in research, and how its shortcomings were tackled over time. However, some shortcomings seem destined to stick around for much longer. For this reason, the field of cross source point cloud registration has been a focus of computer vision research as well. Section 2.3 will summarize the motivation, device combinations and state-of-the-art approaches in this field, as well as the challenges and limitations that come with it.

Finally, Section 2.4 will briefly show how the Kinect family of devices have already been used with the BIM standard in the past. To the best of our knowledge, we did not find any research that combines the Kinect with BIM in the way we intend to, at least not within a fully autonomous, real-time scope. Nevertheless, it will help us outline many of the common pitfalls the Azure Kinect device as well as usage in a BIM setting brings with it.

2.1 Azure Kinect

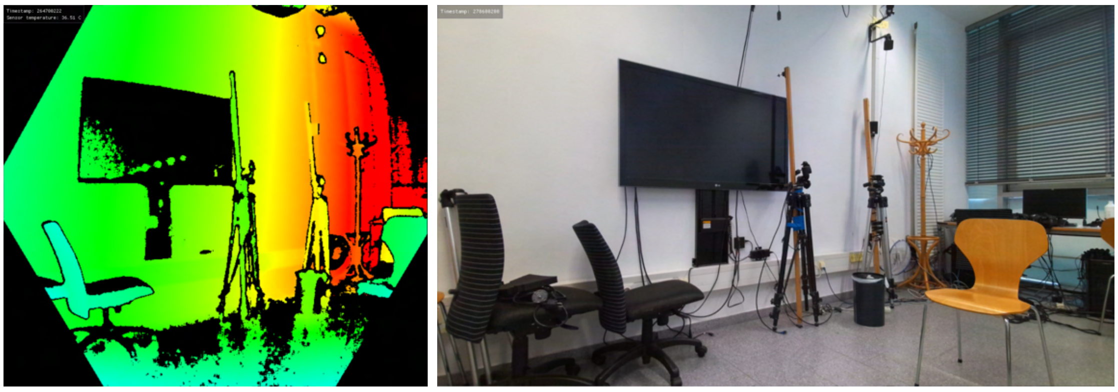
The Azure Kinect can record both RGB as well as depth image data. However, it does not accomplish this with the same sensor but rather two separate cameras. The color camera is like any typical RGB camera with a resolution of up to 2160p for 16:9 aspect



(a) Depth camera output (b) Color camera output

Figure 2.1: Example of Azure Kinect camera output

ratio and 3072p for a 4:3 aspect ratio. In order to compute depth values, the Azure Kinect uses the Amplitude Modulated Continuous Wave (AMCW) Time-of-Flight (ToF) principle by emitting modulated illumination. The time it takes for the light to return back to the sensor is then converted into a depth value.

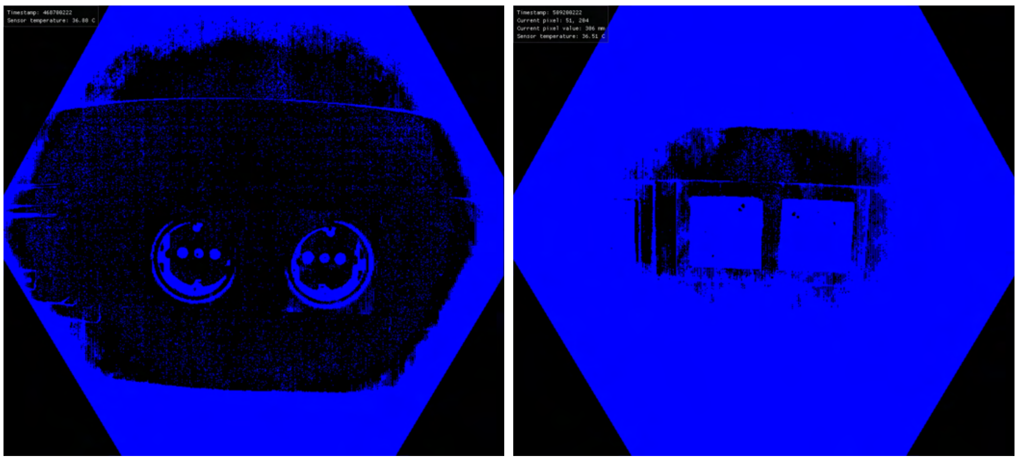


(a) Depth camera output (b) Color camera output

Figure 2.2: Example of Azure Kinect depth camera failures. Both the reflective surface (TV) as well as the intersection between wall and floor are not picked up by the depth camera. Black pixels are missing depth values.

The Azure Kinect calculates depth values for a resolution of less than 640x480 pixels, far lower than the highest possible color resolutions. As can be seen in Figure 2.1a it is even below 640x480 as its corners are cut off. Furthermore, it does not capture the whole area the color camera captures, as Figure 2.1a stops right before the clothes rack on the left, and before the chair seen in 2.1b on the right.

In addition to that, even in its operating field, the Kinect family of devices might be unable to compute a depth value for every pixel, at which point a value of 0 is assigned as well. This can stem from the infrared light being reflected on a reflective surface or an area simply being too far away for the ToF to register. Examples of this can be seen in Figure 2.2. Azure Kinect also fails to compute depth values if objects are too close. Figure 2.3 shows an example of this, where the depth camera is unable to compute depth values for almost all pixels of 2 power sockets when only 20cm away. It gets better at 30cm, but our testings show that at least a distance of 50cm is needed for the depth camera to compute depth values for all pixels.



(a) Capture from 0.2 meters away (b) Capture from 0.3 meters away

Figure 2.3: Example of Azure Kinect depth camera not being able to compute depth values for close objects. Black pixels are missing depth values. Note: Although the human eye can process the scene more easily from (a), (b) is preferred for computer vision tasks as a computer can easily detect even the slightest changes in depth (i.e. change in blue value)

2.2 Point Cloud Registration using Kinect

The first ground-breaking algorithm in aligning multiple Kinect frames was Kinect Fusion [NIH+11]. While its results were vastly surpassed by the research that followed, many of its approaches continue to show up in the following work, as well as our own thesis. Therefore, we will go into more detail on the specifics of its pipeline, as most of it will reappear throughout the chapter.

1. Depth map to point cloud conversion

   Kinect devices and RGB-D cameras in general obtain depth information in addition to a 2D image. The original Kinect does this by using the structured light technique, where a known pattern (e.g. a grid) is projected onto the scene. Depth information can then be extracted from the deformations (i.e. abbreviations) of the expected grid. The Azure Kinect we use estimates depth by emitting light and calculating the time of flight it takes for a reflecting light ray to return to the sensor. In order to convert a calculated depth value for a given pixel into a position in 3D space, we also need the intrinsic camera parameters, which take into account the focal length, the principal point and the skew of the camera. It is important to note, that the resolution of the color and depth image captured by a RGB-D camera may not necessarily be the same. Often, the color image can be of a higher resolution, resulting in the need of interpolation of depth values or other measures to match the higher resolution of the colored image.

2. Registration using Iterative Closest Point (ICP)

   Iterative Closest Point [BM92] is a well-researched algorithm devised to align 2 point clouds given a good initial guess for the transformation of the source point cloud to the target point cloud. It works by minimizing the distance between two sets of points, by iteratively finding the closest point in the target set for each point in the source set. It will then iterate and adjust the transformation to minimize the point pair's distances until it finds a local minimum. Normals for all points can vastly improve the algorithm, due to the additional information provided for finding 2 matching point pairs. Kinect devices do not support recording of normals, but can be estimated quite accurately by various libraries [RC11] [Bra00], especially if the camera's or sensor's position is known. While ICP is perhaps the most widely used algorithm for point cloud registration, one must keep in mind its limitations: First, due to its goal of minimizing the distance between a point pair from both clouds, the input clouds' sizes should roughly be the same. Secondly, due to it being a local optimization algorithm, it is sensitive to the initial guess of the transformation. Without a good initial guess the algorithm will convert very quickly, seemingly without aligning anything. Kinect Fusion creates surface vertices with estimated normals from its depth frames, then applies ICP with the previously connected surface. Since ICP is performed every frame (i.e. with 2 point clouds that are roughly aligned anyway), the initial guess for each consecutive ICP can just be the pose estimation of the last, stacking the two point clouds on top of one another.

3. Volumetric fusion and surface representation using Truncated Signed Distance Function (TSDF)

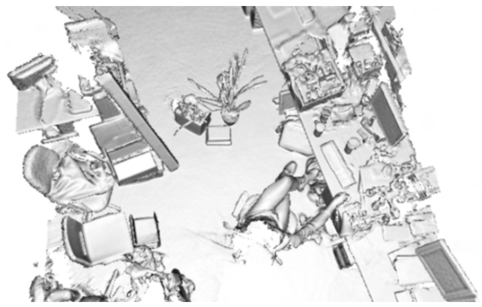
   After the generated surface is aligned with the overall scene, the surface vertices are then added to a TSDF volume [CL96]. This is a 3D grid, where each cell contains a truncated signed distance value, representing the distance to the nearest surface. The value is positive if the position in the grid is in view of the camera (i.e. in front of an object) and negative if a value is occluded by a surface. This is a much more memory efficient way to represent 3D scenes, as volumes with finer detail than the grid size would allow can be accurately reconstructed using only the TSDF values.

4. Raycasting and next iteration

   As a final step, but more importantly first step of the next iteration in the Kinect Fusion algorithm, raycasting is performed on the resulting TSDF volume to create a new surface mesh. This is done to get a less noisier and also sparser version of the volume, which is ideal to align it with the next frame using ICP.

Kinect Fusion [NIH+11] provided and still provides a great starting point for real-time registration using RGB-D cameras, and libraries like OpenCV [Bra00] and Point Cloud Library (PCL) [RC11] even feature implementations of it. But its limitation have also been a focus of research over the years [WKJ+15]. First, in order to align 3D points using ICP, the points must have some sort of characteristic to them, otherwise there is nothing to distinguish them by. A typical example of where this becomes a limitation would be a wall with nothing on it. While 360 degree sensors are not as susceptible to this due to getting a point cloud of the entire scene, a camera is always locked to its view frustum. Secondly, due to constraints from the TSDF representation, Kinect Fusion only works in a small space of a few m^3 around the origin of initialization [WKJ+15].

Whelan et al. improved on the last constraint with their Kintinuous [WMK+12] algorithm. This is achieved by letting the TSDF origin move over time. Older areas, which are not part of the TSDF surface anymore are extracted and represented using a pose-graph, where each pose stores a surface slice. While Kintinuous is therefore able to capture scenes much larger than the original Kinect Fusion implementation [NIH+11] as seen in Figure 2.4, it still needs objects to be in view of the camera to be able to align them. Also, with the incorporation of larger areas, they also inherited the problem of loop closure, which is one of the most common in point cloud registration. Loop closure stems from accumulated drift, either from the sensor or algorithms used. Drift can be negligible when moving in a single direction or operating in small room, but as soon as a sensor re-enters an area already registered it becomes a problem. Because the same walls that are already present in the scene may be aligned to a different position due to the error from the accumulated drift.


(a) Kinect Fusion [NIH+11]

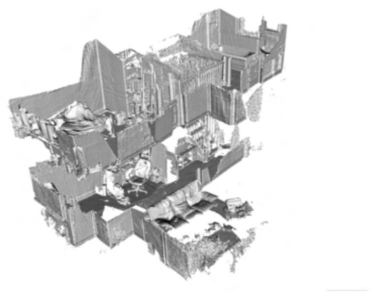
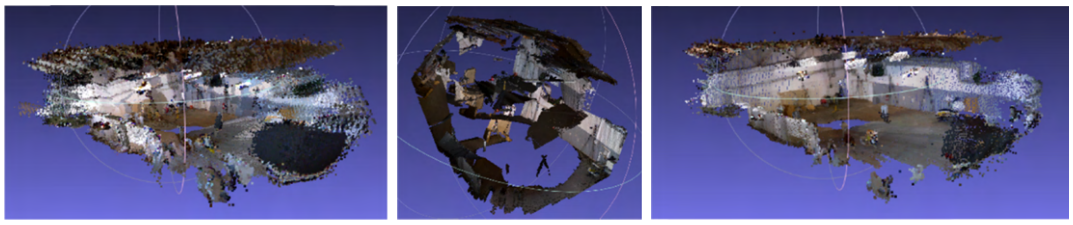

(b) Kintinuous [WMK+12]

Figure 2.4: Comparison of original Kinect Fusion and Kintinuous results

Whelan et al. tackled loop closure in a follow-up paper [WKJ+15], where they split their pipeline into a frontend and a backend. The frontend practically does what Kintinuous does, it tracks the position and extracts surfaces as the camera moves through a scene. Simultaneously, the backend keeps track of the generated map and optimizes the poses as well as the map like a traditional SLAM approach. For that, features are extracted using Speeded Up Robust Features (SURF) [BTVG06] and matched using a bag of words model. If a possible loop detection threshold is reached, Random Sample Consensus (RANSAC) [FB81] is performed on the last few frames to estimate a transformation. If this transformation is still inside the loop detection threshold, ICP is performed, using RANSAC's transformation as an initial guess. If that ICP result is below an error threshold, the loop is closed using relative transformations for every pose position between the 2 frames recording the same part of the map. The backend also optimizes vertex positions using a space deformation graph based on Sumner et al.'s embedded deformation [SSP07]. Still, due to the reliance of feature detection, even this approach is not robust to featureless areas. Also, due to loop closure only happening in the backend, the frontend (i.e. Kintinuous [WMK$^+$12]) may still operate on loop closure problems, due to the backend not catching up quickly enough.

While all the previous papers were optimized for Graphics Processing Unit (GPU) usage, Mur-Artal and Tardós went for a different approach with ORB-SLAM2 that worked on standard Central Processing Unit (CPU)s in real-time [MAT17]. Instead of using ICP, their alignment is based on ORB features [RRKB11] as well as bundle adjustment [TMHF00]. The latter is performed in three consecutive steps: First, motion-only bundle adjustment is performed in order to optimize the camera pose in the tracking thread. Then, a local bundle adjustment optimizes co-visible key frames and their map points. Those key frames are tested against fixed other key frames featuring the same points. Finally, a global bundle adjustment works on the whole map in a separate thread, which is particularly responsible for loop closure. A bundle adjustment working on the whole map cannot work in real-time on current hardware. But, since they wanted to keep recording and registering while also detecting and closing loops, the global adjustment is run only on past key frames and restarted should a new loop be detected while it is still running. Key frames newer than the last global bundle adjustment are then transformed according to their previous reference key frames. ORB-SLAM2 [MAT17] outperforms the ICP methods mentioned before, though their tests using monocular cameras were only conducted in single room environments. If used with a stereo set-up however, their system can even register large urban areas from a car. This might be due to scale drift not occurring or being significantly mitigated on stereo set-ups. But the need for detectable features remains.

In 2022 Pan et al. also proposed a new method for RGB-D registration [PHY22] without using ICP to tremendous results. Instead, They utilized Perspective-n-Point (PnP). Their pipeline is described as follows: First they filter the depth map and extract a point cloud with estimated normals from it. Next, 3D-2D EPnP [LMNF09] pose estimation is used instead of ICP to register the extracted point cloud. PnP is a well known algorithm with a lot of research. It estimates a camera pose from matching pairs of the camera image's pixel positions and known 3D world coordinates. A common problem that arrives from using it is that for any given number of pairs, there is always more than one solution. So the problem becomes finding the correct solution from the calculated ones, which they acquire after some algebraic manipulations from a probability density function. Their registered point cloud is then stored inside a TSDF [CL96] volume, which is separated and extended if bounds are reached. A bag of words model is further utilized to reduce the memory footprint of the TSDF volume. Finally, similarity matching like in Kintinuous [WMK$^+$12] and a global bundle adjustment [TMHF00] [DNZ$^+$16] is used to detect loop-back and achieve loop closure. Their algorithm significantly outperforms both Kintinuous [WMK$^+$12] as well as ORB-SLAM2 [MAT17], even succeeding where those algorithms fail, as seen in Figure 2.5. The computation time is also less or about the same as ORB-SLAM2, i.e. real-time applicable. But their tests only include datasets of large and dense scenes, with the authors stating that this is the environment the algorithm was conceived for. [PHY22]. It seems as though using just an RGB-D camera the need for dense or complex scenes remains, which we cannot allow for our use-case, as we expect most of our scanning environments to be newly built, empty buildings.



(a) ORB-SLAM2 (b) Kintinuous (c) EPnP

Figure 2.5: Comparison of state of the art RGB-D registration given in [PHY22]

2.3 Cross Source Point Cloud Registration

Due to the limitations in registration when using only depth cameras combined with the desire to incorporate denser point clouds into applications, the research field of Cross-Source Point Cloud (CSPC) emerged. While registering point clouds from different sources solves these two problems [SGS$^+$18], it also comes with its own set of challenges. Unlike e.g. a stereo set-up featuring the same camera twice, the point clouds stemming from different sources can be vastly different in character. These differences in characteristics (scale, density, noise, etc.) make most traditional registration algorithms fail, as they are designed to only work with input data of similar type, resolution and/or rigid transforms [HMZ23]. The field of CSPC was surveyed by Huang et al. both in a general point cloud registration survey in 2021 [HMZA21] as well as exclusively in 2023 [HMZ23]. Using those as a guidance, the following section will describe the most important papers since the field's inception, as well as add state-of-the-art methods that arose after the surveys were first handed-in.

CSPC was first tackled in 2014 when Peng et al. [PWF$^+$14] introduced a method for combining a large street-view LiDAR cloud with a small one retrieved from SfM. SfM produces similar point clouds to the one obtained from RGB-D cameras, but instead of using depth information, it estimates a 3D scene based on a set of 2D images obtained

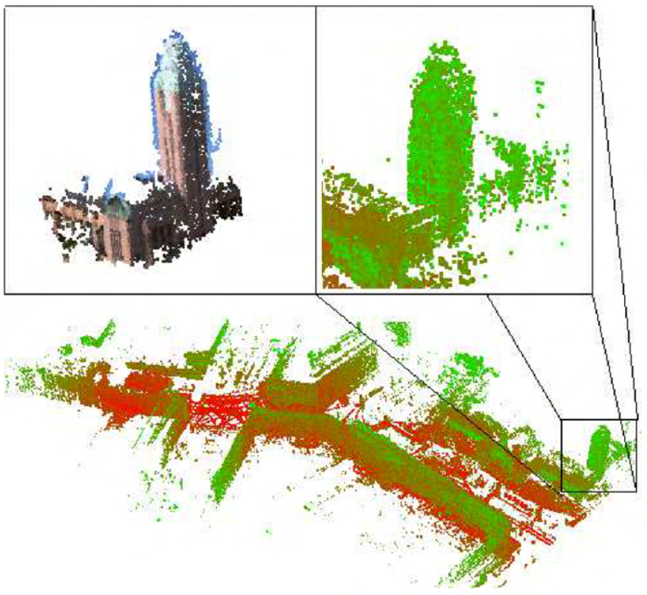


Figure 2.6: Cropping of LiDAR point cloud (top right) to register Structure from Motion (SfM) cloud (top left) using ICP [PWF+14]

from an RGB camera alone. In order to use ICP to align the two point clouds, they first pre-processed the LiDAR cloud to crop out specific regions from it, to get them to a similar size as the SfM cloud. Only then are ESF descriptors obtained for each of the regions as well as the SfM cloud. Using those descriptors, the top 10 matched cropped regions are aligned using ICP, with the least error applied to the SfM cloud. While their method is not real-time applicable, it showcased one way to pre-process point clouds of different sources in order to use them with ICP. A visual representation of their method is shown in Figure 2.6.

Another way to pre-process point clouds to make them more suitable for ICP was proposed in 2017 by Tazir et al. [LGC+18]. Instead of simply matching points to points, they only match points representing a surface in both clouds in their proposed CICP (Cluster Iterative Closest Point). As the name suggests, this is achieved by clustering points in both clouds around a shared radius. Aside from handling a wide variety of different scenes robustly, they also showed a much faster convergence speed than traditional ICP approaches. As they show in their experiments, their method also achieves robust results for same-sensor registration, while converging about 3 times faster.

Huang et al. transformed the registration problem into a graph matching problem [HZF+17]. The point clouds are segmented into many super voxels. Each super voxel's center point and the relations between them are regarded as nodes and edges of a graph respectively. A novel graph matching algorithm then matches the two segmented point clouds, with ICP further refining the registration result. They state the limitations as relying on segmentation for solving the density problem, as well as the fact that "only two points are considered to constraint the graph node correspondence search" [HMZ23]. Huang et. al then iterated on the last constraint by allowing more neighbor correspondence [HFW+19]: Triplet correspondence using tensors is instead applied and a power iteration algorithm matches the two segmented point clouds. While this method is much more robust then the previous, the cost of computation is increased dramatically and grows exponentially with the number of super voxels.

In 2016 Huang et al. also proposed a Gaussian Mixture Model (GMM) to replace ICP [HZW+16]. This utilizes the statistical property of the point clouds and turns the registration into a GMM recovering problem. For that, the point clouds are first down-sampled to solve the density difference problem. Their approach yields better results than contemporary ICP methods, but only if the point clouds are already scale adjusted. If they are not, their following work [HZW+18] improves on handling scale differences of 0.5 to 2 times scale difference by first estimating an affine transformation matrix between the two point clouds. While it is fast and robust for the given scale differences, its downside comes with still relying on a mandatory sub-sampling as a pre-processing step. This is an issue because sub-sampling is hard to generalize for different environments and point cloud types, although recent research into deep learning methods show promising results of solving this problem [YEK+20] [HLZ+22].

While deep learning descriptors still struggle with CSPC registration problems [HMZ23], Liu et al. [LLW+21] proposed a 2D-3D Generative Adversarial Network (GAN) with promising results for this field of research. Cross domain feature descriptors are detected in both the 2D images and 3D point clouds, allowing the 2D images' information to be correctly aligned in the 3D scenes. Another approach would be utilizing deep learning for correctly predicting transformations, however, no research into predicting transformations in CSPC registration have been undertaken [HMZ23]. Of course, the drawback for each deep learning approach is its limitations when used on unseen (i.e. unlearned) data. As long as there are not many differing datasets, the results will probably lack behind more conventional approaches, at least in generalizability.

To the best of our knowledge, the state of the art in CSPC registration was achieved by Pang et al. [PLLZ23]. In their paper, they register different point clouds obtained from satellites. Their approach incorporates filtering of noise and outliers, building facade points (especially on the edges) through density clustering and finally performing a coarse alignment using a Principal Component Analysis (PCA). Their algorithm can align point clouds even if they are completely misaligned as seen in Figure 2.7. While their performance is not real-time applicable for our component, it is also twice as fast as a similar use-case [LQ22] not even two years older.

With the advancements of modern LiDAR systems, the problem of density matching might disappear completely. Giammarino et al. [DGGB+23] recently proposed a bundle adjustment method that produces great results for both modern LiDAR systems (i.e. ones producing dense point clouds) as well as RGB-D cameras. Still, as long as there is a

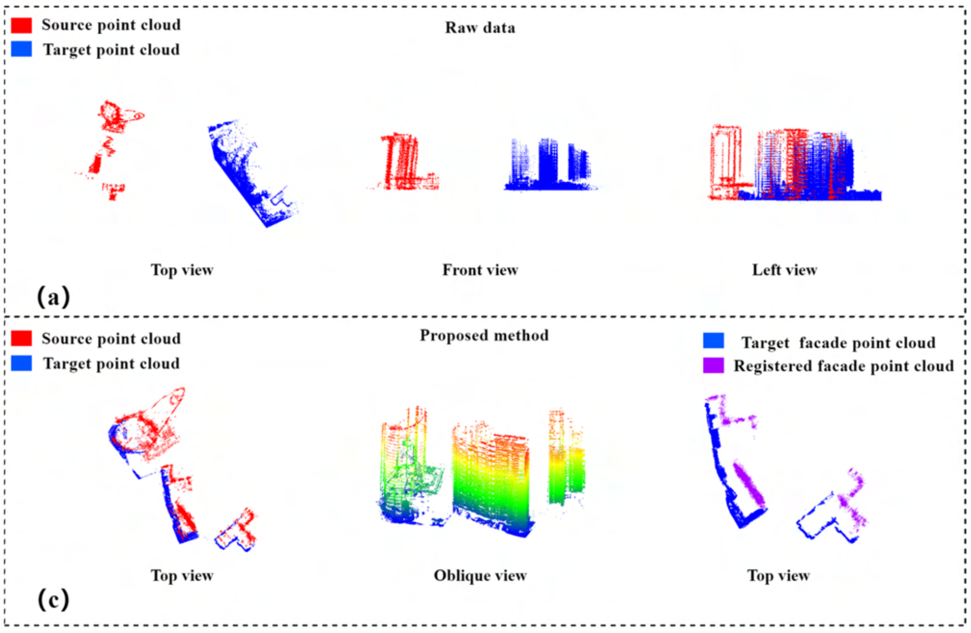


Figure 2.7: Registration of point clouds from different satellites [PLLZ23]

financial reason for sticking with older LiDARs, such as we have, the problem of CSPC registration will accompany it.

2.4 Kinect BIM Integration

Integrating Azure Kinect or other RGB-D cameras into BIM use-cases has been a focus of research over the past decade. Ali et al. [ALP20] devised a LiDAR-Kinect system for near real-time monitoring of BIM construction progress, relying on two depth sensors on top of LiDAR for positioning in order to detect rather large objects. Their procedure is as follows: First, they assign a position for the laser scanner in the BIM area using their positioning tool. They then direct it towards target objects on the construction site and connect two Kinect cameras to their Rhinoceros software [Rob] which in turn registers them to the LiDAR data. An additional three point cloud libraries are used to manage the point cloud data and convert it to point, color and GPS data. Afterwards this is sent as-is from the job-site to the development office leading to near realtime performance. Compared to our proposed system, their system is not designed to be used fully autonomous, as it requires a user to position the laser scanner and direct it towards the target objects.

Haouss et al. [EHMR$^+$22] use only an Azure Kinect device for their comparison with a BIM model. The different frames are registered using ICP, but its parameters need to be adjusted according to the scene. They also note that the trajectory of the operator moving the Kinect around the scene vastly affects the accuracy of the 3D reconstruction. It is even mentioned that for different scenes, different approaches in trajectory yield better results. They also incorporate automatic segmentation of walls, floors and ceilings by using a histogram of altitudes, while manually extracting doors and windows for the BIM model. For better automatic detection, they incorporated a bilateral filter for the depth images of the Kinect. This reduces noise and enhances sharp intensity changes, while blurring smaller ones.

Another recent use case that yields itself for using Azure Kinect with BIM models was given by Huang et al. [HWF+22] where they detected small holes in drainage pipelines. Due to the nature of the task, they could avoid errors caused by irregular operation by keeping the center line of the depth camera consistent with the center of the pipeline. Like the previously discussed paper they also de-noised the resulting point clouds using curvature constraint de-noising [ZY19] and region based growing segmentation [LJQ20] algorithms. For segmentation the authors used a RANSAC implementation based on cylinder model segmentation, which can extract a cylinder model of the pipe even from a noisy point cloud. Finally, damage in the pipeline is detected using Euclidean clustering [CSK05].

2.5 Summary

In this chapter we reviewed the related literature for our thesis. Our conclusions are as follows:

- Section 2.2 showed that while the Kinect family of devices and RGB-D cameras in general have had quite a bit of research done with them, limitations like the need for some sort of characteristic in the captured images remain. Therefore, for a generalizable solution such as ours, a combination of different sensors (i.e. LiDAR and Azure Kinect) is necessary.
- Section 2.3 showed that to combat those limitations comes with introducing new ones as well. Traditional registration algorithms like then ones in section 2.2 fail if two point clouds are of different characteristics. A good example of this is ICP failing to produce any good results if the point clouds are of different densities.
- Finally, reviewing current BIM and Azure Kinect integration in section 2.4 showed that while the combination has been used in the past, it was only ever in very specific use-cases. When used alone, ICP parameters might need to be adjusted according to the scene and without a LiDAR to help with positioning, the trajectory of the operator moving the Kinect around the scene vastly affects the accuracy of the 3D reconstruction. The only use-case that was close to our own was the one by Ali et al. [ALP20], which was far from being fully autonomous.

Since we already have a highly accurate, autonomous LiDAR localization module [Sch22], we propose attaching the Azure Kinect on-top of our LiDAR to correctly align the point clouds from the Azure Kinect with the LiDAR point clouds, that can work in any environment, without the need for any trained personnel. The last part will be aided by only using the clearest frames captured (i.e. with the least motion blur). Finally, registration accuracy of the LiDAR shall be improved using Kinect Fusion (KinFu) and PnP. The system architecture to achieve this in real-time will be described in the next chapter.

CHAPTER 3

System Design

This chapter describes the design of our method for integrating dense Azure Kinect data into the planned-vs-built comparison pipeline of the BIMCheck project. Our method is implemented as a standalone software component that is embedded in the BIMCheck architecture. We will refer to it as *RGB-D data component* from this point forward.

Section 3.1 will start by giving a brief overview of the BIMCheck architecture as well as our *RGB-D data component* and its integration into the BIMCheck project. Section 3.2 will list the requirements for the *RGB-D data component*. These requirements stem from the general BIMCheck requirements [Sch22] as well as our own goals and sub-goals declared in Section 1.3. Finally, Section 3.3 will explain the methods used to design the *RGB-D data component*, such that it conforms to those requirements and can be integrated into the BIMCheck architecture.

3.1 General System Architecture

In this section we will give an overview of the BIMCheck project's overall architecture as well as how it operates as a modular system. We will describe in detail two of its components that directly influence the input and output of our *RGB-D data component*. This will lead into the general design of our own component and its integration into the BIMCheck architecture.

3.1.1 BIMCheck System Design

The BIMCheck architecture is designed as a modular system made up of multiple independent components. That means, that each of its component is responsible for one part of the system (or processing specific inputs to specific outputs). They communicate and provide their data with each other over network. A graph of the proposed complete

BIMCheck architecture and the communication between its components can be seen in Figure 3.1.

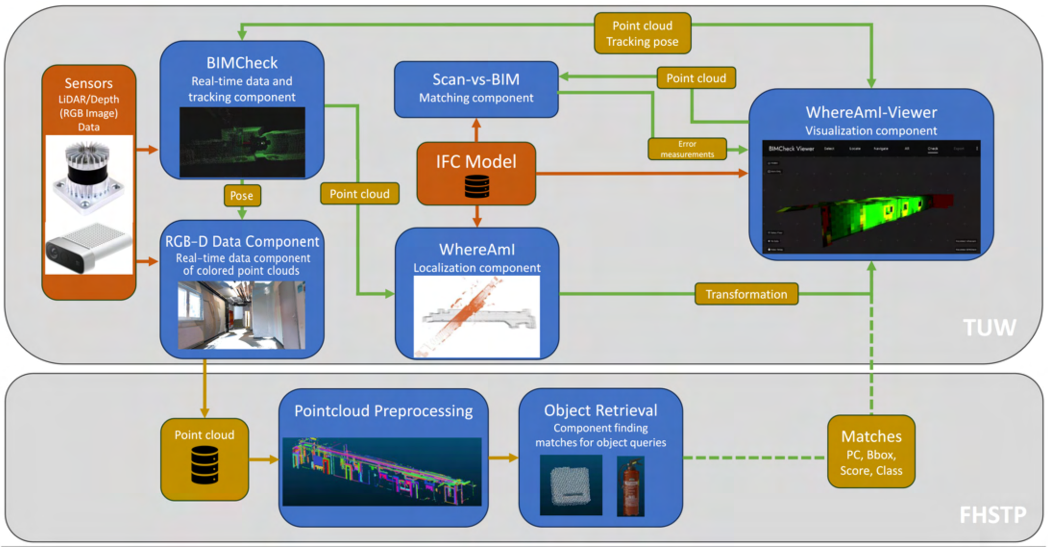


Figure 3.1: Proposed architecture of BIMCheck. LiDAR data is captured by the LiDAR sensor and sent to the BIMCheck component. BIMCheck estimates the LiDAR's pose and sends it to our *RGB-D data Component*. Our component uses the pose to refine its own registration of the input Azure Kinect data. Registered RGB-D data is then sent into the object retrieval part of the pipeline. The WhereAmI component aligns the LiDAR's point cloud from BIMCheck with the BIM (IFC) model. Matches and deviations from this as well as the Object Retrieval component are shown in the WhereAmI-Viewer.

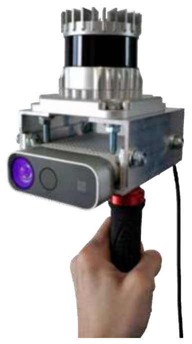
The heart of the BIMCheck system is the live tracking of an Ouster OS0 LiDAR through an environment. Through tracking the LiDAR's position with 6 Degrees of Freedom (DoF), its point clouds can be registered to form a sparse 3D model of the building. What is special about this component, is that it can run completely independent of the other ones. If one would only want to extract a 3D model of an environment (e.g. for manually comparing it with a BIM model), running just this component would suffice. The component can also be run on a recording of the LiDAR data stream. It should be noted that this component requires a license of KudanSLAM [Kud22] to work and is therefore proprietary. The LiDAR's current position and the acquired point cloud map can then be accessed through the network.

The acquired point cloud map is then used by the *Localization component*, in which it is registered to a corresponding BIM file. This component was implemented as the main focus of Schaub's thesis [Sch22]. The position of the LiDAR point cloud inside the BIM model can then be accessed through the network and is returned to the component that requested it. This feature is used by the *Scan-vs-Built Visualization* to display the similarities and differences to a user.

In order for the *Scan-vs-Built Visualization* to visualize finer differences like the correct placement of light switches or emergency installations, we first need to detect them inside a point cloud. For this reason, the object retrieval pipeline was added to the BIMCheck project. It achieves the detection of small objects by first pre-processing a point cloud to extract everything that is not a surface. Then it compares the pre-segmented parts of the point cloud to the objects of interest defined, and returns matches as the final output to the *Scan-vs-Built Visualization*. For this to work, the point clouds need to be of a much finer detail, which is where our proposed *RGB-D data component* comes into play.

3.1.2 RGB-D Data Component and Integration into BIMCheck

This section will explain the details of the software integration of the *RGB-D data component*. Our physical setup is achieved by attaching both sensors to a metal casing as seen in Figure 3.2. By measuring the offset between the centers of the sensors, our component can use the LiDAR pose estimation to correctly position Azure Kinect frames.


(a) LiDAR and Azure Kinect rigidly attached to a metal casing

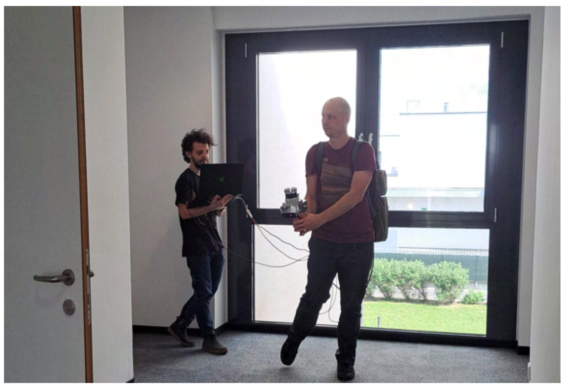

(b) Recording using our hardware set-up. A second operator may immediately view the results to secure a quality recording

Figure 3.2: Our LiDAR and Azure Kinect hardware set-up

We state the input and output of the *RGB-D data component* as follows: the component shall record an Azure Kinect video, extract 3D points from its frames and register (align) them to the point cloud obtained from the *LiDAR component*. To do this, it needs as additional input at least the LiDAR's position inside the environment. The output of the *RGB-D data component* is a point cloud obtained from Azure Kinect depth frames which are correctly aligned in respect to each other and correctly positioned relatively to the sparse point cloud map of the *LiDAR component*. These output registered point clouds should be reduced as much as possible to cut down network latency for further processing inside the object retrieval pipeline of the BIMCheck project. In order for our component to achieve this and for it to provide any benefit in the grand scheme of the BIMCheck project, our component must not lose any of the objects of interest during the reduction.

Our component was designed for integration into the BIMCheck system from the start. This meant, that many of the registration tasks for Azure Kinect's point cloud data could be cross-referenced, or left to other components of the system. How much we could actually rely on both our own localization and the system's LiDAR positioning component only became clear during the implementation and testing phase. We have decided on relying on the LiDAR SLAM position estimation for the most part, which will be discussed in Section 3.3 and evaluated against other set-ups in Chapter 5.

Due to BIMCheck's modular design, a new component like ours can only connect itself to the components it needs to read data from in order to run its own algorithms. In our case, if we just wanted to align the dense colored point clouds extracted from Azure Kinect to a complete 3D model of a building without doing any BIM comparison, one would only need to run the the LiDAR tracking component. The LiDAR's position can then be used to track the Azure Kinect device over greater distances and its point cloud can further help to register or validate Azure Kinect's captures. Even more, if a user is confident in the component's registration algorithms independent from LiDAR, it might even be run without it.

BIMCheck was designed not only to run live on-site, but also in an offline mode on recorded data streams. Therefore, the *RGB-D data component* also supports two modes, one for real-time processing and simultaneous recording and one that works on a recording. During recording, real-time performance is important for direct visual feedback. The processing of a recording can also be run in real-time but different parameters can increase the accuracy if processing time is not a concern. In the processing of a recording, we might even extract and process point clouds from each frame one by one to process every last batch of data. But even though the *RGB-D data component* will always record and save all frames for detailed processing later, the Azure Kinect, like all cameras, can still suffer from motion blur. In order to make sure every corner of a room is captured accurately, a real-time visualization of its own is featured in the component as well.

3.2 Component Requirements

This section collects requirements for both our methods and our *RGB-D data component*. Since our component is designed to be embedded into the larger BIMCheck project, some requirements are derived from the requirements of the BIMCheck project described in Section 3.1. We will only go into detail about BIMCheck's requirements relevant to our use-case. A full list of all BIMCheck requirements can be found in Schaub's thesis. [Sch22]. Other requirements are directly derived from the goals and sub-goals described in Section 1.3 as well as the integration into BIMCheck.

The requirements for the *RGB-D data component* are as follows:

- Low memory footprint

As discussed in Chapter 1, the Azure Kinect device we are using is able to capture massive amounts of points up to 30 times a second. As a first and most essential requirement, our component shall not use more Random Access Memory (RAM) than is available on a typical modern, battery powered computer. It shall definitely never need more storage memory than is available on stated devices. Therefore, we need to be able to process only the most necessary data and free up space of other data it as soon as it is not needed anymore.

- Recording at maximum capabilities

  Even though we may not process all the data we capture, we want to save the recording in its full detail in case we want to re-run our system with different parameters. This means that no matter how much resources the rest of our component (or the BIMCheck system in general) uses, highest priority should be given to capture the environment at Azure Kinect's highest capabilities.

- Performance fast enough to avoid stalling the pipeline

  As will be seen in Section 3.3, our component handles multiple point clouds in a linear pipeline simultaneously, going from capture over registration to data reduction. While we cannot expect our component to be real-time in the strictest sense of the word (i.e. fully process the point cloud the second it is captured), we need to make sure that our component can handle a full environment without any component stalling the pipeline. That means that each step of our pipeline should only take as much time to process a part of an environment as it takes the camera operator to move to the next part of the environment.

- No need for manual input in any part of the component

  Of course, every part of the component must allow a fully autonomous usage. This is stated explicitly here, because other research included extracting doors and windows manually [EHMR+22]. Other, more traditional pipelines also include manually registering (or manually improving registration), which is also something we clearly want to avoid. If our component fails to register the extracted point clouds according to a high accuracy standard, non-real-time processing shall be applied instead.

- Flexibility

  The flexibility requirement of using the system in many different ways also extends to our component. Other research suggests that the way the operator moved the camera vastly affected the registration results [EHMR+22]. Since we have more than one component to handle localization and registration, we think this constraint can be easily mitigated.

- Robustness

  The component shall be able to work in all different environments of buildings. This should include empty buildings which were just built with nothing but hallways, as well as offices where people are actively working in, possibly containing unnecessary data like chairs, tables, and other objects that are not part of the BIM model.

- Little to no manual configuration

  As stated in the flexibility requirement, the system should be able to adjust to many different ways of using it. But it should also work reasonably well 'out of the box'. This means that the system should require little to no manual configuration before it can be used. While there will always be configuration parameters that can be tweaked to possibly get even more accurate results for any given environment, the system should be able to generalize well without the need for adjustment.

- No need for preparation of the environment

  The environment should be able to be scanned and compared with the BIM model "as is". This mainly concerns localization and tracking methods that require the placement of markers around the environment, which is something we cannot allow for our project. But this requirement also expands to making our system robust enough such that it can also handle the presence of people in the environment. If the important parts of the environment are not occluded, "noise" (i.e. objects that are not part of the BIM model) should not affect the evaluation either.

- No access to the internet required

  While our component needs to communicate with the rest of the BIMCheck system, we cannot depend on Wireless Fidelity (WiFi) to be available at the various scan sites (especially not in freshly constructed buildings). Therefore, our component cannot rely on cloud solutions, online converters or other internet services to work.

- Support processing of both live and recorded data

  The *RGB-D data component* shall be able to run both while moving through an environment as well as on recorded data after acquisition. While not all differences can be immediately presented during live acquisition, the component shall be able to provide good feedback for areas and level of accuracy captured. It shall also save the raw data acquired and be able to re-run on only this data after the fact. Such an offline mode may even produce better results, due to not having to adhere to any real-time requirements.

- Estimation of cameras's position and orientation relative to LiDAR at all times

  Another requirement of our *RGB-D data component* is to be able to estimate its own change in position and orientation from its own data at all times as well. This data shall accurately reflect the current position of the camera in relation to the other components of the BIMCheck system and be able to be accessed by other components at any time. While not part of this work, this could become important for refining the position inside the BIM model in the future.

3.3 Azure Kinect Data Integration Method

The goal of the *RGB-D data component* is the integration of Azure Kinect data. This encompasses extraction and registration of point clouds captured by the depth camera of the device as well as reducing them as much as possible before sending them to the object retrieval component.

The integration of Azure Kinect data is achieved in a series of steps that we refer to as the *Azure Kinect data pipeline*. The following section will explain how this pipeline is designed to meet the requirements and goals of the *RGB-D data component* in a real-time fashion. The pipeline described works the same both in a live settings as well as on a saved recording. When being run on a saved recording, where real-time processing does not matter as much, various parameters of the pipeline can be adjusted to improve the quality of the output even further, in exchange for longer computation time. For more details and comparisons between various set-ups refer to Chapters 5 and 6. In this Section we will first give a short overview of the various steps, then go into the theoretic details of each step. How this theory was implemented will be discussed in Chapter 4.

The *Azure Kinect data pipeline* can be seen in Figure 3.3. Its two inputs are the images gained live or from a recording of the Azure Kinect, as well as LiDAR poses with their corresponding timestamps. These two, as well as the extracted point cloud are represented with doubled borders in the Figure. All other rectangles represent the steps the pipeline goes through from an image to the finalized point cloud. Note that most of the rectangles are dotted, representing optional steps of the pipeline. Likewise, solid and dotted arrows represent necessary and optional inputs for every step. The reasoning behind their optionality is to allow for maximum optimization of run-time depending on parameters used. Theoretically, one could run the pipeline with just the 2 necessary steps (Extraction and Transformation from image to LiDAR coordinates). Of course, doing so would not align anything. All other steps can be independently added from there. For example, if one were to only add KinFu to the 2 necessary steps, then the final registration would be done just using KinFu.

Whether run live during a recording or afterwards on a saved recording, the pipeline always handles every frame step by step. But, unless otherwise specified by the most extreme of parameter settings, not all frames proceed through the whole pipeline. This is due to the fact that we simply do not need information from every frame. Two consecutive frames will most likely have the same information in them, with only a small shift in-between them. We can safely assume this due to motion blur hindering fast movement of the Azure Kinect camera anyway. Point clouds extracted from an extremely fast moving operator will fail to provide any usable information at all. For this reason, every frame is only processed as far as it is needed and most are discarded before any point cloud is extracted. However, if a point cloud is in fact extracted, it will make its way through the rest of the *Azure Kinect data pipeline*.

We will now give a brief overview of all of the steps and their order as seen in Figure 3.3, then explain each in detail in the next section. Aside from the fundamentals (reading a

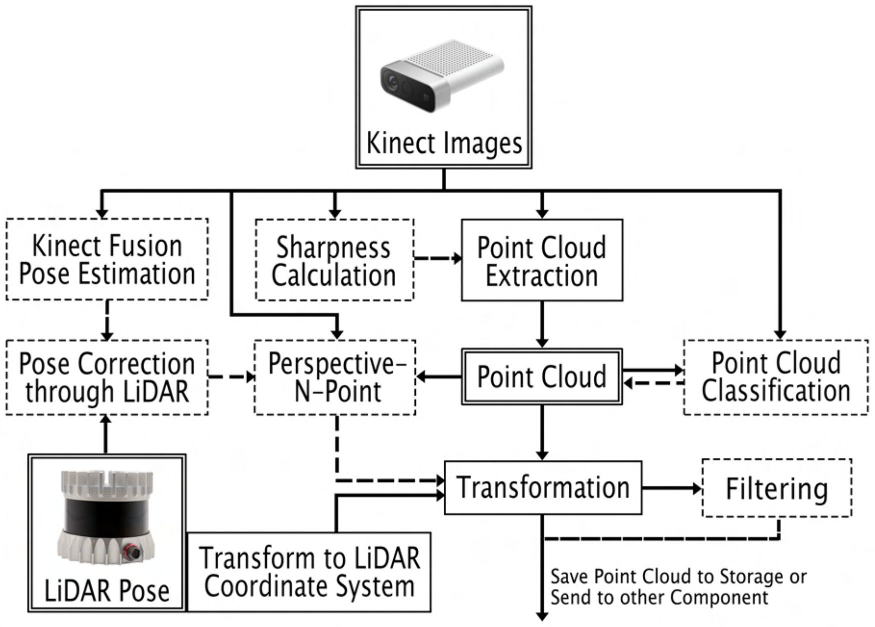

Kinect Images
Kinect Fusion Pose Estimation
Sharpness Calculation
Point Cloud Extraction
Pose Correction through LiDAR
Perspective-N-Point
Point Cloud
Point Cloud Classification
Transformation
Filtering
LiDAR Pose
Transform to LiDAR Coordinate System
Save Point Cloud to Storage or Send to other Component

Figure 3.3: Azure Kinect Data Pipeline: *Doubled borders* represent data objects, *solid borders* represent necessary steps, *dotted borders* represent optional steps. Likewise, *solid arrows* represent necessary inputs, *dotted arrows* represent optional inputs.

frame and extracting a point cloud from them), all steps are theoretically optional and can be encompassed or skipped by the user-set parameters as one sees fit.

1. Frame registration estimation using KinFu

   The pipeline starts with receiving a new frame, either from our currently active device or from a recording. Due to its incredible lightweightness and lack of need for any pre-processing, our component may always keeps track of an active KinFu model, into which every frame is added. This serves as an initial pose estimation for each captured frame, which is especially accurate in a scene with lots of distinct objects.

2. Frame registration adjustment using LiDAR pose

   Because KinFu quickly exceeds its limit [WMK+12], LiDAR poses obtained from the network are compared with the one's obtained from KinFu. In order to compare them, we add the physical offset between the two sensors to the pose estimation we received from KinFu. This offset is set by the parameters. Should the KinFu and LiDAR poses differ too much, KinFu is reset and the LiDAR pose overrides KinFu's pose for current frame.

3. Frame pre-processing via sharpness filter

As stated at the beginning of the section, we do not think it makes much sense to extract point clouds out of every frame we record. One reason is the vast overlap in area covered between two consecutive frames. Another reason is that more computationally expensive parts of the pipeline will not be able to keep up with both the amount of point clouds and the real-time requirements of the *Azure Kinect data pipeline*. Since we cannot handle every frame we record, we only want to extract point clouds from the best frames. For this reason, we calculate a sharpness value from each of the color images and add it to each of the frames in the pipeline.

4. Point cloud extraction

   After we calculated the corresponding sharpness values for each of the recordings frames, we choose the highest scoring inside a set interval for point cloud extraction. For this the image is converted into a colored 3D point cloud using the camera's intrinsic parameters. Most of the frames are discarded by this sharpness filter, with only the sharpest point clouds getting to the next steps of the pipeline. Besides the point cloud, the color image is also stored for future pipeline steps. Note that we cannot transform the extracted point clouds yet, as the image would and point cloud need to be in same coordinate system for future steps.

5. Point cloud classification using YOLO-World [CSG+24]

   The stored color image is sent into a neural network for real-time object detection. Although more sophisticated object detection and classification is handled by a different part of the BIMCheck system (the *object retrieval component*), our pipeline also allows for a rough detection of adjustable prompt descriptions. This can be seen as a binary separator between points-of-interest vs. points-to-be-filtered. It could also serve as an initial guess for further classification. Our chosen network, You Only Look Once (YOLO)-World [CSG+24], detects rectangular areas based on descriptors inside the color frames, whose bounding boxes are then projected onto the point clouds. Obviously, non-rectangular objects (e.g. fire extinguishers) will have a lot of wall around them classified as well, so for a more than just binary detection, the more sophisticated *object retrieval component* is still needed. Detected labels are then added to each point inside the point cloud.

6. Point cloud registration refinement using Perspective-n-Point

   While the registration we obtained from KinFu and adjusted with our LiDAR is already very accurate, we can refine it even further using Perspective-n-Point (PnP). PnP takes as input already extracted points in 3D space and a new 2D image in order to calculate the pose where the new image was taken. In order for this to work, points in the image need to be matched to their counterpart in 3D. Therefore, we extract ORB features [RRKB11] from both the previous as well as the current frame. We match them and filter those matches to the best ones. The resulting pixel coordinates in the previous frame are then projected onto the previous point cloud. Through this, we now have a pair of 3D coordinates of the previous (already aligned) point cloud and the new 2D image. We can further input our current pose estimation to PnP to receive an even better one. This then serves as our final pose for the extracted point cloud.

7. Point cloud transformation

   After we are done adjusting our extracted point cloud through its corresponding image and have arrived at a final pose estimation, we can finally transform the point cloud to its estimated position inside 3D space. This also changes the coordinate system from a typical pinhole-camera model to the one our LiDAR and the rest of the system uses. After which, the memory of the image can also be freed. It is of little use anyway if it is not aligned with its point cloud.

8. Point cloud filtering and reduction

   After we transformed our point clouds, we want to reduce their size, in order to send them to the other components of BIMCheck more efficiently. We implemented several strategies for this, which can be enabled and combined as one sees fit:

   - Filtering unlabeled points

     The simplest method to reduce each point cloud is to use the results of our *object detection step*. For this, we simply remove all points that did not get classified by any of our descriptors. Aside from the case of false negatives (i.e. failing to detect a detail where it should have), all remaining points and their predicted label can be easily sent through the network for visualization.

   - Filtering using plane segmentation

     The idea for filtering through plane segmentation is that we expect our objects to stick out from the walls they're on. So if we detect those walls and simply remove all points on a wall, we should end up with just the possible details we're interested in. Besides, if we store the plane equations we can reconstruct an abstraction of the full point cloud as needed. Though texture and precise depth deviations on the walls would be missing.

     But in order for our plane segmentation to filter the point clouds to their details, each point needs a normal. Neither LiDAR's nor depth cameras usually capture normals, but estimating normals for a given point cloud is a common task in the field. Estimation of normals can be extremely precise if the position of the sensor which captured the points is known [Rus09], which it is.

     After we calculated the normals we can reduce the point clouds to its regions of interest by using plane segmentation. Our plane segmentation method tries to bundle as many points as possible that could be abstracted by a plane equation. We then only keep regions in the original point cloud around points not on these planes.

   - Filtering using overlap filter

Even when reducing our point clouds through our previous filters, there will probably still at least a small overlap between two consecutive extracted point clouds. In order to remove these we implement an *overlap filter*: We compare each point cloud's camera's final pose with all previous poses to find the ones whose view frustum overlaps our current poses' view frustum. Once we have a set of point clouds and their poses, if a point is inside two or more view frustums, we only keep it if it is closest to its own camera's pose. Otherwise, the point from the closer camera position is kept. This not only reduces their size, but also makes the complete point cloud cleaner, as overlapping regions do not always match up perfectly, thus giving a blurred or fuzzy visualization.

The overlap filter is the only of our filters which also reduces older point clouds. Since we cannot keep an unlimited amount of point clouds in memory, it can only act on a finite amount of previous ones. Newer point clouds which have overlapping regions with old poses, but no corresponding point clouds to access can be dealt with in two ways: Either keep all points as they are, definitely leading to overlaps or filter all points inside old view frustum, potentially removing points not inside the old view frustum's actual capture.

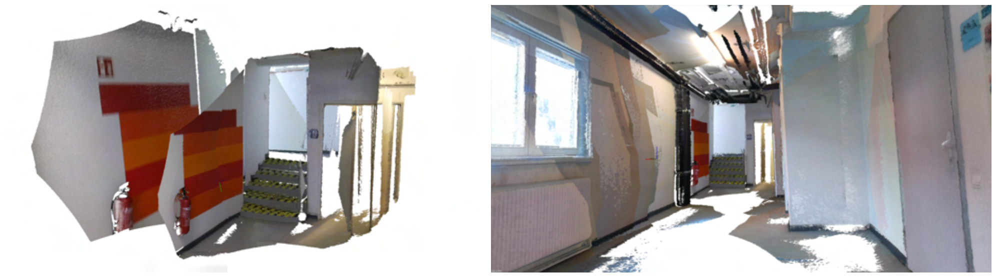
9. Point cloud saving, transferal and releasing of memory

   In order to free up memory, we send or write out older point clouds in the pipeline at a specific pace. This way, the area able to be captured is not limited by RAM but by storage memory. We cannot do this right away though, as the overlap filter might still filter older point clouds by newer ones. If the overlap filter is activated, a parameter must decide between less active point clouds in memory and sooner point cloud transferal vs. more reduced point clouds and therefore faster but later point cloud transferal.

3.3.1 Frame Registration Estimation using KinFu

As discussed in Section 2.2, KinFu [NIH+11] is an incredibly lightweight algorithm to register point clouds. What is more, it works directly on input depth images without any need for pre-processing of any kind. Although we have no use for its actual output, i.e. an extracted surface mesh, it can provide us with a sensor pose estimation for Azure Kinect's 3D position in space. Because it is barely using any computation power even when being run on the maximum 30 frames that can be recorded each second, we can always keep a KinFu model updated and extract transformation estimations for each capture we want to further process. Should KinFu fail (e.g. due to exceeding its bounds [WMK+12]), we simply reset it at origin. The frame where it failed will be stored by the *Azure Kinect data pipeline* along with the correseponding LiDAR's pose. Then, each new estimated KinFu pose can be added to the position of the LiDAR at failing. A normal extraction rate of point clouds usually cannot properly take advantage of the transforms estimated here, due to it failing after a few seconds (depending on the handling of the device).

3.3.2 Frame Registration Adjustment using LiDAR Pose



(a) KinFu without LiDAR (b) KinFu adjusted by LiDAR

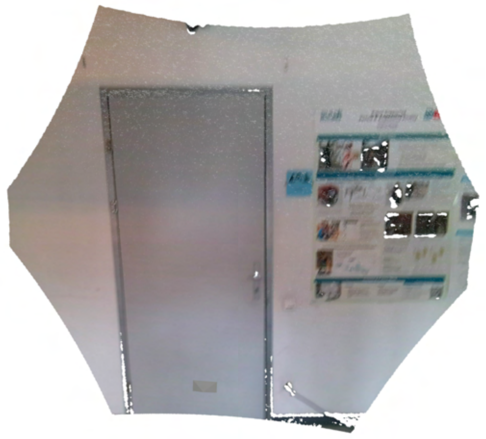
Figure 3.4: Example of KinFu failing. The 90 degree turn from the stairs to the fire extinguisher in (a) is already too much for KinFu to handle, as a new frame featuring the same extinguisher is severely off-set from the previous ones. (b) features the same scene and registration using LiDAR as a back-up. By comparing estimated poses to those of the LiDAR's pose, the off-set frame is detected and registered using LiDAR's estimate.

KinFu quickly exceeds its area of operation. Therefore, we request LiDAR poses from the LiDAR tracking component and compare those against the estimated KinFu poses. The *LiDAR component* uses a sophisticated SLAM algorithm [Kud22] which can provide much more accurate results for areas bigger than KinFu's operational range of a few cubic metres. But in a small and cluttered area, KinFu pose estimations are often more accurate than the LiDAR poses received from the network. Therefore, we want to use LiDAR's position only as a fallback. When deciding whether to use the KinFu's or LiDAR's pose, we compare the two given transforms. If the distance or angle between them exceed a given rotational or distance threshold, we reset KinFu and keep the LiDAR pose for our current frame. We also store the LiDAR's position at the point of the last reset of KinFu. Doing that, if we do not reach the threshold we take the last LiDAR transform when KinFu failed and add the current KinFu pose to it to obtain our current KinFu adjusted threshold registration.

3.3.3 Frame Pre-Processing via Sharpness Filter

The operations on image frames are already rather computationally expensive. But as we move from 2D images to 3D point clouds in our pipeline, the computational cost increases drastically. If we want to keep a real-time performance and stay memory efficient, we cannot extract a point cloud out of every frame. Besides, extracting 3D information out of every frame would most likely lead to many of the same points being added multiple times.

For all these reasons, we intend to process only the best frames captured. In order to decide which frames should be further processed, we introduce a *sharpness filter*. Due to the physical constraints of every camera, motion blur is a common occurrence when recording. In a typical video recording it is not unusual that most frames are at least


(a) Point cloud extracted from blurry frame

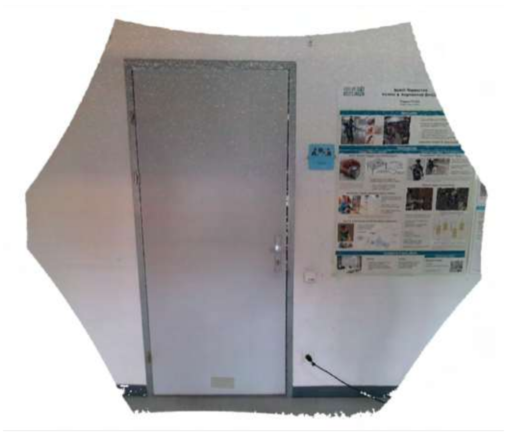

(b) Point cloud extracted from sharp frame

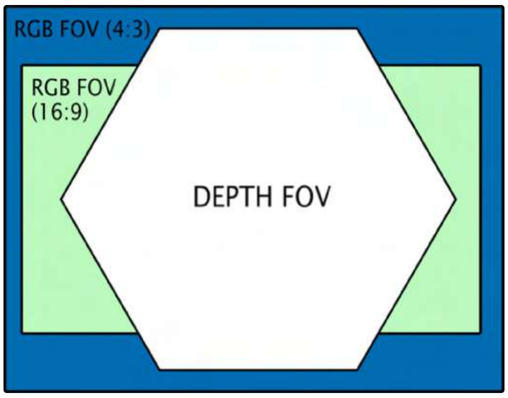
Figure 3.5: Comparison of point clouds extracted from blurry and sharp frames. Aside from the poster on the right being unreadable in (a), notice how the point cloud of a blurry image features more holes due to erroneous depth data, as well as making objects like the light switch appear bigger than they actually are.

somewhat blurred. While the human eye might not pick up on blur during playback of a video, in a still frame the difference between blurry and sharp can mean the difference between correct and noisy or outright false information. Our *Azure Kinect data pipeline* can calculate a sharpness value using either Fast Fourier Transform (FFT), a Laplacian approach or a Sobel filter. The choice between them is a choice between accuracy and CPU resources or time needed. However, even the slowest method, (FFT), can process 30 frames per second on regular hardware, if no other CPU-heavy task is running. One problem that all three methods suffer from is that they can only differentiate between sharpness relatively. This is due to them mostly looking for edges and details inside a frame. A completely sharp but empty wall might have a lower sharpness score than a fully blurred image of a painting (although a sharp image of the same painting will always have a higher value). So while it is difficult to completely distinguish blurry from sharp images using this approach, one can at least detect the sharpest frame for a given recording interval. The calculated sharpness value is added to the frame in the *Azure Kinect data pipeline*.

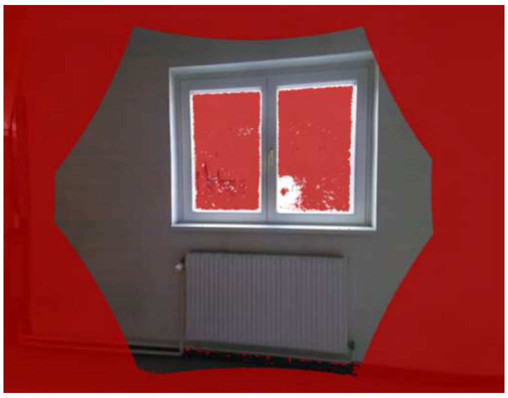
3.3.4 Point Cloud Extraction

The point cloud extraction part of the *Azure Kinect data pipeline* works on an interval set by the user-defined parameters. For simplicity, let us assume an interval of exactly 1 second. In order to extract the first point cloud, the extraction component must first wait until all 30 frames of the first second have a corresponding sharpness value attached to them, if the *sharpness filter* is enabled. Only then are we extracting a 3D point cloud from the sharpest 2D frame using the camera's intrinsic parameters.

The Azure Kinect device does not feature one camera able to capture color and depth values representing a colored point in 3D space, but rather one camera for each. These cameras are positioned a few centimeters apart from each other, which means that the depth values are not perfectly aligned with the color values. In order to align the depth image and the color image, we could either transform the depth image to the color image or the color image to the depth image. As per official documentation, transforming the depth image to the color camera is recommended [Mic22], because the color image can be of much higher resolution. Doing it the other way around would downsize the color image to the resolution of the depth camera. Furthermore, using Microsoft's functions, each color pixel that cannot be mapped to a depth pixel leaves a blank spot in the transformed color image, whereas depth values are interpolated when they are transformed.



(a) Azure Kinect depth and RGB Field of View (FoV)



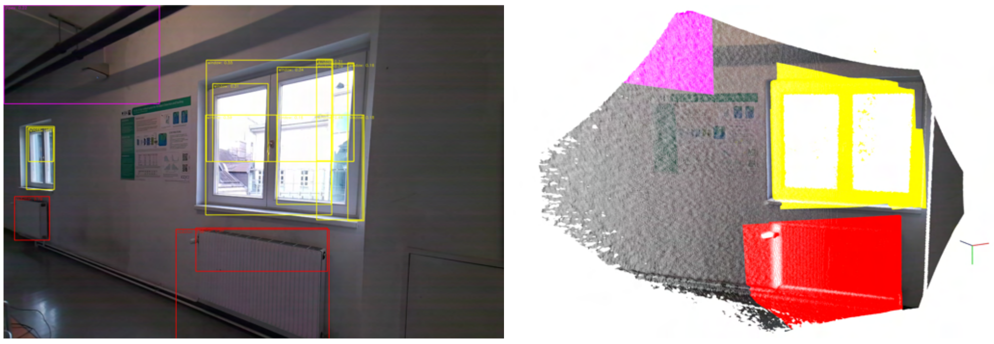
(b) 4:3 RGB image with missing depth data

Figure 3.6: Azure Kinect FoV Comparison. (a) shows a comparison between the regions covered by both the RGB and Depth camera. (b) is an extracted RGB frame with red pixels signaling missing depth information. Not only are the edges cut-off (like seen in (a), but areas outside the window or on the edge of the radiator also show missing depth values. Without 3D information, these points of no use to us.

Another challenge of combing color and depth information is posed by Azure Kinect's untypical view frustum. As seen in Figure 3.6, Azure Kinect's depth camera does not record rectangles like a typical camera, but rather a hexagon. Because of this, a certain portion of pixels from the color camera's view frustum cannot be extracted to 3D points properly. To combat this, when writing to our point cloud object from the Azure Kinect's image buffers, we check for invalid pixels by both their depth and color values. Pixels with zero depth values or pure black color values (since that does not exist in the physical world) are marked as invalid. However, we cannot discard them immediately as every following operation using the point cloud and its corresponding image needs their indices to represent the same pixel in 2D and point in 3D space. We only discard them once the pipeline has finished all operations in need of the image, which coincides for us when transforming the point cloud.

3.3.5 Point Cloud Classification using YOLO-World [CSG+24]

While not stated in our goals, we also experimented with object detection and classification in our extracted point clouds. The more sophisticated classification still happens in the *object retrieval component*, which takes as input our processed point clouds. But if nothing else, our own classification can provide a real-time estimation of a class for each point inside the point cloud. It can also be used as a quick filter to throw away any points not classified (or classified as something we want to filter).



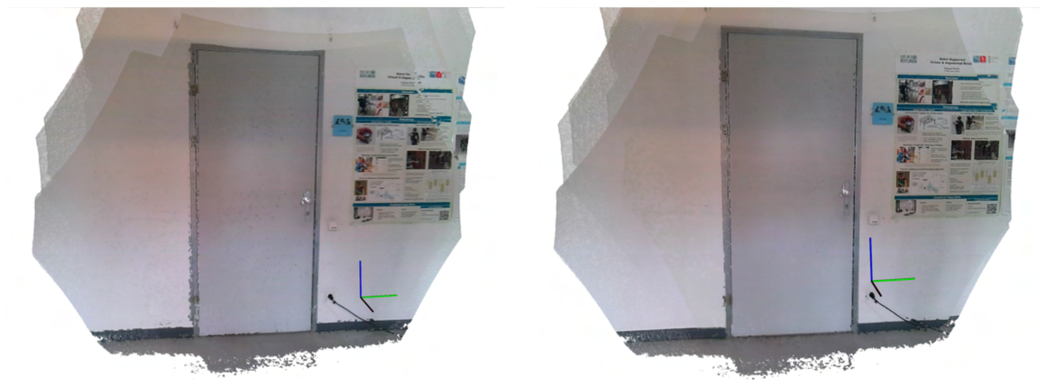
(a) Detected classes inside frame (b) Detected rectangles in point cloud

Figure 3.7: YOLO-World frame and resulting point cloud. Due to just rectangular detection, 3D points around each object are (wrongly) classified as well

We chose to opt for a neural network for image classification due to them having vastly outperformed traditional methods for the last 5-10 years [MCC+20]. We chose YOLO-World [CSG+24] not only for its real-time applicability but also because it is designed for an open vocabulary. This means that it can detect objects from prompts, even if they were not part of its original training data. So when the pipeline of our *RGB-D data component* is being run on a recording, one can change the prompts to detect different classes in each run. Another benefit this brings is the option to chose a different set of objects according to each environment. That way, if we know what we can expect in an environment, we can remove conflicting classes or let a user detect object we did not even know exist. Although more error-prone, due to YOLO-World working on any prompts one could even go as far as input more complex classes like "multiple light switches besides door", without changing anything about the code or model. Our parameters give the option of detecting either a multiple of 32 class descriptors (or prompts) for each point, or even more while saving only the highest scoring class.

Since YOLO-World [CSG+24] works on images with a size of 640x640, we can chose to down-sample our images to that size, or split our images into multiple parts and let the network run one after another. As Chapter 5 will show, our set-up can easily handle splitting the image even at Azure Kinect's highest resolution, given a reasonable extraction rate. The network then returns rectangular bounding boxes of detected classes on our color image, which we project onto our point clouds. One downside of using YOLO-World is that it cannot provide us with the exact outline of our objects. So objects like fire extinguishers and especially pipes will have a lot of falsely classified wall points around them as well. But for a binary check if e.g. a fire extinguisher was installed or not this estimation already provides highly accurate results.

3.3.6 Point Cloud Registration Refinement using PnP

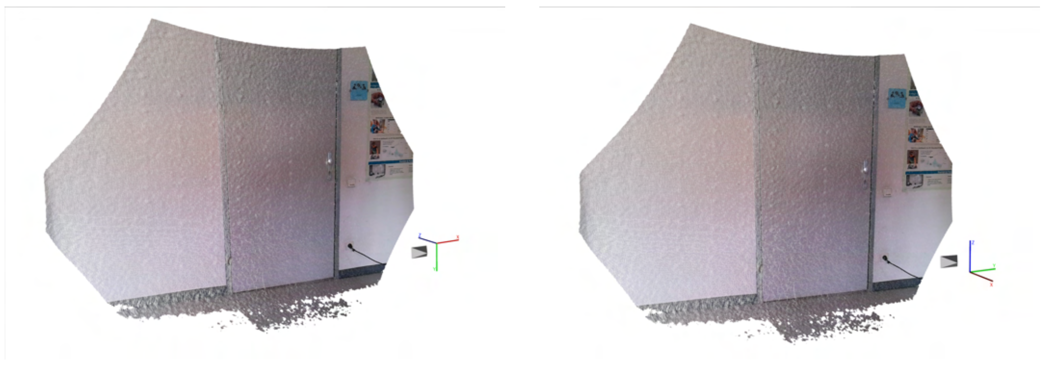


(a) Registration without PnP (b) Registration with PnP

Figure 3.8: Refinement of pose estimation using PnP. Although the KinFu and LiDAR registration (a) is already very accurate, PnP improves the results around the knob and left edge of the door

Even though the pose estimates from both KinFu and LiDAR are already accurate enough to create a 3D model of a whole building, they are still slightly misaligned when one looks at the details. For an example of this, look at the way the left edge of the door looks in Figure 3.8a. To combat this small misalignment, we can use the initial pose estimation as a start and refine it by PnP. PnP estimates our Azure Kinect's pose by finding matching points between already known 3D point locations, a newly captured 2D image and the intrinsic camera parameters. In order to find matching points between the already extracted point clouds and the newly taken image, we look at the last extracted point cloud, or more precisely, its 2D image. There, as well as in our new image, we detect and extract ORB features [RRKB11] and compare them in order to find the best matching pairs. After that, we find the corresponding 3D points of the last point cloud by its image indices. Now that we have 3D points of the last point cloud with matching 2D points of the current point cloud as well as a pose estimation, PnP should easily find the correct 3D pose, such that all small details align.

The only downside of this method (or any PnP approach) is that it needs detectable features in the scene [MAT17] [LMNF09]. But in our case we have a highly accurate pose coming from our LiDAR anyway. The only times its errors are visible are when detailed objects appear in the scene, which is where PnP works best. This serves as the final pose for our point cloud.



(a) Azure Kinect's image space coordinates (b) BIMCheck's right hand side coordinates

Figure 3.9: Image space and BIMCheck coordinates. Image space has positive x towards the right, positive y towards the bottom and positive z forwards. BIMCheck uses the right hand side coordinate system, i.e. positive y is towards the right, positive z is up and negative x forwards.

3.3.7 Point Cloud Transformation

In the transformation step, we do not only transform our point cloud to its estimated position, but also from image space coordinates into the right hand side coordinate system the LiDAR and the rest of BIMCheck uses. Both coordinate systems and their axis can be seen in Figure 3.9. While we are rotating and flipping our point cloud into this coordinate system, we also apply the real-world offset between our Azure Kinect and our LiDAR's sensors, so that their respective point clouds could be overlaid exactly. The offset is set by user-defined parameters. Our hardware offset can be seen in Figure 3.2a.

There are several reasons why we are not transforming our point clouds earlier in the pipeline. One reason is that before our pose estimation is final, any transformation would have to be overwritten again anyway, taking away precious CPU and/or GPU resources. The more important reason is that the pose refinement using PnP needs the last extracted point cloud before the current one to be in image coordinates. Therefore, we only transform our point cloud after the current point cloud's pose has been adjusted via PnP. Before transforming the point cloud into the coordinate system of the LiDAR, we may also remove all invalid points, as the point cloud will no longer be compared to its corresponding color image anymore. Therefore, its indices no longer need to match up with indices of pixels in the color image. This also releases the color image from memory.

3.3.8 Point Cloud Filtering and Reduction

After transforming our point clouds, we want to reduce their size as much as possible. Since we need to transfer our clouds to the object detection component over network, doing so can vastly improve the time needed for this. Furthermore, the geometric architecture is captured by the LiDAR as well. The only thing the LiDAR struggles with is capturing details or small objects of interest. Ideally, we would remove all featureless

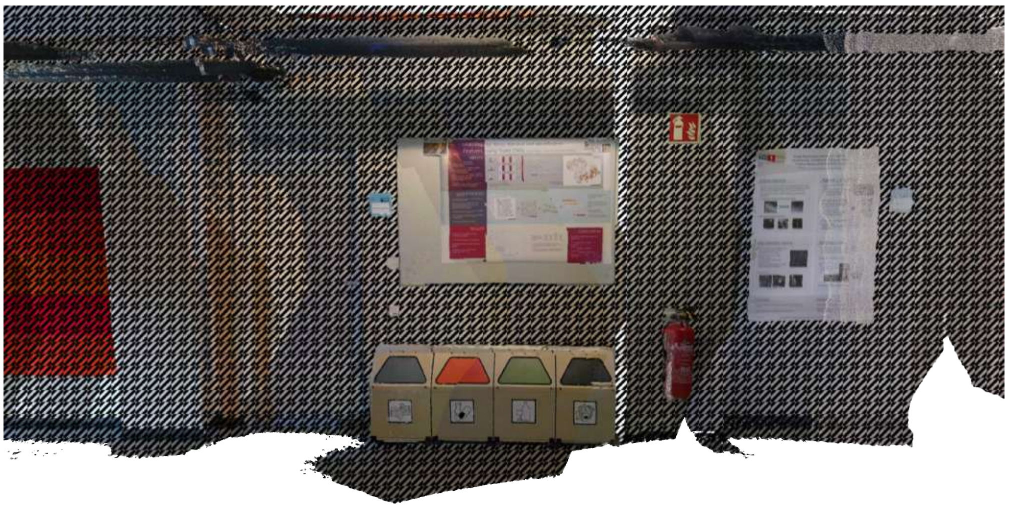


Figure 3.10: Ideal filtering of point clouds. Only objects we might be interested in are kept

.

areas of the point clouds. But we must be careful not to lose any of the necessary data in the process. A theoretical perfect filter is shown in Figure 3.10. We have experimented with three distinct methods to reduce the memory footprint of our points clouds, which we will discuss here and evaluate in the following Chapter's Section 5.3.

Filtering Unlabeled Points

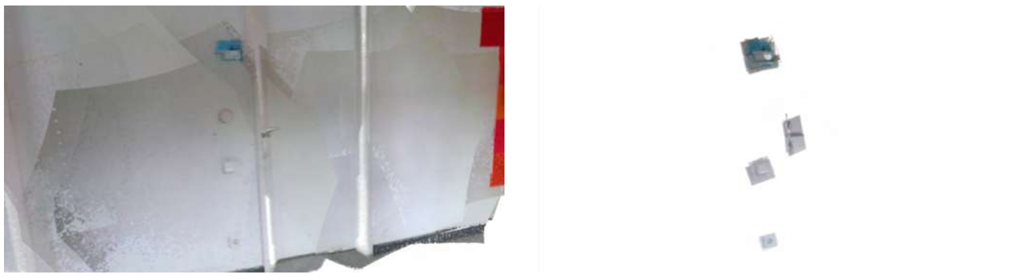

(a) Combined point cloud unfiltered (b) Points without classification filtered

Figure 3.11: Filtering using object detection

If our classification step is active in the pipeline, we can reduce the points in our cloud to feature only those which were classified by it. At this point, we do not care about the correct labeling of each point, just that we do not delete any point belonging to an object of interest. Therefore, our detection can be classified into a binary "object of interest was detected" vs. "object of interest was missed". Given this binary distinction, the object detection can arrive at 4 different results for each point:

1. True positive: The point is an object of interest and was detected as such.
2. True negative: The point is not an object of interest and was therefore not classified.
3. False positive: The point is not an object of interest but was detected as such.
4. False negative: The point is an object of interest but was not detected as such.

Cases 1 and 2 signal the point cloud classification step to be working correctly and should be the norm. Case 3, while failing to reduce the point cloud to its minimum is not a deal-breaker, as we do not lose any information. The problem comes with Case 4, where we are losing an object of interest. So the success or failing of this filter comes down to the rate of false negatives. Once false negatives have been eliminated (e.g. by specifying an extreme amount of descriptors), one can look to the false positive rate for a reduction score.

Figure 3.11 shows an example of multiple point clouds being filtered using this filter. The blue nameplate and door handle would be false positives, as it they are not part of the descriptors. To combat those, one can also provide additional descriptors to detect objects for filtering out instead of keeping.

Filtering using Plane Segmentation

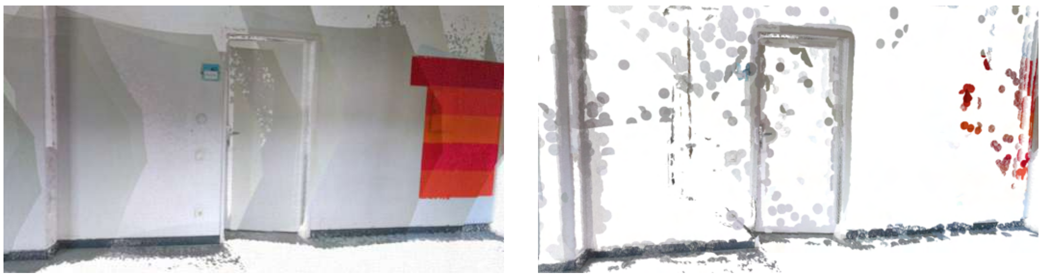

(a) Combined point cloud unfiltered (b) Points on planes filtered

Figure 3.12: Filtering using plane segmentation

Plane segmentation is another way to reduce a point cloud to its objects of interest. In other words, we want to abstract or completely discard all points on flat walls, doors and other surfaces by detecting a plane that covers their points. After removing those, we should have left only those points that are elevated above the detected planes. Since we do not expect our objects to be completely flat (or just textures on a wall) those points should still contain the features we are looking for to send to the *object retrieval component*.

For our plane segmentation algorithm to work our points first need normals. Even though we do not have them, estimating normals is a normal procedure when working with point clouds and the PCL includes an implementation of Rusu's algortihm for it [Rus09]. It is especially accurate when the sensor's position is known, which it is. But estimating normals for point clouds still as big as ours would take too long for a real-time application. Therefore, we first sub-sample our point clouds to a sparser voxel grid, estimate normals for those and run our plane segmentation algorithm.

Our plane segmentation algorithm is based on Araujo's and Oliveira's work and code [AO19]. An octree [Mea82] is constructed for the input point cloud. The octree is partitioned until a child leaf contains only points whose normals and positions are all roughly aligned with the average plane between them (from now on called plane criteria). These are extracted as the initial plane patches. A k-d tree [Ben75] in constructed over the input point cloud to find n neighbors for each point inside it. The plane patches extracted from the octree are then grown by checking if its points' neighbors fit their own plane criteria. Two neighboring plane patches are merged if two of their points fulfill both of the patches plane criteria. The growing and merging is looped until only separate patches remain. The function returns not only the final plane equations but also the indices of all points on the detected planes. We can now use the inverse of those indices to get all points in the sub-sampled cloud which are not on any detected plane. After constructing a k-d tree over our complete point cloud, we can use the indices to query the tree to extract all points around a small radius of a certain point. This makes sure to keep even the smallest objects like light switches in their full detail. Results can be seen in Figure 3.12.

Filtering using Overlap Filter

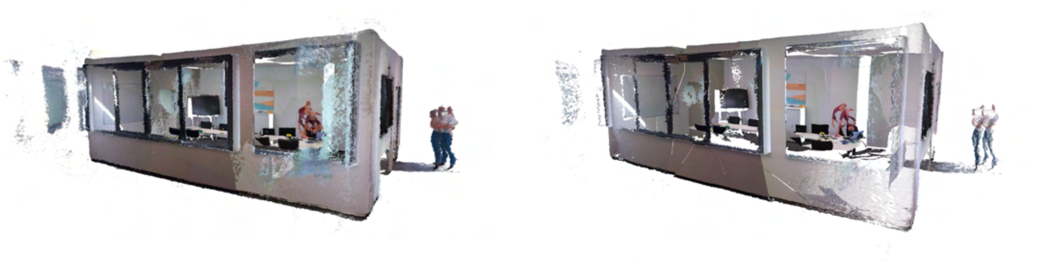
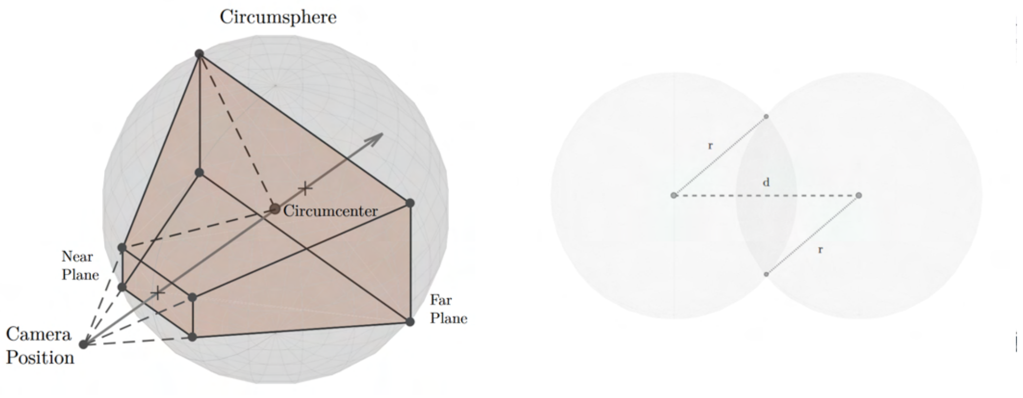

(a) Combined point cloud unfiltered (b) Overlapping point clouds filtered to only have one point per 3D coordinate

Figure 3.13: Filtering using overlap filter

As explained in Section 3.3.3, we expect neighboring image frames to feature roughly the same information. For that reason, we only extract point clouds at a certain rate. However, even after doing that and possibly filtering them using the above steps, point clouds might still have overlapping regions inside them. We therefore implemented an *overlap filter*, which only keeps points inside a certain point cloud if they are nearest to their corresponding camera pose. An example of the results can be seen in Figure 3.13. The overlap filter is the only of the three filters which also reduce already filtered point clouds, as older points will be compared with newer poses. But since we cannot store an infinite amount of point clouds inside RAM, the *overlap filter* can only be run on a finite amount of previous point clouds. Problems (or rather a compromise) must occur if the same area is captured after that. But we will first discuss how our overlap filter works in detail.

Detecting an overlap by comparing 3D point coordinates between a dozen points clouds would be far too costly. Therefore, we have to abstract the area our point clouds occupy in some way. A bounding box aligned with the coordinate system would be a simple abstraction to do this. But the amount of overlap between two point cloud's bounding boxes is influenced more by the point cloud's alignment with the coordinate system than their actual overlap. A more novel approach is to measure the overlap of the two point cloud's camera poses' view frustums. Due to the way view frustums are geometrically constructed (especially the Azure Kinect one's as seen in Figure 3.6), there is no easy way to calculate the exact overlap efficiently. But we can approximate view frustums as spheres whose center is equidistant to all eight corners of the view frustum, as seen in Figure 3.14a. Then we can calculate an approximate overlap of their frustums using those spheres' overlap [ZSK17]. Since the parameters of our view frustum do not change for our camera, we can calculate the position and radius of the sphere inside our view frustum only once and simply transform it for all poses.



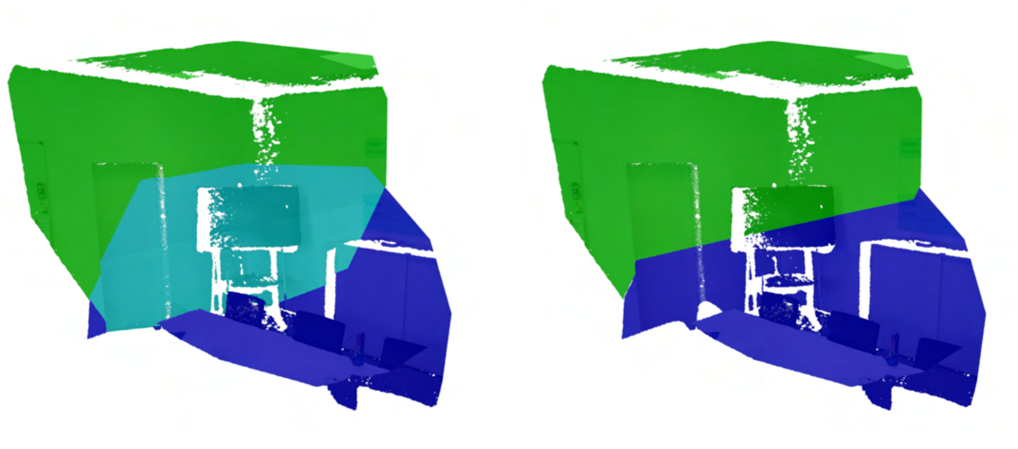
(a) Abstraction of view frustum by a sphere (b) Sphere attributes used for measurement of overlap

Figure 3.14: Measure overlap of view frustums with spheres [ZSK17]

So for each new point cloud we first calculate the distance of the transformed sphere's center to all other spheres' centers. If it is smaller than the sphere's radius, two point clouds might feature the same points. Since we want to reduce the new point clouds with **all** point clouds still in memory (and vice versa), we should do it efficiently. Therefore, after detecting overlaps, we sort them by the amount of overlap of each view frustum's sphere by using this formula [ZSK17]:.

$$\frac{(d-2r)^2(d+4r)}{16r^3}$$

where d is the distance between the two spheres and r their (shared) radius. Since we only care about an overlap score (and not an exact measure of overlap), we can skip dividing by $16r^3$. A visual example for measured overlap can be seen in Figure 3.14b.


(a) Two overlapping point clouds. Blue and green represent points only seen from their respective view frustum, turquoise points are seen in both.

(b) Two point clouds after filtering overlapping regions. Turquoise points of (a) are only kept if they are closer to their own camera pose.

Figure 3.15: Combination of two overlapping point clouds. The turquoise region in (a) is seen in both, therefore featuring twice as many points as the same region in (b). Turquoise points also part of green's view frustum's point cloud are only kept if they are closer to that camera position than they are to blue's camera position, and vice versa.

After we sorted the overlapping point clouds, we start by taking the most overlapping view frustum, and separating our own point cloud into *seen* and *not seen* by the other view frustum using a precise view frustum filter (see Algorithm 4.3). Points *not seen* are kept no matter what. Each other point is seen by both view frustums, so we compare the distance of the point to both cameras poses. The point is kept **only** if it is nearer to its own camera pose. Then, the same is applied to the old point cloud given the new view frustum. Therefore, if the two resulting point clouds would be combined, they should not feature any 3D coordinate present in both, as one pose should always be nearer (except for the highly unlikely case of them being equidistant, in which case both are kept). The same procedure is then applied with and to as many previous point clouds still in memory as set by the parameters. Filtering the point cloud with the more exact formula for all poses would quickly become too slow, but doing it in two steps allows all poses to be checked no matter how far behind they are.

However, two view frustums, even if overlapping, can feature completely different aspects of a scene. Think of a pillar which is captured by going around it like seen in Figure 3.16. Each of the view frustums feature the same pillar, but from a different side. Our overlap filter, naively implemented, would filter out all but one capture of the pillar, losing its structure in the process. Therefore, for a view frustum to be considered overlapping,

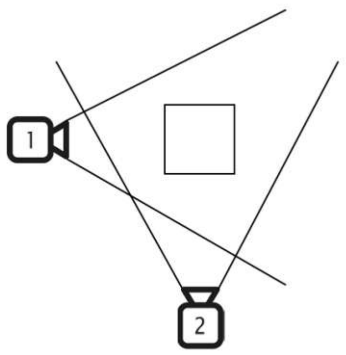


Figure 3.16: Example edge case of overlap filter shown with a pillar. If we let all poses filter all other poses, camera 2 would not be able to capture the south side of the pillar, which is invisible to camera 1. The same would happen when moving around any corner. Therefore, we only filter 2 view frustums if they do not exceed a rotational threshold.

they must not exceed a rotational threshold (combining all three rotation axes). That threshold can be set in the parameters but should be somewhere around the Azure Kinect's vertical FoV of 75° to 90°. This ensures a pillar or the corner of a building is captured in its fullest.

Another edge case comes when the same area is captured, but the previous point cloud is no longer in memory. In this case we have two options. First, we can either filter **only** the new point cloud by the old view frustum and the points' distances to it. This would lead to certain overlaps where the new pose is closer than the old pose. Or we could filter **all** of the newly captured points by the old overlap. This certainly avoids overlaps, though some details of the point clouds might be lost in the process. Ideally, we would accumulate a little bit of overlap using the first way, which is later filtered by another component before finalization of the point cloud.

3.3.9 Point Clouds Saving, Transferal and Releasing of Memory

After a point cloud has made it through our pipeline (given that it is not further reduced by future point clouds via the *overlap filter*) it is ready to be sent through the network or written to storage. This allows the point cloud to be freed from RAM and allows for quasi unlimited recording. The only thing that gets kept from each extracted point cloud is the pose, which can still be used by the *overlap filter* even without the point cloud.

CHAPTER 4

Implementation

This section describes how the system architecture and methods described in Chapter 3 were implemented in a real-time application component.

We chose to implement our component in C++, not only for efficiency reasons but also for library support reasons. The Azure Kinect Software Development Kit (SDK) only features an Application Programming Interface (API) for C and C++. Due to the fact that C++ includes various libraries covering many of our needs combined with the ease-of-use at a small performance cost compared to C, we opted to use C++. It was written and tested on x64 Windows 10 and 11. Due to the Azure Kinect SDK only being officially supported on Ubuntu 18.04, we did not opt for a Linux environment. To the best of our knowledge, all libraries should work on Ubuntu 18.04 as well, but we do not expect it to run out of the box without small adjustments. MacOS is not officially supported by the Kinect family of devices.

We will first discuss the general architecture of our component, including third-party libraries used. The sections following will then go over each of the *Azure Kinect Data Pipeline* steps seen in Figure 3.3. Section 4.3 will cover converting an image into a 3D point cloud. Section 4.4 will cover the registration process for each captured image and point cloud. Section 4.5 will go into detail about the 3 filters we implemented. Although not part of the pipeline, Section 4.6 will cover the implementation of the visualization, which can be activated both during and after recording.

4.1 Software dependencies

Our component makes use of many other software and libraries, without which none of the funcionality would have been possible. First and foremost, we use the Azure Kinect SDK. The SDK helps us start a device and record to .mkv, without needing to worry about the input and output stream ourselves. Furthermore, it pairs depth and color images around a timestamp and provides us with pointers to buffers for both of them. Lastly, transformations between the depth and color images as well as converting them into 3D points is also handled by the SDK for the most part. Only the decompression of JPEG color data needs an additional library, for which we use libjpeg-turbo [Com23].

Our final output may be 3D data, but we are still interested in many of the properties of Azure Kinect's images itself. For this reason, our component makes use of OpenCV [Bra00]. OpenCV is a library for displaying and operating on images. It includes implementations for filters, deep learning tools and even registration algorithms like KinFu [NIH+11] and PnP. The software also incorporates YOLO-World [CSG+24], which our software communicates with via the Open Neural Network Exchange (ONNX) Runtime [dev21].

For processing and visualization of the 3D data we opted to use PCL [RC11]. PCL is made specifically for point clouds, resulting in much faster processing and smoother visualizations than OpenCV's equivalent provides. Aside from being the fastest data structure to store and transform our 3D points, it also provides filters for sub-sampling, normal estimations as well as reading and writing of point cloud data. It makes use of the Eigen library [GJ+10], which is a math libary for linear algebra, which is also used for all of our non-PCL calculations.

4.2 Data handling of Azure Kinect Data Pipeline

This section will show how the different parts of our *Azure Kinect data pipeline* communicate and interact with each other.

Many parts of our program are computationally expensive. If the *Azure Kinect data pipeline* shown in Figure 3.3 would be run from beginning to end as each frame got in, we would miss out on a lot of data. This is because a live recording Azure Kinect only keeps two to five of the most recent captures in its queue available at any time. Until we would be done processing our current frame, the next frame might already be out of the queue. Therefore, in order to secure the recording of every frame, we built our component around multi-threading, such that each step inside our pipeline can work on its own. Furthermore, the most performance heavy steps were built from the ground up to be able to run multiple times on separate threads as well. So for example, if step A takes two seconds and step B, which follows step A, only takes 1, one could run twice as many threads for step A as for B. This way, even though each point cloud inside A still takes two seconds, the combined throughput of both threads would still provide enough point clouds for B to not wait idly. We implemented most of the functionality using just structs and functions, in order to increase performance by avoiding overheads that come with creating classes.

C++'s `std` library provides `<thread>` to easily handle multi-threading. In order to avoid race conditions, it also provides various mutex implementations. Using `<shared_mutex>`, multiple threads can read from the same capture or point cloud data array but only one is ever allowed to alter the data. We use C++'s `<vector>` instead of standard C

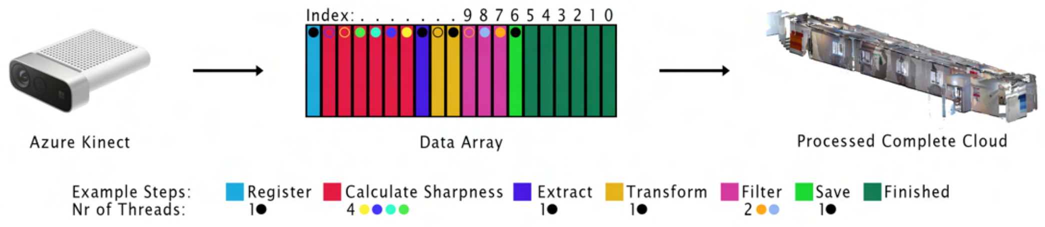


Figure 4.1: Data processing using multiple threads. Multiple captures are handled simultaneously, with one thread always processing an element from one state to the next. Multi-threaded steps always act on every *n-th* element, where *n* is the number of threads the step utilizes, seen here be the colored dots in the array.

arrays, but will continue to refer to them as arrays in order to avoid confusion with the mathematical term *vector*. So every step inside the pipeline always creates their own copy of the data, takes however long it needs to alter the data on their own copy and only overwrites the changed data afterwards. It is important to note that *copying the data* of a point cloud is handled by simply copying the `pointer` to the point cloud's memory location, as copying two to eight million points for each step would take an unreasonable amount of time. In order to write the data, a step needs exclusive access to the array by setting an `unique_lock` on the `shared_mutex`. Even though we hinder working on the same data element in other ways this is still mandatory, as adding elements to an array can change the position of the data inside the memory. If this coincides with the altering of data, changes could be written to the old place, lead to memory leaks or a read access violation altogether, crashing the program. A pipeline's step knows when it is their time to alter the data by checking the pipeline step attached to each capture or point cloud. Multiple threads acting on the same step know which data they should start operating on by their thread id (0 to *amount of threads - 1*) and can deduce the next data by incrementing that number by the amount of threads. If the next element is not at the correct pipeline step yet, the thread waits until it is. Once they are finished processing, the data's pipeline step is incremented, which is shown as colors in Figure 4.1. The colored dots show which thread is actively working on which element, with the unfilled circles representing the element said thread will be work on next.

But even using the techniques above, the recording of all frames cannot be guaranteed. Azure Kinect data should be requested at a minimum rate of 33ms (30 times per second). While highly unlikely, a big amount of threads altering the data before a newly recorded frame can get added could in fact happen. For this we split up our data into three distinct arrays: One for recording captures, one for processing captures and one for extracted point cloud data. With our three arrays, the most important thread responsible for recording has only one task: Read data from the Azure Kinect device, copy it to our own RAM and add it to our recorded captures array. A different thread moves the recorded

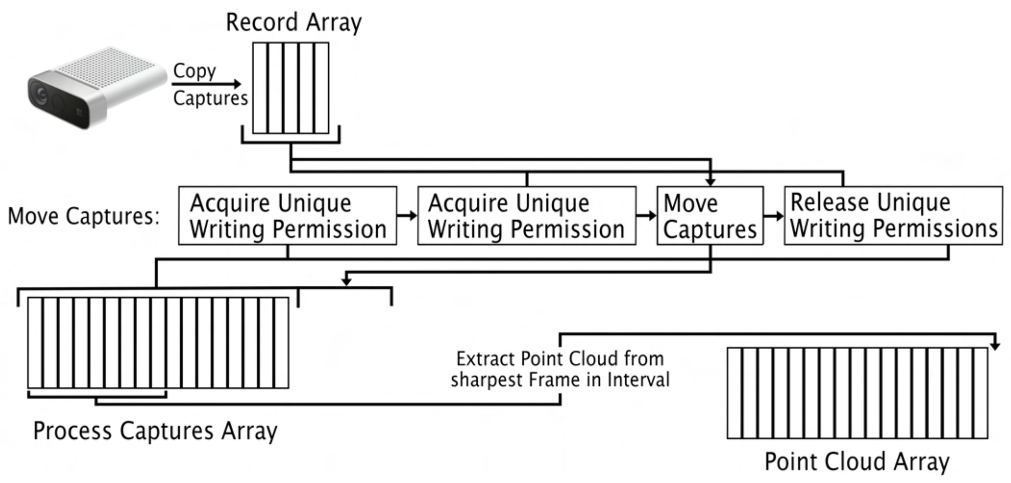

Record Array
Copy Captures
Move Captures:
Acquire Unique Writing Permission
Acquire Unique Writing Permission
Move Captures
Release Unique Writing Permissions
Extract Point Cloud from sharpest Frame in Interval
Process Captures Array
Point Cloud Array

Figure 4.2: Data processing implementation using 3 data arrays. While all activity of the *recording array* are shown, both the *processing array* and *point cloud array* can have many more threads active on them. Due to the *recording array* only being locked by one other thread, continuous copying from the Azure Kinect device can be guaranteed.

data to the processing array. Its only task is to wait until no other thread is currently reading or altering the array in which captures are processed. Only then does it wait for the recording thread to finish writing its current recorded frame and quickly transfers over all captures from the recording array to the processing array. Since this takes only a few milliseconds each time and no other thread ever blocks the adding of elements to the recording array, continuous recording can be guaranteed even at the highest framerate. Figure 4.2 shows how data is passed through the three arrays.

Once a capture is inside our array for processing captures, it undergoes various steps until only one capture per defined interval is extracted into a point cloud. After which, all captures and images are released from RAM. The same applies for the most part to the point cloud array. The only difference is that even after a point cloud has made its way through the pipeline, it's data might still be needed for the next point cloud's PnP step (see Section 3.3.6), or altered by the Overlap Filter (see Section 3.3.7). After which, the point cloud data is either saved or sent over the network and finally released from memory as well.

The following sections will give with an in-depth view into each threadable function.

4.3 Point Cloud Extraction

Point Cloud Extraction is a two part process, both of which can be multi-threaded as one sees fit. The first is calculating a sharpness value based on the color image captured.

Extraction then sorts all captures inside an interval by their calculated sharpness value and extracts the highest one.

4.3.1 Sharpness Filter

The functionality of the sharpness filter is mostly provided by OpenCV [Bra00]. Given an image buffer by the Azure Kinect SDK, we create a `cv::Mat` to represent that image. To the best of our knowledge, their is no perfect mathematical way to separate blurry from sharp images when real-time processing in concerned. But we can abstract sharpness by the presence of other features inside the image, by way of detecting sharp edges.

Detecting sharp edges is well researched within computer vision. One method is to use a Sobel filter [KVB88], which OpenCV provides an implementation for. After filtering the detected edges through a threshold, our function gives a score based on the ratio of sharp edges to the amount of pixels. Laplacian filter, also implemented in OpenCV is also able to calculate sharp edges. Compared to Sobel, it only detects radical changes inside an image. In other words, smooth, gradual changes are not detected by Laplacian. The most accurate sharpness value but also the most expensive to compute is calculated using FFT. FFT transforms an image from a spatial to a frequency domain, from 2D to 3D. Inside the frequency domain, high frequencies correspond to edges and higher detailed points. The higher the ratio of those frequencies for the size of an image, the sharper the image should be.

Each of these calculated scores only work when compared to images featuring roughly the same objects. If using Sobel or Laplacian, a blurry image featuring just one edge might score higher than a completely sharp image featuring no edge. This is why we cannot throw away all blurry images, but rather opted to always extract the sharpest image inside an interval.

4.3.2 RGB-D to Point Cloud Transformation

Extraction of 3D information from an Azure Kinect capture goes through various stages. As stated in Section 3.3.3, we do not want to extract all point clouds but rather one per interval. Each capture features a timestamp when it was taken. A thread running the extraction loop starts on the first point cloud and marks it as being processed. Then it calculates the timestamp at the end of the current interval, by adding the extraction rate as milliseconds to the capture's timestamp. Every other thread running the extraction loop will see that the first capture is marked as being processed, calculate the same end of the interval and marks the first capture after that for extraction, if it is already available. Since the rest of the extraction does not take that long anyway, multi-threading this function is really only necessary when operating on a high extraction rate, or on a recording with no other bottlenecks.

After a thread found the start and end index of an interval, it sorts the indices inside that interval by their sharpness value. If the sharpness filter is disabled, it just picks the first capture. The color image of the capture is decompressed from the JPEG storage using libjpeg-turbo [Com23], leading to a color value for each pixel. This is needed, since each pixel seen by the depth camera will be transformed into its own 3D point, with its own color value. After we have a decompressed color image we project the depth onto the color camera using `k4a_transformation_depth_image_to_color_camera`. This not only eliminates the off-set between the RGB and depth camera of the Azure Kinect device, but also changes the depth cameras resolution into that of the color camera's, interpolating between the values to make up for the missing pixels. Afterwards 3D points are extracted from the transformed depth image using `k4a_transformation_depth_image_to_point_cloud`. Now we have 2 buffers of data, one featuring 3D coordinates and one featuring color values of those coordinates. We combine the two into a colored PCLpoint cloud.

But not all 3D coordinates will be valid. As seen in Figure 3.6, Azure Kinect's depth camera does not capture everything the color camera sees. Normally, we would just dismiss those points and end up with an extracted point cloud like seen in 3.6b. But, some of our future steps in the processing pipeline only work on images and not point clouds. For those, the indices of the point cloud must still match the resolution of the image. This is why, if one of the steps is activated, points without depth information are just set to origin (an impossible position to capture), where they will be removed before transformation takes place. Since we still need the point cloud's corresponding image, we store the `cv::Mat` we created for the *sharpness filter* inside a the same struct the point cloud resides in, and free the memory of all captures and their images inside the interval.

4.4 Point Cloud Registration

Registration is done in three steps. Each of the steps is able to build on the previous one(s), or provide a pose estimate independently. It is important to note that none of the registration steps can be multi-threaded, as they are either consecutive algorithms (KinFu and PnP) or run so fast that multi-threading would defeat the purpose (LiDAR registration). In order to save computation time, none of the steps transform the point cloud right away, as another thread takes care of that once it has passed all activated registration steps.

While not a registration step in itself, a final matrix transforming the point cloud from Azure Kinect (image coordinates) to LiDAR coordinates (BIMCheck or world coordinates) as well as the offset between the Azure Kinect and LiDAR positions is added to all calculated transforms.

4.4.1 Registration using KinFu

Registration using KinFu is done before any point cloud is extracted from the capture. Its functionality is implemented using OpenCV [Bra00]. In order to pass a depth image from the Azure Kinect camera into KinFu, one must first rid it of distortion. While the SDK does not feature a function for this, Microsoft provides example code, which we integrated into our component. It sets up a Look-Up Table (LUT) from the camera parameters, which can swiftly transform distorted depth image coordinates into an undistorted one. A `cv::Mat` is created from the undistorted buffer, which can be added to a `cv::kinfu::KinFu` element. If successful, one can receive the estimated pose from the KinFu element, otherwise the pose is reset to origin. Functions outside the KinFu thread can also reset the KinFu object if they think its pose is no longer accurate. The estimated pose is saved in a struct with the capture.

4.4.2 Registration using LiDAR

Like KinFu registation, the estimated LiDAR position is also added before any point cloud is extracted. The problem with using LiDAR poses is two-fold: First, the Ouster LiDAR we are using is operating at a 20Hz rate, whereas the Azure Kinect records at 30fps. Secondly, even if they were operating at the same Hz, one would need to synchronize the two devices perfectly. For this reason, in order to estimate the pose for one of our frames, we interpolate 2 surrounding LiDAR poses by their timestamps. Our implementation is as follows: We request LiDAR and their timestamps from the network at a higher speed than the LiDAR operates. Poses with the same timestamp as the last are skipped. Once we have one LiDAR pose with a timestamp before the frame we are currently processing, and one with a timestamp after that, we can interpolate between the 2 poses. For the translation we can simply interpolate them linearly based on the capture's timestamp. For the rotation we can do the same by converting them to quaternions and using Spherical Linear Interpolation (Slerp). If KinFu Registration is actived, the interpolated LiDAR pose is only used if the rotational or translational difference between it and the KinFu pose exceeds a threshold. If that is the case, KinFu is reset from the LiDAR registration thread. All future KinFu poses are then added to the LiDAR pose where KinFu was last reset.

4.4.3 Registration using PnP

Compared to both KinFu and LiDAR registration, PnP is a bit more computationally expensive. This, combined with the reason that it needs the data of the previous point cloud, is why we run it on each extracted point cloud instead of each received capture. PnP estimates the pose where an image was taken based on a camera's intrinsic parameters as well as corresponding 3D points inside that image. OpenCV [Bra00] provides functionality for both finding features in an image (`cv::ORB`), as well as matching similar features between 2 images (`cv::BFMatcher`). Since we extract our point clouds pixel per pixel, the indices of the extracted point cloud match up with the indices of our image. So for each detected matching pixel in the previous image, we add the corresponding 3D coordinate from the previous point cloud into an array of corresponding world points. We extract the matched pixel coordinates of our current image into another array to create the inputs for the PnP problem. To solve it, OpenCV also provides us with `cv::solvePnPRansac`. If we already have an estimated pose from either KinFu or LiDAR we can vastly improve the results by providing the pose change from the previous to the current point cloud as an initial guess. If we have such an initial guess, we also compare the final PnP result to it. Sometimes, PnP can fail miserable (e.g. due to a low number or mismatching features in the image). So if PnP's result is too far off from our initial guess, we discard it in favour of the LiDAR's (or KinFu's) pose.

4.5 Point Cloud Filtering

Filtering point clouds is by far the most computationally expensive step of the *Azure Kinect data pipeline*. While the *plane segmentation filter* can easily be multithreaded, the *point cloud classification* cannot, due to YOLO-World [CSG+24] already being run multi-threaded on the GPU. The overlap filter can be multi-threaded, but not without caveats, which we'll discuss in its corresponding subsection.

4.5.1 Filter Invalid Points

Each of the filters alters the indices of the point cloud. They are therefore only applied after the corresponding image of the point cloud is no longer needed. In our pipeline, this step coincides with transforming the point cloud based on the estimated camera pose, as the *overlap filter* needs the points to be at their transformed location. It is also at the transformation step, that invalid points discussed in Section 4.3.2 can finally be discarded from the point cloud, already cutting away almost $\frac{1}{3}$ of of its size as seen in Figure 3.6b. In a way, transformation of the point cloud can be seen as the initial and only required filter step, which will speed up all future filters by a large margin.

4.5.2 Classification Filter

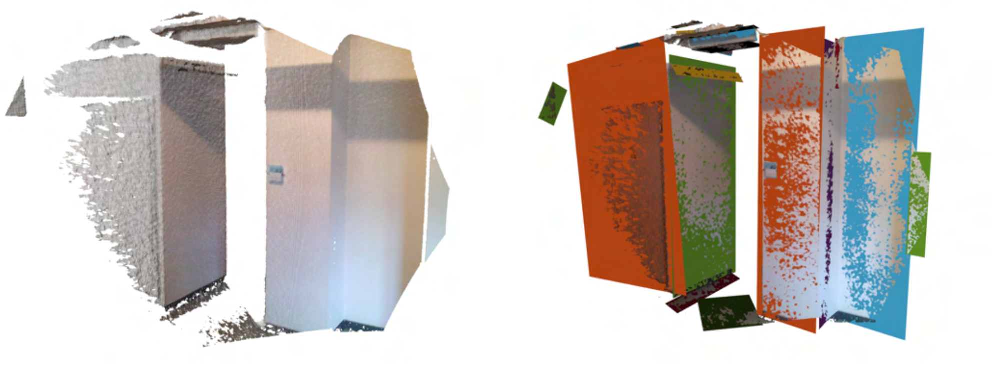
The *classification filter* filters points based on if a class was detected for one of the input prompts using YOLO-World. We can send an image into the network using ONNX Runtime. As a return, we get a 2D image coordinate, as well as a width and height. These together form a rectangle inside the image. Just like with PnP registration, we can directly use image coordinates as indices for our point clouds to find the corresponding 3D points. It is important to note, that the detection step is done **before** point clouds are transformed, such that the indices still match up with the image they were extracted from. The actual filtering takes place after transformation, where we simply throw away all points which were not classified by YOLO-World.

We also implemented a few options for our filter. First of all, one can decide between saving only one class, or up to a maximum of 32 classes per point. If one class is selected, the class with the highest score is set as the `label` of the point cloud's point. Since the `label` attribute the PCL point cloud provides is a 32-bit integer, we can also set each of its bits *on* individually, thus being able to represent all combinations of up to 32 classes. This can be helpful, if one wants to use the detected classes as an initial guess, which a separate, more sophisticated object detection network can improve upon.

Another parameter our filter provides is the maximum number of partitions for a given image. YOLO-World, like many neural networks for image processing, only handle an input size of 640x640. We could just down-sample our image, but in the process would lose a lot of detail that might aid our point cloud classification. For this reason, the maximum number of partitions continues to divide an image by its width or height (whichever is bigger) until the desired amount of partitions is reached, or the partition are already below 640x640. The number and areas of these partitions are calculated once and applied to every image the same. Once separated into the partitions, they can be sent into the neural network one by one. We need only keep track on where inside the original image our current partition starts, so that we can remap its coordinates to our point cloud. But the number of times the network is called for each point cloud also determines how many point clouds we can handle per second, while still keeping the pipeline real-time. For more details refer to Chapter 5.

When one changes the class descriptors (input prompts) of our neural network, YOLO-World does not need to be retrained. However, it needs to be rebuilt into an ONNX file using those descriptors. This can be done in a few seconds using their *huggingface* page, or by building it oneself via Python, both available on their *GitHub* page [CSG+24].

4.5.3 Plane Segmentation Filter



(a) Input point cloud (b) Point cloud with detected planes

Figure 4.3: Plane segmentation example

The goal of the plane segmentation filter is simple: detect planes and remove any points lying on those planes from the point cloud. Our detection is based on Araujo's and Oliveira's code [AO19]. It takes as input a point cloud with information of normals attached to each point. We can estimate the normals by using PCL's `NormalEstimationOMP` [Rus09]. Normal estimation works by providing the sensor's pose and either a radius or set number of neighbors. Given that, a points surface can be estimated by its surrounding 3D coordinates. The vector that points perpendicular away from the estimated surface is the points normal. As one can already guess, even estimating normals is a costly undertaking, which scales linearly by the amount of neighbors checked. To improve runtime, we can sub-sample the point cloud using PCL's VoxelGrid. Depending on a given grid size, only one point is kept for each coordinate inside the grid. Even without the coming steps, filtering to a grid adds virtually no time to the execution of the function, but already vastly increases the estimation of normals. It also makes the decision between number of neighbors and radius redundant, as all points on the grid will roughly have the same number of neighbors around their position.

Algorithm 4.1: Detection of planar patches given an octree of a point cloud

```
Function detect_planar_patches(octree_node):
    if octree_node.points() < MINIMUM_NUMBER_OF_POINTS_PER_PATCH then
        return false
    end
    octree_partitions ← octree_node.partition()
    child_could_have_planar_patch ← false
    for $i = 0$ to 7 do
        if detect_planar_patches(octree_partitions[i]) then
            child_could_have_planar_patch ← true
        end
    end
    if child_could_have_planar_patch then
        return true
    end
    else if octree_node.level() > 2 then
        if PlanarPatch(octree_node).is_planar() then
            add_planar_patch()
            return true
        end
    return false
```

Points bundled with their normals are then sent into the plane detection function. Here we want to group the points into planar patches. We start with an octree node [Mea82] and start our detect_planar_patches described in Algorithm 4.1. Whether a given patch counts as planar is determined by the ratio of outliers in a set of points. Given a set of points, an average plane is constructed using their coordinates. Depending on the outlier ratio, enough points must be under a distance threshold from the plane, as well as have their normals be under a rotational threshold to the plane's normal, but not both at the same time. E.g. if a patch has 2 points and a permitted outlier ratio of 0.51, point A may only satisfy the first condition and point B may only satisfy the second condition. Since both are tested separately, both tests would satisfy an outlier ratio with only 0.5 and the 2 points would still form a planar patch. But all outliers are removed from the planar patch, which in this case would lead to an empty set of points. This edge case will be discussed at the end of this section.

Algorithm 4.2: Growing and merging of planar patches

```
while true do
    // Grow Patches
    foreach patch in planar patches do
        if patch.is_stable() then
            continue
        end
        foreach point in patch do
            neighbors ← get_neighbors(point)
            foreach neighbor in neighbors do
                if neighbor.inside_other_patch() then
                    continue
                end
                if neighbor.satisfies_plane_thresholds() then
                    patch.add_point(neighbor)
                end
            end
        end
    end
    // Merge Patches
    foreach patch in planar_patches do
        foreach point in patch do
            neighbors ← get_neighbors(point)
            foreach neighbor in neighbors do
                neighbor_patch ← neighbor.get_patch()
                if neighbor_patch == NULL OR neighbor_patch ==
                 patch then
                    continue
                end
                if point.satisfies_neighbor_plane_thresholds()
                 AND neighbor.satisfies_plane_thresholds() then
                    merge(patch, neighbor_patch)
                end
            end
        end
    end
    // Update Patches
    foreach patch in planar patches do
        if patch.nr_points_now < patch.nr_points_before * 1.5
         then
            patch.set_stable ← true
        end
    end
    if all_patches_are_stable then
        break
    end
end
```

After we have found our initial planar patches, we set up a k-d tree [Ben75] to find the next n neighbors for each point inside the input point cloud. With them, the planar patches are grown and merged in a loop seen in Algorithm 4.2. Once the exit condition for the loop is satisfied, planar patches are grown one last time, this time without satisfying the normal's rotational difference threshold (only the maximum distance to the plane). Planes are constructed geometrically from the patches, which give them their u and v vectors. As a final step, false positive planes featuring too few points are removed. All other planes' equations and indices of their points are returned. An example of a point cloud and its segmented planes can be seen in Figure 4.3.

If we ran plane segmentation without sub-sampling the full point cloud, we could just create the inverse of the returned indices array (an outlier array) to receive the final indices of our filtered point cloud. If we did sub-sample them, we must first construct another k-d tree on the complete point cloud. Once constructed, we query it using the 3D positions of the outlier indices and extract all points around a given radius into our plane filtered point cloud. The final point cloud then overwrites the one in the point cloud array.

4.5.4 Overlap Filter

The objective of the overlap filter is to remove duplicate points across all point clouds. Compared to the *point cloud classification*, the *plane segmentation* or even the *sharpness filter*, the *overlap filter* should not lose any information doing this. This is why we could not simply use a voxel grid or TSDF to sub-sample our clouds to, filtering overlaps by only keeping one point per grid cell. Besides, such an approach would not be very dynamic, as adding new point clouds would lead to rerunning the process on all combined point clouds, each time. Once a point cloud is out of memory, it would also be impossible to filter new arriving point clouds featuring the same area. For these reasons, we chose to adopt our *overlap filter* based on camera view frustums. A camera's pose can be represented by only 12 32-bit floating point values. So even an array of 1.000.000 camera poses should theoretically only take up 48 megabytes.

The main idea is simple. We have a camera's pose, know its view frustum by their intrinsic parameters and can therefore make out which points are seen inside that view frustum. For this, we only need to project a points' 3D position to the view plane at the points distance from the camera. Azure Kinect's view frustum is unlike other camera though, as seen in Figure 3.6a. It is for this reason, that we add additional checks for each of the depth camera's edges, to see if a point really is inside the depth camera's view frustum. The final Algorithm can be seen in 4.3.

This gives us a way to know which point might be seen in two or more view frustums, but it does not tell us anything about overlaps. Since we want to run this algorithm on as many previous point clouds as possible, constructing a k-d tree and checking each point for a potential duplicate in the other point cloud would be far too expensive. Instead, we try to keep only one point for each coordinate by calculating distances for each point seen

Algorithm 4.3: Determine if a point is inside Azure Kinect's view frustum

```
Function point, cam:
    Vector3 pos_relative_to_camera ← point.pos - cam.pos
    float pos_front_distance ←
      pos_relative_to_camera.dot(cam.front_vec)
    if pos_front_distance < cam.minimum_depth or
      pos_front_distance > cam.maximum_depth then
        return false
    end
    Vector3 proj_pos ← (pos_relative_to_camera /
      pos_front_distance) - cam.dir
    float pos_right_distance ← proj_pos.dot(cam.right_vec)
    if pos_right_distance < -cam.projection_width_half or
      pos_right_distance > cam.projection_width_half then
        return false
    end
    float point_up_distance ← proj_pos.dot(cam.up_vec)
    if point_up_distance < -cam.projection_height_half or
      point_up_distance > cam.projection_height_half then
        return false
    end
    if inside_top_left_fov_cutoff(proj_pos, cam) or
      inside_top_right_fov_cutoff(proj_pos, cam) or
      inside_bottom_left_fov_cutoff(proj_pos, cam) or
      inside_bottom_right_fov_cutoff(proj_pos, cam) then
        return false
    end
    return true
```

in two view frustums to both camera's positions. A point is only kept if it is nearer to the camera that captured it. This also means that points inside both view frustums, but only captured by the camera further away from it are lost as well. To combat extreme cases of this (Figure 3.16), we only compare two view frustums if their rotational difference is below a threshold. In order to speed up computation as much as possible, we also compare view frustums by their volume of overlap. This should filter out the most points first, leading to less points to check for all future comparisons. We do this by abstracting view frustums as spheres and comparing their volumetric overlap [ZSK17] as described in Section 3.3.8.

The final algorithm can be described as follows:

- Sort all previous view frustums by overlap with the current one.

- Remove all previous view frustums whose rotation to the current view frustum is too high.
- Loop start: Filter the **current point cloud** into *seen by the previous frustum* and *not seen by the previous frustum*.
- For all points in the **current point cloud** *seen by the previous view frustum*: Remove them if they are nearer to the **overlapping camera position** than to the current one
- Combine the remaining points with the current points *not seen by the previous view frustum*.
- Filter the **overlapping point cloud** into *seen by the current view frustum* and *not seen by the current view frustum*.
- For all points in the **overlapping point cloud** *seen by the current view frustum*: Remove them if they are nearer to the **current camera position** than to the previous one.
- Combine the remaining points with the previous points *not seen by the current view frustum*.
- Continue loop with next most overlapping view frustum until the current point cloud has been filtered by all previous view frustums and all previous point clouds have been filtered by the current view frustum.

The only thing left to decide is what to do with previous view frustums whose point clouds are no longer in memory. After all, we can still compare the distances of all new points to the old view frustum, but we cannot filter already released point clouds by the new view frustum. For this edge case, the parameters let the user decide what should happen: Either throw away all point clouds seen in the previous view frustums, potentially losing information but keeping the point cloud free of duplicates. Or add points nearer than the previous view frustum to the combined point cloud, not losing any information but keeping duplicates. Points nearer to the old view frustum should always thrown away, as that is the *overlap filter* working as intended.

The *overlap filter* is still very computationally expensive though, depending on how many previous point clouds we want to take into account. It can be multi-threaded, however this does not speed up computation as much as multi-threading the other steps of the pipeline. This is because every other step works on only one point cloud at a time, which is why doubling the amount of threads roughly doubles the output a step can achieve. Every *overlap filter* thread must work on the same previous point clouds, with much time spent on waiting for the other threads to finish. We optimized this by skipping over previous frustums currently being filtered by other threads, adding their indices to a separate array. We then re-run the loop with all skipped indices until all threads have filtered each previous point cloud. However, as soon as 3 or more threads are introduced, they tend to wait on the same point clouds, massively impacting the overall performance. Each thread altering a point cloud also needs exclusive writing access to the whole point cloud array. Due to these reasons, the overlap filter's runtime can only be somewhat improved. But depending on the parameters, real-time applicability without overlaps can still be provided. For more information and results, see Chapter 5.

4.6 Visualization

The Visualization does not play any part in reaching our set-out goals. Still, it was hugely important during debugging and can still be used to verify a correct set-up quickly. It is split into two modes. One is for real-time visualization of what is being captured, and one for analyzing the results frame by frame.

4.6.1 Visualization during Processing

Live visualization works on a chosen pipeline step and set number of maximum point clouds to show during recording or processing. The pipeline steps available for live visualization only cover point clouds after transformation or filtering though, as before point clouds would just get stuck on-top of one another. Figure 4.4 shows the live visualization.

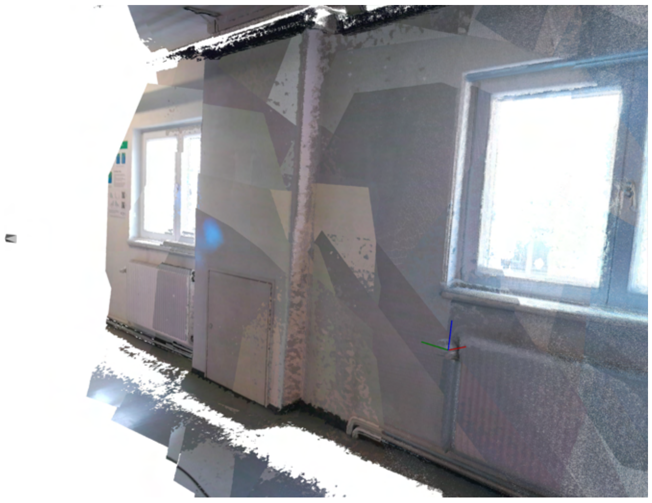
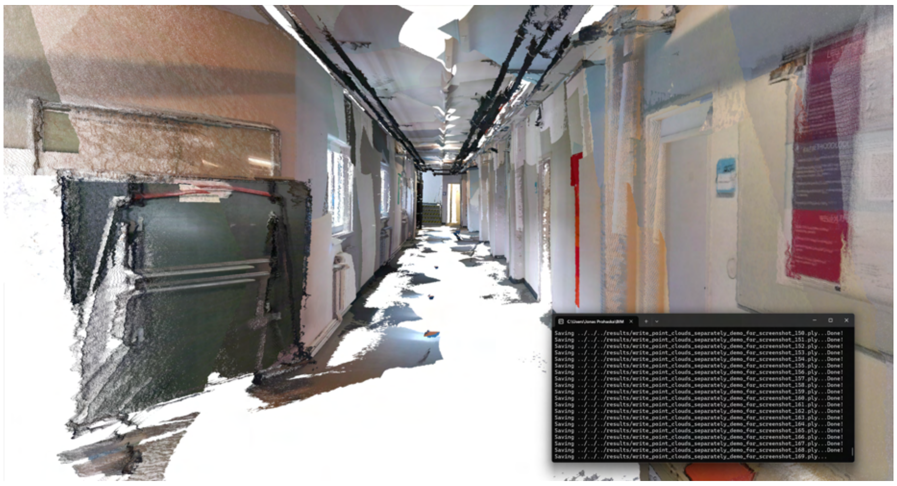


Figure 4.4: Live visualization. The combined point cloud is constructed as new captures are processed. The grey pyramid represents the last estimated camera position.

Inside the pipeline step chosen for the visualization, each point cloud is copied into the set-up visualization class. Copying ensures that the point cloud inside the visualization stays the same, even if it is further filtered or even removed from memory afterwards. The visualization has an array the size of the maximum number of point clouds. If the set-up visualization is already at maximum capacity, the function that adds point clouds simply starts inserting at index 0 again, automatically removing the last point cloud from the visualization and memory. Inside the visualization window, a user may rotate, move, or zoom in or out of the currently displayed point clouds. A gizmo showing the current estimated pose is also shown. Once normal operation is confirmed, the visualization window can be closed, releasing all memory and processing power to the other parts of the pipeline to use.

4.6.2 Visualization after Processing



```
Saving ../../../results/write_point_clouds_separately_demo_for_screenshot_150.ply...Done!
Saving ../../../results/write_point_clouds_separately_demo_for_screenshot_151.ply...Done!
Saving ../../../results/write_point_clouds_separately_demo_for_screenshot_152.ply...Done!
Saving ../../../results/write_point_clouds_separately_demo_for_screenshot_153.ply...Done!
Saving ../../../results/write_point_clouds_separately_demo_for_screenshot_154.ply...Done!
Saving ../../../results/write_point_clouds_separately_demo_for_screenshot_155.ply...Done!
Saving ../../../results/write_point_clouds_separately_demo_for_screenshot_156.ply...Done!
Saving ../../../results/write_point_clouds_separately_demo_for_screenshot_157.ply...Done!
Saving ../../../results/write_point_clouds_separately_demo_for_screenshot_158.ply...Done!
Saving ../../../results/write_point_clouds_separately_demo_for_screenshot_159.ply...Done!
Saving ../../../results/write_point_clouds_separately_demo_for_screenshot_160.ply...Done!
Saving ../../../results/write_point_clouds_separately_demo_for_screenshot_161.ply...Done!
Saving ../../../results/write_point_clouds_separately_demo_for_screenshot_162.ply...Done!
Saving ../../../results/write_point_clouds_separately_demo_for_screenshot_163.ply...Done!
Saving ../../../results/write_point_clouds_separately_demo_for_screenshot_164.ply...Done!
Saving ../../../results/write_point_clouds_separately_demo_for_screenshot_165.ply...Done!
Saving ../../../results/write_point_clouds_separately_demo_for_screenshot_166.ply...Done!
Saving ../../../results/write_point_clouds_separately_demo_for_screenshot_167.ply...Done!
Saving ../../../results/write_point_clouds_separately_demo_for_screenshot_168.ply...Done!
Saving ../../../results/write_point_clouds_separately_demo_for_screenshot_169.ply...
```

Figure 4.5: Debug visualization. Individual point clouds can be added, removed, saved or even plane segmented as one sees fit.

For more precise debugging, our program also features the option to display all point clouds at the end of the pipeline, if releasing of memory during processing is disabled. Figure 4.5 shows this more sophisticated visualizer, where the user has a number of debugging options: One can switch between combined and single point cloud view and choose or remove specific ones from the combined view. Any combined or single point cloud can be written out to the hard drive. When saving a combined one, there is an option for saving each point cloud individually. Plane segmentation can be run, opening an additional visualization showing the planes. For more information, see Chapter 6.

If one alters the code, the `Visualization.hpp` also provides helpful functions for visualizing a single or multiple point clouds (including all debug features discussed above) anywhere in the program using a single line of code. They are appropriately named `visualize_single_point_cloud` and `visualize_multiple_point_clouds`.

CHAPTER 5

Evaluation

In order to test the performance of our designed software we conducted an evaluation, in which recordings and reconstruction results based on them were compared against ground truth data. We chose to evaluate on recordings rather than live sessions in order for the results to be repeatable. Testing on recordings also lets us directly compare different parameters of our *RGB-D data component* pipeline. Real-time applicability for a live setting was asserted additionally, with no differences detected between the live results and an offline re-run on the same recording afterwards. While the highest-resolution setting could not be run in real-time on our hardware, we found lowered settings still adequately accomplished the goals set by the thesis. Our evaluation can be broadly separated into three distinct categories:

- **Registration accuracy**

  In which the registration is compared to ground truth data.

- **Compression rate**

  In which our various filters are compared based on the percentage of data reduction versus the loss of important data.

- **Runtime performance**

  In which the times needed for each step of the pipeline (and their variations depending on parameters and recording settings) are examined.

Section 5.1 will explain our evaluation methodology for each of the categories in detail. It will also describe the environments as well as the motivation for choosing them. After we defined our ground truth, baselines and settings to test and compare, they are evaluated in Sections 5.2 and 5.3. Section 5.4 will show average run-times for each pipeline step as well as how multi-threading can improve their runtimes. While each of the separate sections will already elaborate on each finding, Section 5.5 will discuss the conclusions drawn from the results more generally. This discussion will also show the trade-off between accuracy and performance the user-set parameters provide, highlighting their benefits and shortcomings. It will also show the limits of real-time applicability for our tested hardware. Thus it shall also act as a comprehensive guide for adjusting the parameters to one's own liking using the information in Chapter 6.

5.1 Methodology

This section will describe the set-up and methodology of our evaluation. We will first describe the environments in which the recordings were conducted. We then describe how we use the ground truth data of the environment to set-up a baseline for all alterations of settings our program provides. The following subsection will then describe how each of them builds on the results of the previous ones in order to thoroughly test accuracy, compression as well as runtime performance.

5.1.1 Testing Environments and Recording Approach

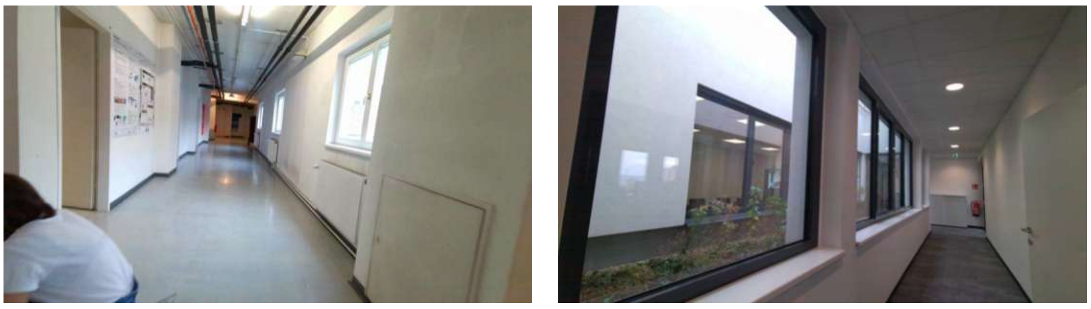
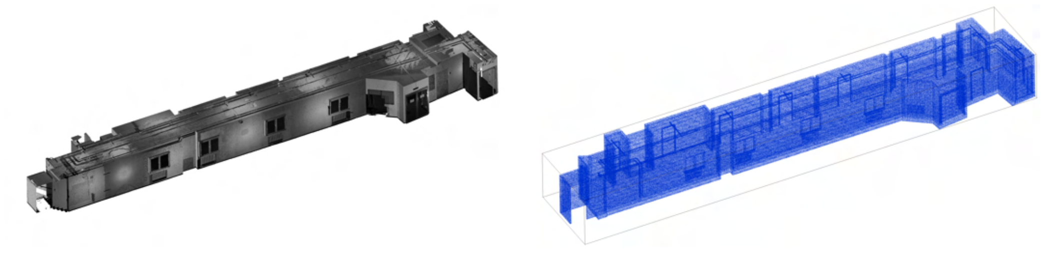

(a) TU Wien hallway (b) Large office space

Figure 5.1: Evaluation environments

We chose to test our implementation in two different set-ups. The first one is an empty hallway, which we think best represents the typical environment our software will be used in. We chose one at the TU Wien for this, as we have a corresponding BIM model of it. While it is devoid of furniture and people, it features doors with light switches, power sockets and fire dampers on one side, while windows and radiators fill the other side. Fire extinguishers and pipes are scattered across it. Our second environment is a roundabout loop through an office building, featuring a room in the middle. We also chose to record this environment at the highest data rate the Azure Kinect provides, which is 3840x2160 pixels at 30 fps. With this, we will evaluate the scalability of our system. Images of the two environments can be seen in Figure 5.1. The two recordings were done without taking into account how the underlying system will handle the data, i.e. how an untrained personnel would probably record it. We did not move the camera around too cautiously, but we did not flail it around either. Likewise, while we captured the walls in their entirety, ceilings and especially floors were not captured completely, as the Azure Kinect's field of view requires it be rotated quite unnaturally to catch those. Besides the pipes seen in Figure 5.1a, ceilings and floors hardly feature the information we care about anyway.

5.1.2 Registration Accuracy


(a) Ground truth obtained from a state-of-the-art Leica LiDAR scanner

(b) Ground truth obtained from our BIM model converted into a point cloud

Figure 5.2: Ground truths for our hallway environment

In order to evaluate our registration quality, it is essential to compare our program's output with the ground truth provided by the environment's BIM model. This comparison and evaluation is intended to be the final step of the BIMCheck project. However, as of the time of writing, this component has not yet been implemented. Therefore, we had to evaluate registration quality using other software. CloudCompare [GM24] features a *Cloud-to-Cloud Distance* metric, where an input cloud is compared to a reference cloud. For each point inside the input cloud, the nearest points inside the reference cloud are found. The application then computes a surface for the neighboring points in the reference cloud. Only then is the euclidean distance to that surface calculated. After doing this for all points, the program returns a mean distance (μ), as well as a standard derivation (σ). Roughly 68% of all our registered points will lie between $\mu \pm \sigma$, 95% between $\mu \pm 2\sigma$ and 99.7% between $\mu \pm 3\sigma$. All values will be given in centimeters. In order to use our BIM model's ground truth for this function, we must first convert it to a point cloud. This is also done using CloudCompare, which can sub-sample our BIM mesh to a desired amount of random points. We chose 100.000.000 points, which we deemed dense but still reasonably fast to evaluate with. Besides BIM models, we also have scans of the TU Wien hallway from a state-of-the-art Leica LiDAR. Since we have two ground truths for this environment, in-depth registration evaluation will focus on the TU hallway. They can be seen in Figure 5.2. The registration will be evaluated both using a sub-sample of a complete point cloud, as well as one filtered by the *overlap filter*. We will start by only taking LiDAR SLAM's pose estimation as a baseline, and continue to introduce KinFu and PnP as additional registration afterwards.

5.1.3 Data Compression Rate

Using the point cloud filtered by the *overlap filter* as a baseline, the *object detection* and *plane segmentation filter* will be tested on their compression rate as well. Besides percentage of compression, both will be further tested on loss of information (i.e. removal of objects of interest). After defining the objects of interest, presence of them will be checked manually for the filtered point clouds. Different parameter settings for each of the filters will show how far their results can be improved depending on one's set of requirements. A trade-off between reduction rate and amount of data lost will be shown, while visual examples accompany the data.

5.1.4 Runtime Performance

Since many of our pipeline's steps are far from being lightweight, runtimes will be tested using an Intel Core i9-11900k CPU, 32GB of RAM, and an NVIDIA GeForce GTX 3090 GPU. For this we compiled our program using MSVC and CMake's MinSizeRel build type. Runtimes for each of the components will be tested on the 3 highest resolutions the Azure Kinect provides, as well as how much multi-threading can improve each of our pipeline's steps. Additional focus will be lain on the biggest performance bottleneck: the three filters and their lavish parameters, which were tested in the previous sections. Using the results of the previous sections, a direct correlation between accuracy and runtime performance for the pipeline and recording parameters will be shown in Section 5.5. We will also list what we think are the best set of parameters for the real-time use of our component.

5.2 Registration Accuracy

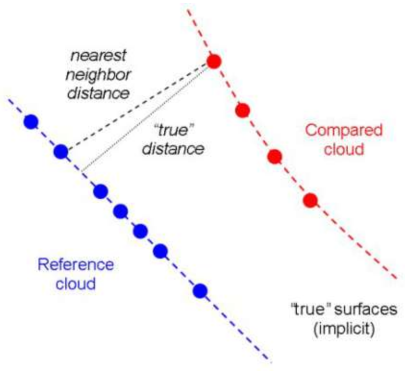
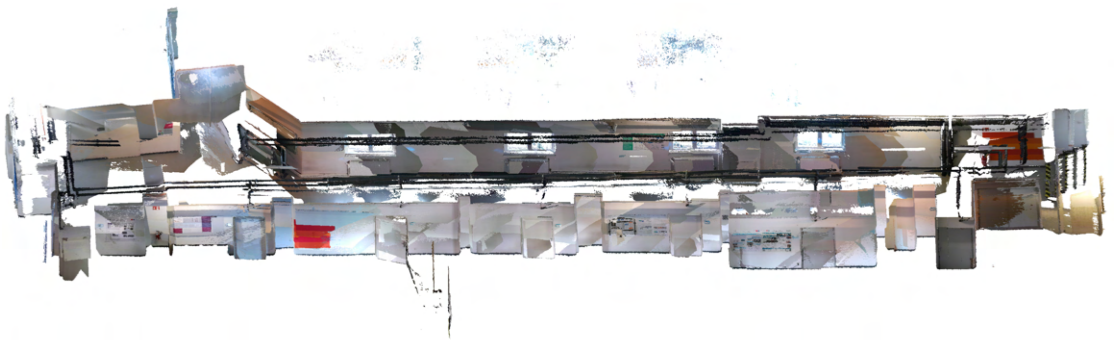


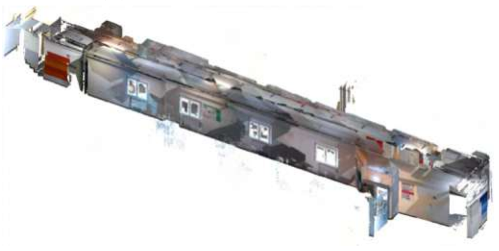
Figure 5.3: CloudCompare's distance calculation between two point clouds [GM24]

In this section the results of our *Azure Kinect data pipeline* will be compared to a ground truth BIM model. For our first environment, the TU Wien hallway, we even have additional ground truth from a state-of-the-art Leica scanner. Registration accuracy was tested by importing both the ground truth and resulting point clouds into CloudCompare [GM24]. From there we used ICP to align the two point clouds. Once correct alignment was confirmed manually, we used CloudCompare's *Cloud-to-Cloud Distance* function, which returns a mean distance and standard derivation between the two point clouds. It does this by calculating distances from every point in our resulting point cloud to the nearest points in the reference point cloud. But instead of just picking the nearest point, CloudCompare generates a surface between neighboring points, whose closest distance to the point is taken instead. Figure 5.3 explains this procedure visually.

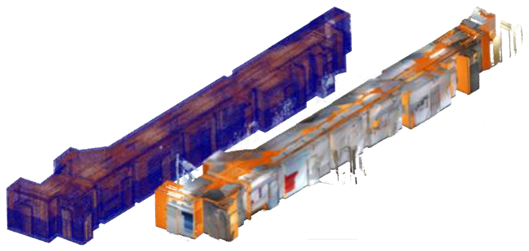
5.2.1 TU Wien Hallway Environment



(a) Complete output cloud from our *Azure Kinect data pipeline* without ceiling



(b) Complete output cloud with ceiling



(c) Comparison of BIM (blue), Leica cloud (orange) and own overlap filtered (colored/grey)

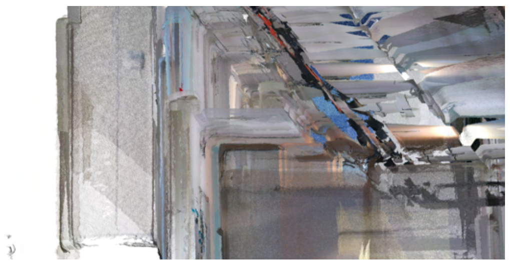
Figure 5.4: Registration accuracy results. (a) is the complete cloud with its ceiling removed. (b) features the ceiling. (c) compares the best results from Table 5.1.

First we compared the point cloud obtained from the Leica scanner to the BIM model. As can be seen in Figure 5.4c, they did not align quite as well as we would have hoped, with a mean distance of 4.5cm and a standard deviation of 5.4cm. We suspect this is because of the different positions the walls occupy in both ground truths. The BIM model features points to represent their outsides, whereas the Leica scanned them from the inside. Notice how the blue BIM point cloud clearly encapsulates the orange Leica LiDAR cloud. This cannot be due to a scaling difference, as the windows still match perfectly. Because of this rather big difference between the two ground truths, we compared and evaluated each of our resulting point clouds against both of them.

Another problem arose from the sheer size of our point clouds if they are output unfiltered. CloudCompare crashed every time we tried to compare our unfiltered point clouds to either the BIM or Leica ground truth. Therefore, we could not compare the combined raw point clouds extracted from the Azure Kinect. We circumvented this by randomly sub-sampling the clouds inside CloudCompare to 100.000.000 points, the same amount of points inside the BIM model's converted point cloud. Random sampled point clouds should give an equally accurate mean registration comparison as the full point cloud. We then additionally compared the results of CloudCompare's filters to a point cloud ran through our own *overlap filter*. We did this by comparing each against the BIM and the Leica ground truth both. Results when using only LiDAR SLAM poses for registration can be seen in Figure 5.1.

	mean distance	std deviation
Leica vs. BIM	4.54cm	5.44cm
Random sub-sample vs. Leica	4.47cm	8.95cm
Random sub-sample vs. BIM	3.42cm	8.98cm
Overlap filtered vs. Leica	2.19cm	7.19cm
Overlap filtered vs. BIM	3.69cm	8.85cm

Table 5.1: TU Wien hallway registration accuracy. Distances between two point clouds measure the error to the compared ground truth, therefore less is better. Leica is taken from a state-of the art scanner. BIM is the BIM model ground truth compared to a point cloud.


(a) Doubled walls and corners in the complete point cloud

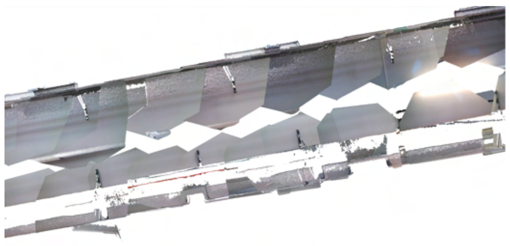
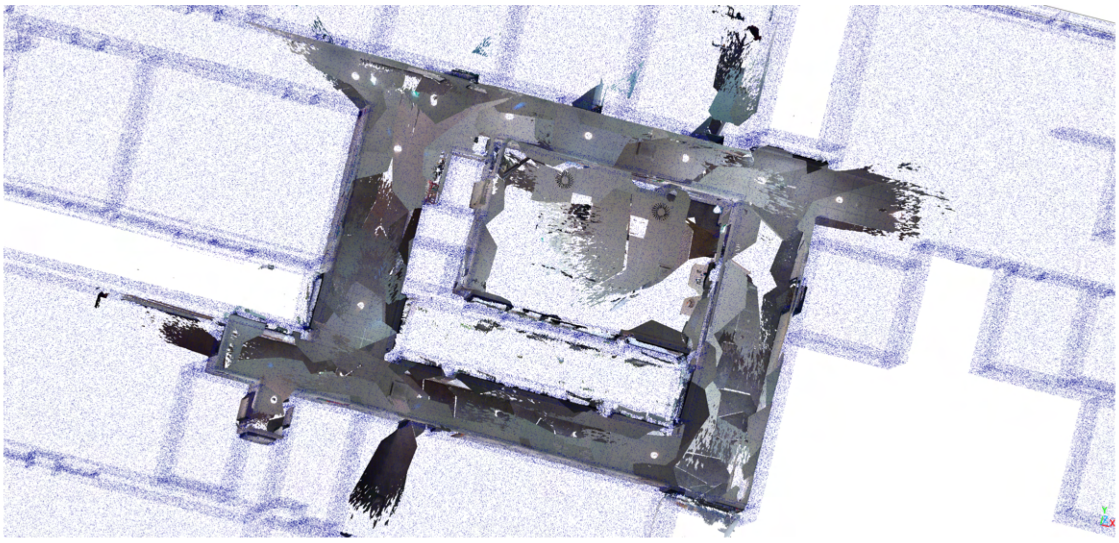
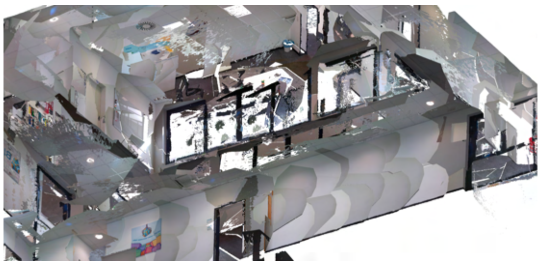

(b) Top view of misaligned walls on the bottom even after using the *overlap filter*. The outlier on the bottom left is not an error but an open door.

Figure 5.5: Registration errors

When looking at the results of Table 5.1, the distances can seem kind of random at first. The comparison between the Leica and BIM ground truth show the least standard deviation, which makes sense, since those two point clouds are much more accurate when it comes to the plumbness and alignment of their walls. Compared to our point clouds, a wall inside the Leica and especially the BIM point cloud is as straight as it gets. Meanwhile our randomly sub-sampled cloud features a lot of doubled walls, as can be seen in Figure 5.5a. Our own *overlap filter* actually performs best compared to the Leica point cloud. This is very much an effect of the *overlap filter*: since it only constructs a point cloud out of points which are nearest to their respective camera position, their depth accuracy is increased, at least on average. It therefore makes sense that it matches the Leica cloud more closely, especially compared to the random sub-sampled one. That its mean error is less than the the mean error between Leica's and BIM's point cloud shows that our own point must be somewhere between the two. As can be seen in Figure 5.5b, the *overlap filter* still cannot combat all of the Azure Kinect's shortcomings, as walls still have differing depth positions, leading to the error measured.

5.2.2 Office Environment


(a) Our output cloud laid on-top of the office's BIM model.


(b) Close view

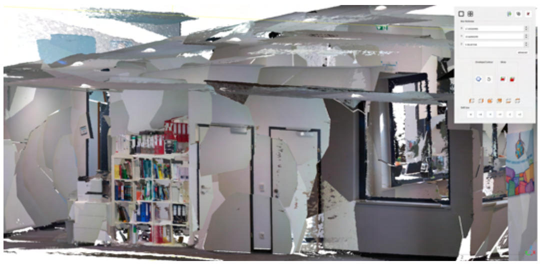

(c) Inside view

Figure 5.6: Office environment registration results

Our set-up performed slightly worse in the larger office model. When comparing our point cloud received from our *overlap filter* (seen in Figure 5.6) to the converted BIM point cloud we calculated a mean distance of 6.74cm and a standard deviation of 9.16cm. However, we do not think this is a failing of the scaling, as the overlap between our cloud and the BIM model in Figure 5.6a seems about as well aligned as our hallway environment. Rather, there are some obvious differences between the BIM model and our captured point cloud. Compared to the hallway environment, this point cloud features furniture, whereas the BIM model features none. What is more, this BIM model does not even feature any installations in them, as did the hallway BIM model. So any table, cupboard or fire extinguisher negatively affects the mean distance.

Of course, this is not to say our registration is not without issues. Some finer misalignments like in the TU hallway environment still apply. But we could not detect any differences by moving further away from the starting points. Of course, windows give each 3D sensor issues, and incorrectly detected points outside of them as seen in Figure 5.6b do indeed correctly count towards the mean distance error.

5.2.3 KinFu and PnP Registration

We tried to further improve registration accuracy by utilizing KinFu and PnP registration. But at least for our chosen point cloud extraction rate (slightly below 1 sec), their results were indifferent or worse to what was achieved when just using LiDAR. We expected that we could never accurately register a whole building by only using the bare KinFu or PnP functions as they come from OpenCV [Bra00]. LiDAR was therefore always intended to be a fallback, should their pose estimations differ too much from the LiDAR's. But the results were still disappointing. At an extraction rate below 1 second, KinFu turned out to be useless. When more than 30 frames pass between extraction of two point clouds, KinFu's pose estimation was almost always deemed too inaccurate by our LiDAR fallback. It was therefore reset and did nothing to register the two extracted point clouds. PnP performed better, but only slightly. The most it can do is make some misalignments less worse (as seen in Figure 3.8), but it did not outright fix any of them. If that were all it did, we would still call it at least a small success. But for it to provide any benefit at all, the extraction rate needs to be set quite high too, which comes not only with big performance trade-offs, but also to more misaligned walls as seen in 5.5. We will discuss these results in more detail in Section 5.5.

5.3 Data Compression Rate

This section will evaluate the data compression rate of our three filters. But while the reduction of a point cloud's size is their main task, it is even more important that they do not lose any object the LiDAR cannot catch in the process. For our TU Wien hallway this includes the following objects: 4 light switches, 3 power sockets, 2 fire dampers, all of the pipes and to a lesser extent 2 fire extinguishers as well as 4 radiators. We say lesser extent, because fire extinguishers should also be visible in the LiDAR data. The radiators are definitely visible in them, as we know from detailed examination. The office environment only features less objects, which is why we will focus on the hallway environment, and only briefly list results for the office at the end of each section.

The baseline for an initial reduction rate comes from our own *overlap filter* seen in Figure 5.7. Since it does not lose any information (at least in environments without occlusions), its reduction to only 1.41GB of 5.96GB (or 23%) shall serve as a baseline. If any other filter goes above this data limit or loses some of the important objects in the process, it will be deemed worse than the *overlap filter*. Of course, the total point cloud size (in this case 5.96GB) largely depends on the extraction rate set. But the *overlap filter* is unique in this regard, as any extraction rate should be reduced to roughly the same final size, if every point cloud is filtered with enough previous poses and point clouds. Each other filter reduces the point clouds independent of another, resulting in each filtered overlapping area retaining twice as many points or more.

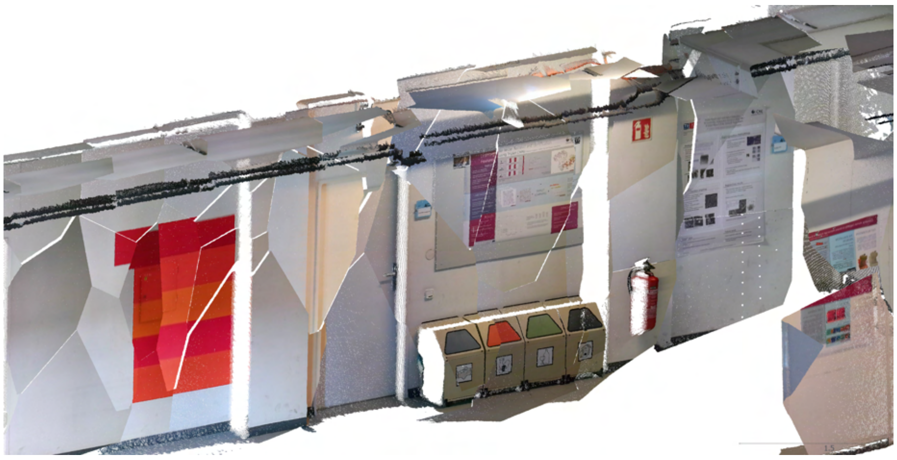
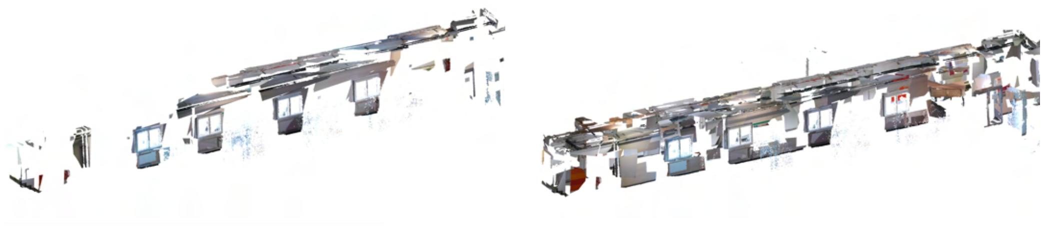


Figure 5.7: TU hallway reduced by *overlap filter*

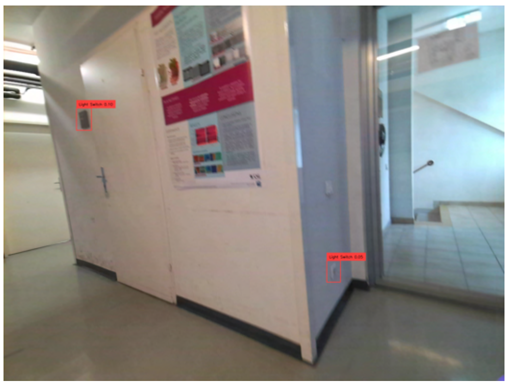
5.3.1 Point Cloud Classification Filter



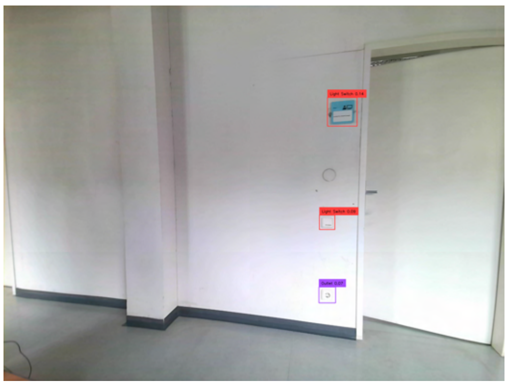
(a) Using 1 partition per image (b) Using 12 partitions per image

Figure 5.8: *Point cloud classification filter* results. Number of objects detected can be seen in Table 5.2. Increasing the number of partitions increases the number of true positive as well as false positive classifications.

Our *point cloud classification filter* detects objects inside images, and reduces their corresponding point clouds by all points not classified as anything. It works by querying an image for prompts. We set-up YOLO-World [CSG+24] to look for the following prompts: *Outlet, Light Switch, Radiator, Pipes, Fire Extinguisher, Fire Damper.* We chose *outlet* instead of *power socket*, as it seems to refer more precisely to what we are looking for inside the hallway. We will continue to refer to them as power sockets, as we have been doing so far. Aside from the prompts, there are only two parameters that changes our filter's behavior. One is the accuracy threshold, which simply sets the required accuracy for a detection to actually count. Too little and all things might be detected as light switches. Too high and nothing will ever be. We chose a detection threshold of 5%, which is the default in YOLO-World's demo. The more impacting parameter sets the amount of partitions we want to separate our images into, before sending it into the network. Since YOLO-World only takes images of 640x640, we can choose to down-sample our 2048x1536 image to 640x640, or split it into as many partitions as we like, detect classes in each of them and combine their results afterwards. We tested for 1 partition, 4 partitions and 12 partitions. The last one represents the maximum amount possible without up-sampling the image, and is also perfectly split-able into the 4:3 aspect ratio of the image.



(a) False negative of a light switch. A false positive saves a power socket to be filtered as well.



(b) Fire dampers (white circle in the middle) did not respond to any prompt input.

Figure 5.9: Point cloud classification errors

Table 5.2 shows the reduction as well as data loss rate of the *point cloud classification filter*. A visual comparison between using 1 and 12 partitions can be seen in Figure 5.8. While the maximum amount of chunks almost catches all of the objects of interest, its reduction rate is only slightly lower than that of the *overlap filter*. The light switch that was missed was also the one that was furthest away from the camera, where possibly no image captured was sharp enough to detect it (see Figure 5.9a). The fire dampers are another problem though. In our TU hallway they appear as white circles on an already white wall. No matter how we tried to prompt the network (e.g. *white circle on wall*), the network could never detect them (see Figure 5.9b). When using 12 partitions, one got wrongly detected as a light switch, but the other was still missed. Pipes are another problem. As can be seen from the results, they were fully captured when exactly 4 partitions were used. When using the maximum of 12, some of their areas were missed.

This is because if a part of a pipe appears in a small chunk of an image, it might not get detected as such. This shows another downside of the object detection filter. To get the best results for each prompt descriptions, different partition sizes of images should be used. Even though the maximum partitions captured the most objects, it also reduced the point clouds by the least amount. It seems as though increasing the true positive rate comes with increasing the false positive rate.

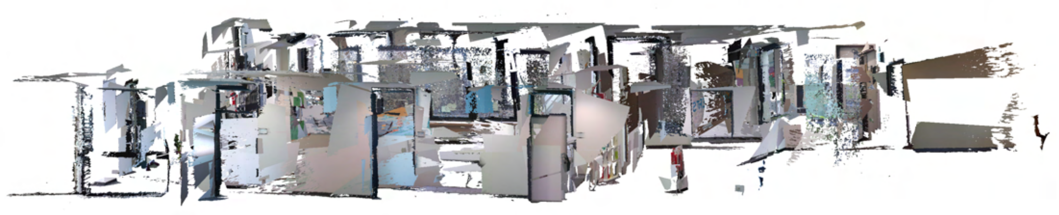
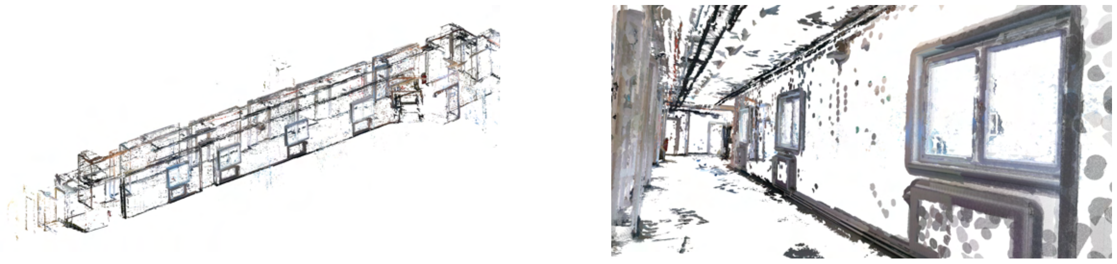


Figure 5.10: Office environment filtered with the *point cloud classification filter*

The *point cloud classification filter*'s reduction compared to the *overlap filter* is much bigger for our office environment. The *overlap filter* reduces the complete cloud from 30GB to 12.6GB. Point Cloud classification running on 18 partitions (forming a 6x3 grid for our 16:9 recording) reduces it to just 3.89GB. This is because this environment features much more empty space, with only 5 power sockets, 3 light switches and 2 fire extinguishers to detect. It correctly detected all but one power socket, which was once again only captured too far away. Results can be seen in Figure 5.10.

5.3.2 Plane Segmentation Filter



(a) Filter with no keep radius (b) Filter with keep radius of 3mm

Figure 5.11: Plane segmentation filter comparison between different keep radii. The keep radius decides how big of an area is kept around each point not on any detected plane. A bigger radius leads to more points being kept and potentially less objects being lost. For results see Table 5.2.

The *plane segmentation filter* has a few adjustable parameters, but from our tests only two of them significantly alter the outcome: the voxel grid size to which an input point cloud is sub-sampled to and the radius that specifies a spherical area to keep around each point not on any plane. This radius is used to query and extract points from the full point cloud around each of the sub-sampled cloud's points remaining after filtering planes. This is why clouds filtered with the *plane segmentation filter* as seen in Figure 5.11 have so many dots in them. Without the radius, most objects would be completely lost, even when not using any sub-sampling beforehand, as seen in Figure 5.11a. While the keep radius is a trade-off between point cloud reduction and keeping of information, the voxel grid sub-sample is a trade-off between performance and accuracy. When testing a grid size significantly bigger than 1cm, the *plane segmentation filter* is already losing a lot of small objects as can be seen in Figure 5.12b. As Section 5.4 will show and Section 5.5 will discuss, a sub-sample grid size of 1cm can be handled quite efficiently.

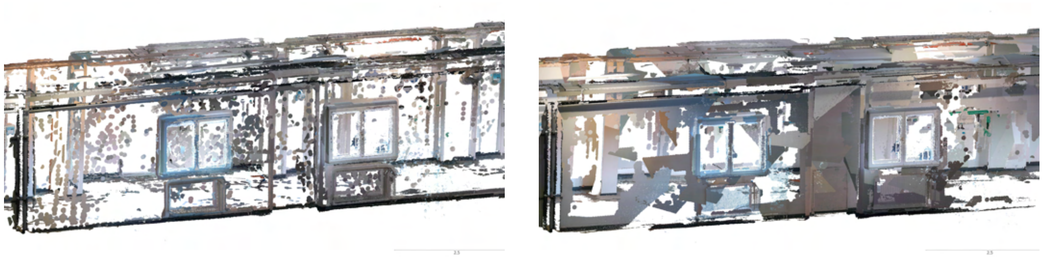

(a) Sub-sample to voxel grid of 1cm. (b) Sub-sample to voxel grid of 5cm.

Figure 5.12: Differences in sub-sampling voxel grid sizes for the *plane segmentation filter*. Even though (b) has almost twice as many points as (a), the light switch and power socket inside the left window got filtered out.

Table 5.2 summarizes results using a voxel grid size of 1cm, and keep radius sizes of 1mm, 2mm and 3mm for the TU hallway environment. While 1mm achieves a reduction of 83%, it sadly loses one of the power sockets. Just as with the light switch not caught by the *point cloud classification filter*, this is the one furthest away from the Azure Kinect. A less sloppy recording might achieve good results for 1mm as well. 3mm features all of the objects but is already bigger than the cloud reduced by the *overlap filter*. A radius of 2mm achieves the best result out of all the filters, while keeping all the objects. It shall be noted that big objects featuring planes on themselves (radiators) might be butchered by this filter. This should not worry us, as they will either be captured by the LiDAR or can be reconstructed using the calculated plane equations.

The office environment was reduced to 6.13GB using a voxel grid size of 1cm and a keep radius of 2mm. This is almost twice as much as the *point cloud classification filter* (3.89GB), but twice as small as running the *overlap filter* (12.6GB) and a fifth from the complete cloud (30GB). Like the *point cloud classification filter*, it also missed a power socket which was only captured too far away. Results can be in seen in Figure 5.13.

	OF	CF1	CF4	CF12	PSF1	**PSF2**	PSF3
Light switches	4/4	1/4	2/4	3/4	4/4	**4/4**	4/4
Power Sockets	3/3	3/3	3/3	3/3	2/3	**3/3**	3/3
Fire Dampers	2/2	0/2	0/2	1/2	2/2	**2/2**	2/2
Pipes	100%	50%	100%	66%	100%	**100%**	100%
Fire Extinguishers	2/2	2/2	1.5/2	2/2	2/2	**2/2**	2/2
Radiators	4/4	4/4	4/4	4/4	4/4*	**4/4***	4/4*
Total size (in GB)	1.41	0.55	1.17	1.32	1.00	**1.27**	1.48

Table 5.2: Reduction and data loss rate for tested filter configurations. Unfiltered size is 5.96GB. OF = Overlap filter. CF(x) = Classification filter (number of partitions). PSF(x) = Plane segmentation filter (keep radius size). *Only the outlines of radiators were kept when using the *plane segmentation filter*. However, we believe this should be enough to recognize them as such.

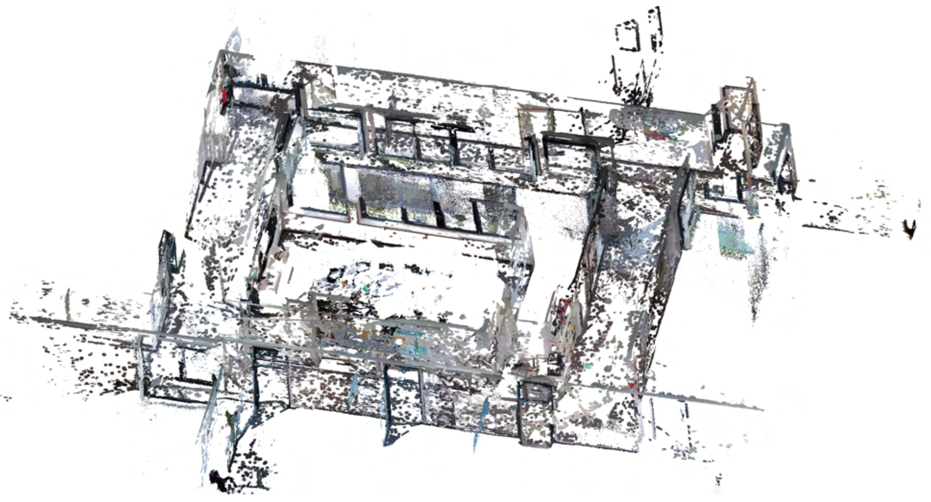


Figure 5.13: Office environment filtered with *plane segmentation filter*

5.4 Runtime Performance

Runtime optimization played an important part in the designing of our software. Even though not all parts of the BIMCheck system run in real-time, we wanted to make sure our *RGB-D data component* at least features a mode for real-time processing and visualization. But that real-time mode comes at a trade-off. More often than not, cheaper algorithms simply produce worse results. Ideally, a balance can be struck: since we know how many captures we need to process (30 per second) and can set how many of those get turned into point clouds (extraction rate), our pipeline's other parameters can be adjusted s.t. each capture and point cloud is still being processed in real-time.

This part of the evaluation shall not only inspect how fast our software performs its tasks, but also act as a reference guide to anyone wanting to optimize our software. Of course, the biggest factor for runtimes remains the underlying hardware used to run the software. The following runtimes stem from a system using an Intel Core i9-11900k CPU, 32GB of RAM, and an NVIDIA GeForce GTX 3090 GPU. If you use significantly better or worse hardware than ours, you will need to scale the runtimes accordingly.

We will first start with a summary of runtimes of the general pipeline steps. These cannot be adjusted by any additional parameters. After that, an additional section will be dedicated to the most computationally expensive part of our pipeline, the point cloud filters, and how their parameters can significantly increase or drop their runtime.

5.4.1 General Runtimes

This section covers the more basic steps of our *Azure Kinect data pipeline*. We say more basic, as their run-times cannot be adjusted by any parameters. Four of the steps are essential for each running of the pipeline. First, images need to be copied from the Azure Kinect queue to RAM. We call this step copy from queue. Next, the copied images need to be moved into the processing pipeline, which we call *move captures*. Once inside, point clouds need to be extracted by *point cloud extraction* and finally transformed by *transform*. From those, only the extraction of point clouds can be multi-threaded, so the others can therefore be seen as constant. Each capture will need to be copied from the queue taking a specific time, and each extracted point cloud will likewise be transformed.

Tables 5.3 and 5.4 show the minimum required time for each capture and point cloud processing step respectively. We say minimum as these times were taken by adding exactly one additional thread to the four required threads described above and timing their average runtime for each resolution. Once more steps of the pipeline are activated or additional threads are added for each step, their performance drops as the CPU needs resources to manage the threads as well. This is important to keep in mind, as some steps cannot work in real-time on a single thread. For example, the FFT sharpness calculation is too slow to handle 30 frames per second at a resolution of 2048x1536, since frames arrive every 33ms. But activating 4 FFT sharpness calculation steps increases their runtime to 130ms each. Since they handle 4 frames simultaneously, i.e. 132ms of data, the whole batch taking 130ms to process is still real-time applicable. It would be impossible to document each possible variation of parameters and time each step's respective runtime, so using these minimum execution times as a guidance seems like the next best thing. We will give a set of parameters working in real-time for each of the evaluated resolutions in Section 5.5.

When looking at Table 5.3, a few things stand out. First of all, both KinFu and LiDAR registration take the same time to execute indifferent of the resolution. This makes sense, as the resolution only affects the color camera. Azure Kinect's depth camera always has a fixed resolution of 640x480, which is the only input KinFu uses. LiDAR registration does not use any data from the Azure Kinect at all, it only interpolates poses it gets from the network. The network latency is left out here, making the LiDAR take less than 1 millionth of a second. One can expect the network latency to be as high as 50ms (our

	2048x1536	3840x2160	4096x3072
Copy from Queue	0.27ms	0.29ms	0.58ms
Move Captures	0.15ms	0.19ms	0.09ms
KinFu Registration	7.9ms	7.9ms	7.9ms
LiDAR Registration	$<1\mu s$	$<1\mu s$	$<1\mu s$
Sharpness Calculation: Sobel	25ms	63ms	103ms
Sharpness Calculation: Laplacian	28ms	73ms	118ms
Sharpness Calculation: FFT	61ms	157ms	262ms

Table 5.3: Single-thread capture processing minimum runtimes

LiDAR runs at 20Hz), but the LiDAR registration thread is not doing any operations during that time. While calculating of sharpness values roughly rises with the amount of pixels present, copying from the queue and moving captures seem to be less consistent. This may of course be due to the extremely low runtimes leading to more deviations, or could stem from the Azure Kinect's SDK's not creating images of size 4096x3072 as efficiently. But at least the moving of captures can be explained by the different fps operating modes between the resolutions. When using the 4096x3072 resolution, the Azure Kinect can only record at a maximum of 15 fps. With less data objects to copy from one array to another, moving captures must be faster. Following this logic, in order to compare the runtimes of the 4096x3072 resolution to the others, one would need to cut their runtimes in half. Doing that, the 3840x2160 is actually the most demanding resolution for processing captures. Keep in mind that halving the runtimes does not apply as soon as point clouds are extracted.

	2048x1536	3840x2160	4096x3072
Point Cloud Extraction	65ms	146ms	229ms
PnP Registration	30ms	49ms	84ms
Transform Cloud	5ms	15ms	22ms
Clean and Transform Cloud	28ms	88ms	137ms
Save Complete Point Cloud	348ms	1112ms	1439ms

Table 5.4: Single-thread point cloud processing minimum runtimes

Table 5.4 lists runtimes for the point cloud processing steps. It is less exciting, as the runtimes scale accordingly with the resolution. One small exception is PnP registration, which hints at either OpenCV's [Bra00] ORB feature detection, matching and/or PnP implementation being optimized for 16:9 resolution images. Extraction of point clouds can be multi-threaded, but the same restraints as with the FFT example given above apply. When using 2 threads for example, each of them already takes 90ms to extract a point cloud. One should keep in mind that extraction of the point cloud only runs on a specified interval. Reasonable extraction rates (i.e. around or even less than 1 second) will never need to run more than one thread. Simply transforming and additionally cleaning the point cloud beforehand differs depending if PnP registration or *point cloud* *classification* are activated. As both of them need point clouds indices with corresponding image pixels, invalid points are kept and removed only before transforming. This adds quite a bit of time onto the processing runtime. Saving of a complete point cloud is the only step that seems unreasonably high, and it might be lower when using a faster storage medium. We tested on a Samsung SSD 970 EVO Plus 2TB. Keep in mind that most point clouds will most likely be filtered beforehand, reducing the time needed to save the point clouds approximately by the point cloud's reduction rate. If this is still too slow, saving can be multi-threaded, with which at least our SSD could easily write out complete clouds in real-time. The mentioned multi-thread performance decreases apply.

5.4.2 Filter Runtimes

This section lists the runtimes for the various filter set-ups we tested in Section 5.3.

Point Cloud Classification Filter

Classification runtimes seen in Table 5.5 do not change as much when increasing the resolution, since the network is always handling the same number of partitions at an image size of 640x640. Of course, higher resolutions allow for more partitions, increasing the runtime again for higher accuracy. But their runtimes are also not completely equal when given the same partition size. The increased milliseconds stem from down-sampling the images to fit the model size, with bigger images taking quite a bit more time.

	2048x1536	3840x2160	4096x3072
Classification: 1 partition	35ms	66ms	75ms
Classification: 4 partitions	82ms	114ms	126ms
Classification: 12 partitions	234ms	269ms	282ms
Classification: 24 partitions		481ms	487ms
Classification: 35 partitions			681ms

Table 5.5: Point cloud classification minimum runtimes. Input images need to be sub-sampled or partitioned to chunks of 640x640.

Plane Segmentation Filter

Plane segmentation filter runtimes can be seen in Table 5.6 as well as 5.7. As one can see, reducing or increasing the amount of neighbors used for both normal estimation and growing of the plane patches does not change the runtime of the filter at a resolution of 2048x1536 too much. For higher resolutions, the differences can be dismissed completely. On the other hand, going from a complete point cloud to a 1cm voxel grid already drastically increases the performance. But as the results of 5cm and 10cm show, at some point even doubling the grid size still leaves a big chunk of calculation. Going higher is not advised, as at a grid size of 10cm the majority of point clouds already do not have any planes detected in them. Though 1735ms for 1cm still seems rather much for the filter.

Luckily, multi-threading this does not tax the runtimes nearly as much as multi-threading any other of the functions. 2 threads run at 1920ms each, 4 threads at 2100ms and 8 threads of plane segmentation filtering each finish in only 2367ms. As this is the only part of our code not written by us [AO19], we do not really have an explanation for this. We could not find any obvious clues in the provided source code. Our best guess is that is spends most of its time looking for points scattered across multiple classes, arrays and memory locations. No matter the reason, thanks to this multi-threading property, the *plane segmentation filter* is still real-time viable for all sensible extractions rates.

	2048x1536	3840x2160	4096x3072
Plane Segmentation: Full Size	11803ms	57689ms	74500ms
Plane Segmentation: 1cm Grid Size	1735ms	3416ms	4438ms
Plane Segmentation: 3cm Grid Size	963ms	2091ms	2765ms
Plane Segmentation: 5cm Grid Size	625ms	1846ms	2445ms
Plane Segmentation: 10cm Grid Size	447ms	1328ms	1719ms

Table 5.6: Plane Segmentation Filter minimum runtimes optimized by sub-sampling input point clouds to only feature one point per grid-size, significantly improving runtimes. All times given using 75 neighbors for normal estimation and plane patch growing.

	2048x1536	3840x2160	4096x3072
Plane Segmentation: 75 neighbors	1735ms	3416ms	4928ms
Plane Segmentation: 50 neighbors	1609ms	3231ms	4796ms
Plane Segmentation: 25 neighbors	1416ms	3152ms	4656ms
Plane Segmentation: 15 neighbors	1191ms	3004ms	4461ms

Table 5.7: Plane Segmentation Filter minimum runtimes optimized by number of neighbors used to calculate normals and grow plane patches. All times given are for clouds sub-sampled to a grid size of 1cm.

Overlap Filter

The *overlap filter* is also quite costly. However, it is unique in the regard that its cost stems not by complex calculations, but due to the sheer amount of objects it is acted on. Filtering two full point clouds by each other takes about 150ms for both of them. Additionally, each object being part of the same point cloud processing array, at least some milliseconds are also lost waiting on other threads to finish writing their changes to the array before we can copy or change the next element. While the amount of time needed to filter each point cloud organically drops with each additional cloud (due to the point size getting smaller after each), this waiting delay stays. It is also the reason why the *overlap filter* cannot utilize multi-threading as much as the other components of the pipeline. When using 15 inputs, i.e. 30 filtering operations for each added cloud, going from one thread to two at least drops their time from 613ms to a combined average of 432ms for a resolution of 2048x1536. The 3840x2160 resolution recording achieves 1381ms and 975ms respectively. But adding a third thread does almost nothing to further reduce this number. Going to four threads achieves the same as three threads. This can be seen for our TU hallway recording in Figure 5.14. The only thing left to optimize is therefore the amount of inputs. Of course, by limiting the amount of inputs, one might also leave overlaps in-tact. Realistically, the *overlap filter* number of inputs should therefore be adjusted to the extraction rate and the speed of the movement of the recording operator. As Figure 5.14 shows, the number of inputs might be set quite high, as in this test the runtimes flatten around 25 inputs. This is because the additional input point clouds do not feature overlapping regions, and checking those only adds 1-2 additional milliseconds to the runtime.

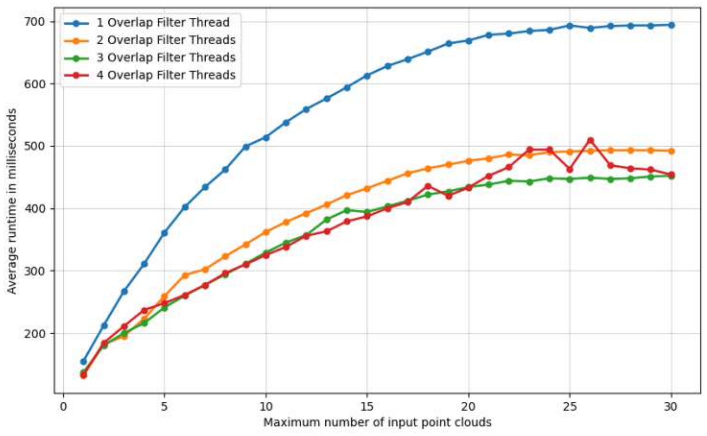


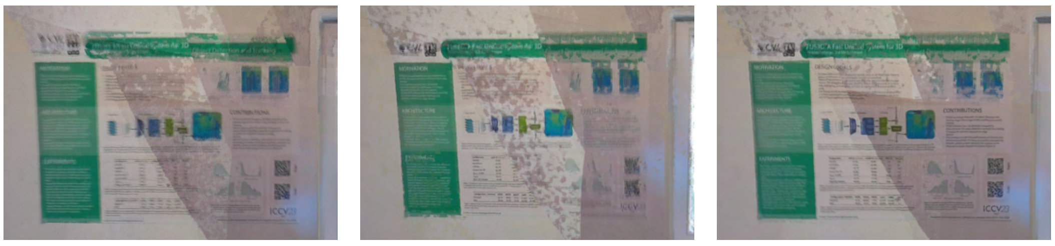
Figure 5.14: Overlap filter runtimes for a resolution of 2048x1536 compared by number of threads and number of input point clouds. Going from two to three threads already only decreases the average runtime by 20ms. Adding a fourth thread does not speed that up further, it even makes the graph more erratic. Runtimes do not scale linearly with additional input clouds either, as each additional overlap filtering of a cloud takes less time than the previous one. Runtimes flatten around 25 inputs, hinting at an extraction rate where more than 25 consecutive point clouds rarely overlap.

5.5 Discussion

In this section we will discuss the conclusions drawn from the evaluation we conducted. We will recommend what we think are the best parameters for recording and processing using the Azure Kinect and our *RGB-D data component*.

Choice of sharpness filters

Even though not visible from the evaluation, the biggest problem plaguing the Azure Kinect is the amount of motion blur in its images. This is why all of the evaluated results were created using the FFT sharpness filter. As seen in Figure 5.15, it is by far



(a) Sobel (b) Laplacian (c) FFT

Figure 5.15: Comparison of sharpness filters as seen on a textured poster. FFT should be preferred whenever possible.

the most accurate of the three we implemented, as the Sobel as well as the Laplacian filters still extracted blurry images from time to time. Of course, given an interval full of motion blur, even the FFT sharpness filter will extract a blurry image, but otherwise it performs as advertised. That comes with a harsh performance cost though. While it is no problem for 4 to 6 threads to handle sharpness calculation in real-time at our typical resolution of 2048x1536, for the most demanding resolution (3840x2160) one really needs a better CPU or GPU implementation to be real-time applicable. Of course, real-time performance for higher resolutions can still be achieved using e.g. the Laplacian filter. But if that higher resolution then also extracts blurry images, we do not think it is worth it. Rather, instead of going for the higher resolution, we think it would be best to just move the Azure Kinect closer to objects that need higher detail, also effectively doubling their density. Likewise, we would not recommend running the Azure Kinect at lower framerates in order to keep up with FFT, as that would lead to more blur in the first place.

Choice of point cloud extraction rate

As described above, there can still be intervals full of motion blur from which a point cloud will be extracted. One can combat this, either by moving the Azure Kinect slowly enough or by simply reducing the point cloud extraction rate. But when moving the device slower, one would want a lower extraction rate anyway, as the slower the movement, the longer it takes for the next batch of unseen data. This of course goes directly against the strengths of the PnP. Luckily, as the results of Section 5.2 show, the LiDAR pose estimation obtained from BIMCheck is already extremely accurate. The benefit gained from slight improvement using PnP is not worth the amount of processing time added by sending all those additional point clouds through our pipeline. Also, if one does not use the *overlap filter*, sending more point clouds through the pipeline also increases the amount of points the pipeline writes our or sends over network.

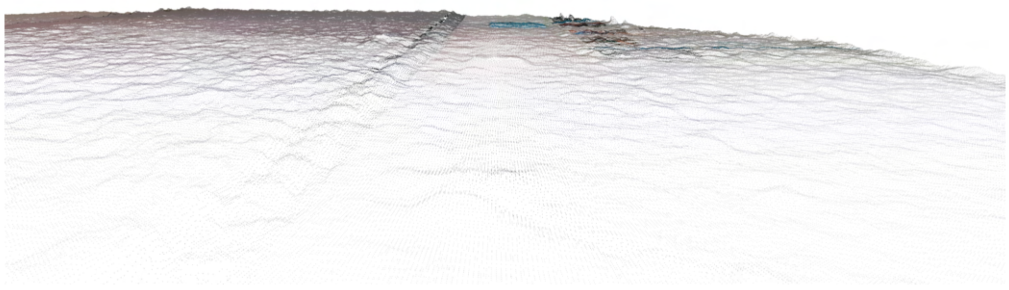


Figure 5.16: Depth errors in Azure Kinect captures on a flat wall. Filtering all the erroneous studs without filtering the light switch (center of the image) is asking too much of our *plane segmentation filter*.

Conclusions on point cloud filtering

When it comes to filtering, the *point cloud classification filter* is too unreliable for general use right now. If artificial intelligence trends continue, an open vocabulary network like YOLO-World [CSG+24] might make the other filters proposed in this thesis redundant. Right now, without re-training the network to more closely fit our own environments, it fails to detect more subtle objects like the fire damper in our TU hallway environment. Pipes or other objects that span over multiple frames might also have areas filtered away, leading to an incomplete cloud. For easily detectable objects it performs by far the best, as seen by the 90% reduction rate in the office environment. On the other hand, our filter based on plane segmentation [AO19] can trim down the point clouds without losing any valuable information, but not as much as we would like them to. This may certainly be due to the inaccuracies of the Azure Kinect depth camera. As seen in Figure 5.16, a wall is not exactly flat, but rather filled with small studs. One can lower the criteria of a plane flatness to filter these studs as well, but then one quickly loses light switches or power sockets in the process as well, as those barely stand out more than those studs. Given a more accurate depth camera, we are certain the *plane segmentation filter* could double its reduction rate. The *overlap filter* of course has a different goal, which rather than reducing point clouds to their essentials tries to clean them up both for storage and presentation purposes. In our opinion, it achieves what it sets out to do, with the artifacts (rips inside the walls) only occurring through misalignment errors. But it does come at a performance cost. Even though the *plane segmentation filter* may take longer, CPU usage is higher when using the *overlap filter*. Multi-threading the plane segmentation should take priority over it. Perhaps the best filtering results could be achieved by combining the *plane segmentation* and *overlap filter*. However, combination of the two is not trivial considering their vastly different characteristics. We believe a combination of the two should rather be implemented as a separate filter combining their ideas, rather than changing each filter's operations to fit the other.

Conclusions on capture resolution

For a resolution at or below 2048x1536, we do not think an adequately equipped battery-powered computer should struggle keeping a real-time performance. We would advise to use 6 threads of FFT sharpness calculation, and depending on the point cloud extraction rate, the amount of plane segmentation threads needed to handle the data in a real-time manner. We do not think results can get much better using our current tools. If one wants the initial class description guesses, turning on the classification without the filter does not hurt either.

For a real-time setting, we would advise against using a resolution of 3840x2160, due to most systems not being able to handle the FFT sharpness filter. But if quality for an offline processing later is prioritized, one cannot do better than this resolution. Real-time visualization of what is recorded can be estimated by turning on 6 to 8 threads of the Laplacian sharpness filter. Obviously, since real-time processing is not a concern, we would turn the rest of the pipeline off. In the offline setting afterwards, one should find a balance between FFT threads and plane segmentation filtering threads depending on the point cloud extraction rate. It should be noted that while this does provide the highest quality of data, the widescreen FoV of the Azure Kinect is in fact significantly smaller, as seen in Figure 3.6a.

We cannot recommend the 4096x3072 resolution, due to it being locked to 15 fps. When recording using this resolution, great care must be put into holding the camera still enough at important areas. The rest will probably be lost to motion blur. We also found the registration quality from the LiDAR poses to be worse in this mode, though the reason is not clear to us.

CHAPTER 6

Usage

Usage of our software is easy if it is provided as a pre-compiled executable. If one wants to alter the software, installing of the required libraries and SDKs is necessary. For this, please refer to the `Readme.md` file in our git repository.

This section will go into the usage of a compiled executable. Section 6.1 will explain how to start or replay a recording. Section 6.2 will show how to configure and fine-tune the various parameters of our application. Finally, 6.3 will go over the keyboard commands available when using the debug visualization after processing.

6.1 Recording and Replaying Data

In order to start our application, one first needs to provide it with a mode. There are only two modes, `RECORD` and `PLAYBACK`. `RECORD` can be run without any additional arguments:

```
> ./BIMKinectAzure.exe RECORD
```

which would just start the Azure Kinect, connect to the LiDAR and present a live visualization without actually saving the recording. If one does not think it will be necessary to run the program on the recording again, everything can be processed and turned into saved point clouds with this command as well. But it is more likely one also wants to save the `.mkv` of the recorded data as well, if only to be on the safe side should the parameters be set incorrectly. In which case it should be run with:

```
> ./BIMKinectAzure.exe RECORD "path/to/output.mkv"
```

This will not only create the `output.mkv` at the specified location, but also one .txt file containing all received LiDAR poses, and 2 additional .txt files containing timestamps of the LiDAR poses and the Azure Kinect captures. These are needed for playback of the recording. **ATTENTION: In order for these files to be created, the program**

must always be quit by pressing Q inside the terminal interface. Although less important, the program will also write out the last received LiDAR point cloud. This can also be used for comparison with the extracted point clouds. Once recorded, a file can be replayed using the PLAYBACK mode like this:

```
> ./BIMKinectAzure.exe PLAYBACK "path/to/input.mkv"
```

The program will look for the .txt files itself, but they can also be provided as optional arguments like this:

```
> ./BIMKinectAzure.exe PLAYBACK "path/to/input.mkv"
"path/to/kinect_capture_times.txt" "path/to/lidar_poses.txt"
"path/to/lidar_capture_times.txt" "path/to/lidar_cloud.ply"
```

But easiest is to just provide the path to a folder containing exactly one .mkv (and corresponding .txt files). For example:

```
> ./BIMKinectAzure.exe PLAYBACK "path/to/"
```

will do the exact same thing as providing the `input.mkv` to the end of the command. The path can be stripped down to `"path/to/input"`, without the `".mkv"`

As a last parameter, one may set the configuration file to be used. If none if provided, the program will look for a file named `config.ini` inside the folder where the .exe is located. Specification of a different configuration is possible by adding it to any of the previous commands:

```
> ./BIMKinectAzure.exe RECORD "path/to/output.mkv" "record.ini"
```

```
> ./BIMKinectAzure.exe PLAYBACK "input.mkv" "slow_config.ini"
```

```
> ./BIMKinectAzure.exe RECORD "test_config.ini"
```

```
> ./BIMKinectAzure.exe PLAYBACK "path/to/input.mkv"
"path/to/kinect_capture_times.txt" "path/to/lidar_poses.txt"
"path/to/lidar_capture_times.txt" "path/to/lidar_cloud.ply"
"path/to/superfast_process.ini"
```

How the .ini can be adjusted to one's liking will be discussed in the next section.

6.2 Configuration

Just like our differing set-ups for Chapter 5, one may want to have multiple configurations. It is important to note that whatever the configuration, the program will always record the full data stream to a `.mkv` if it is provided with an output file path. But one should be mindful of memory usage - if memory freeing is disabled and the program records for several minutes, the hard drive space might get taken over by RAM, potentially using up all space in which case the recording can no longer be saved. Therefore, the only firm suggestion we can give is to leave the freeing of memory **on** while recording. All others can be freely played around with and should not break any integral part of the program (but of course, might lead to empty point clouds).

The `.ini` provided is split into sections that roughly represent our pipeline's steps. In order for a custom configuration to work, a value should be provided to each parameter. If in doubt, insert the default parameters from these sections.

The following sections will feature the associated .ini-Header in brackets as `[Header]` and all parameters associated with it underneath as `parameter_name=default_value`.

6.2.1 Record Config

`[Record]`

- `lidar_port=127.0.0.1:5021`

  The IP of the LiDAR the program will try to connect to.

- `color_resolution_mode=4`

  Color resolution used by the Azure Kinect. Will be read from the `.mkv` if not recording. Mode numbers represent `k4a_color_resolution_t` enum from the Azure Kinect SDK: 1 = 720p, 2 = 1080p, 3 = 1440p, 4 = 1536p (4:3), 5 = 2160p, 6 = 3072 (4:3). Mode 6 only allows recording at 15 fps.

- `depth_mode=2`

  Depth mode used by the Azure Kinect. Will be read from the `.mkv` if not recording. Mode numbers represent the `k4a_depth_mode_t` enum from the Azure Kinect SDK: 1=NFOV_2X2BINNED, 2=NFOV_UNBINNED, 3=WFOV_2X2BINNED, 4=WFOV_UNBINNED, 5=PASSIVE_IR, 0=OFF. 0 will break the program.

- `fps_mode=2`

  fps mode used by the Azure Kinect. Will be read from the `.mkv` if not recording. Mode numbers represent the `k4a_fps_t` enum from the Azure Kinect SDK: 0 = 5 fps, 1 = 15 fps, 2 = 30 fps. If color_resolution_mode is set to 3072p and this is set to 30 fps, the program will override it to 15 fps.

- `frames_to_record=0`

  If set to anything less than 1, will record indefinitely until Q is pressed. Can also be used during playback to only extract, process and later debug the first x frames of the recording, potentially saving a lot of time.

- `lidar_point_cloud_request_rate_in_s=5.0`

  Rate at which the LiDAR's point cloud is requested and updated inside the Azure Kinect component.

- `debug=false`

  Outputs warnings regarding the copying of Azure Kinect data. SDK errors will still be printed even if set to false.

6.2.2 Playback Config

`[Playback]`

- `emulate_recording_delay=false`

  If playback mode is activated, makes the program wait before reading each next capture as if it were coming from a live recording device. Can be used to test or demo real-time applicability of other parameters.

6.2.3 Sharpness Filter Config

`[SharpnessFilter]`

- `enable_sharpness_filter=true`

  Enables the *sharpness filter*, vastly improving the output's point cloud's RGB quality.

- `sharpness_filter_name=fft`

  The *sharpness filter* has three modes: Sobel, Laplacian and FFT. Sobel is the fastest, while FFT is the most accurate. As a rule of thumb, number of threads should be roughly doubled if using FFT.

- `nr_of_threads=6`

  The sharpness filter took between 25 and 60 milliseconds to process a frame on our set-up, depending on the resolution and sharpness filter used. Since it is run on every frame, number of threads needs to be quite high in order to match real-time applicability.

- `debug=false`

  Outputs runtimes, warning and error messages for the *sharpness filter*.

6.2.4 Point Cloud Extraction Config

`[PointCloudExtraction]`

- `nr_of_threads=2`

  Specifies the number of concurrent threads that should extract point clouds from image frames. Extraction takes between 65 and 229 milliseconds on our system, depending on the resolution. But since the extraction rate is likely below that, most of the times even one thread should more than suffice.

- `extraction_rate_per_second=1.00`

  Sets the number of point clouds to be extracted from frames each second. For example, at a rate of 3.0 and an fps of 30 will extract Frame 1, Frame 11 and Frame 21 from the recording. If *sharpness filter* is enabled, will instead extract the sharpest frame between 1-10, 2-20 and 3-30 respectively.

- `max_depth_in_meters=5.0`

  Filters all points further away from the Azure Kinect than this distance. 5.0 represents more or less the maximum the Azure Kinect's depth camera can capture accurately. Is also used by the *overlap filter* to detect overlapping view frustums.

- `min_depth_in_meters=0.5`

  Filters all points nearer to the Azure Kinect than this distance. 0.5 represents more or less the minimum at which the Azure Kinect's depth camera can capture accurately. Is also used by the *overlap filter* to detect overlapping view frustums.

- `debug=false`

  Outputs runtimes, warning and error messages for the extraction of point clouds.

6.2.5 Registration Config

```
[Registration]
```

- `enable_kinect_fusion=false`

  Enables KinFu registration. Takes one thread with minimal CPU cost.

- `enable_lidar=true`

  Enables LiDAR registration. Takes one thread with little to no CPU cost. If run live, is essential to capture LiDAR poses and timestamps.

- `enable_pnp=true`

  Enables PnP registration. Takes one thread with minimal CPU cost. Will lock point clouds for 1 step longer than normally necessary, as input of previous cloud is needed unfiltered. Will also increases the memory footprint of point clouds by 50% before they are transformed. They will be shrunk to their normal size afterwards.

- `pnp_use_initial_guess=true`

  If enabled, uses pose estimation of either KinFu or LiDAR as initial guess for PnP. Vastly improves results at no additional cost.

- `pnp_iterations=100`

  Sets the number of iterations for PnP. For more information, see the OpenCV documentation [Bra00].

- `pnp_reprojection_error=8.0`

  Sets the PnP reprojection error. For more information, see the OpenCV documentation [Bra00].

- `pnp_confidence=0.99`

  Sets the PnP confidence. For more information, see the OpenCV documentation [Bra00].

- `debug=false`

  Outputs runtimes, warning and error messages for KinFu, LiDAR and PnP Registration.

6.2.6 Point Cloud Classification Config

`[ObjectDetection]`

- `enable_object_detection=false`

  Enables *point cloud classification* using YOLO-World, which will be saved to the `.ply`'s output. Takes one thread. Needs an NVIDIA GPU with Cuda 11.8 installed on the system to run in real-time. But depending on point cloud extraction rate, child parameters and GPU, might still struggle. Will increase the memory footprint of point clouds by 50% before they are transformed, but will be shrunk to their normal size afterwards.

- `enable_object_detection_filter=false`

  Enables the *classification filter*. Will remove all points from point clouds without a detected class.

- `model_filepath=yolow-l.onnx`

  Path to where the downloaded or trained YOLO-World model is located relative to the `.exe`'s location (not the `config.ini`!)

- `class_names=Outlet, Light Switch, Radiator, Pipes, Fire Extinguisher, Fire Damper`

  Class names / prompt descriptors to be detected. **Must be** the same as were input during model's build (although not training)!

- `maximum_partitions_per_frame=0`

  At 0, splits image into as many parts of 640x640 as possible. Otherwise splits the image into the number of partitions. Then sends each separately into the network.

- `detection_score_threshold=0.05`

  Threshold, under which all object classifications will be thrown away.

- `allow_multiple_classes_per_pixel=true`

  Allows multiple classes to be added to each point cloud's point. Number of classes must be less than 32.

- `debug=false`

  Debugs the *classification filter* visually by showing output of image frames fed into it.

6.2.7 Plane Segmentation Config

`[PlaneSegmentation]`

- `enable_plane_segmentation=false`

  Enables the *plane segmentation filter*. Will take a tremendous amount of time and CPU power without sub-sampling.

- `nr_of_threads=4`

  Number of point clouds being plane segmented at the same time. As segmentation of one point cloud can take anywhere from 2 to 4 seconds even with sub-sampling, should be adjusted to fit the point cloud extraction rate and resolution.

- `subsample_voxel_grid_size=0.01`

  Grid size to which the point cloud shall be sub-sampled to in meters.

- `normal_estimation_radius=0.0`

  All neighboring points inside a point's radius that will be used for normal estimation. If set to 0, will use the next parameter instead.

- `normal_estimation_nr_of_neighbors=75`

  Number of a point's nearest neighbors that will be used for normal estimation.

- `patch_to_point_min_normal_diff_in_deg=25.0`

  Maximum angle in degrees between a plane's normal and a point's normal, for the point to be considered on the plane.

- `patch_to_point_max_dist_in_deg=30.0`

  Maximum angle in degrees of the angle formed by the 3D point, the plane's center and the projected 3D point on the plane. Points belonging to the planar surface will have a smaller angle.

- `patch_outlier_ratio=0.1`

  Number of outliers permitted to count a set of points as a planar patch.

- `patch_point_nr_of_neighbors=75`

  Number of potential neighbors with which each point will grow their planar patch.

- `keep_points_with_max_distance_to_non_plane_point_in_m=0.002`

  After *plane segmentation filter*, keeps all points around a point not on a plane around this radius.

6.2.8 Overlap Filter Config

`[OverlapFilter]`

- `enable_overlap_filter=true`

  Enables the *overlap filter*.

- `nr_of_inputs=15`

  Number of previous point clouds used as input for each new point cloud. Will keep point clouds from being released from memory until at least nr_of_inputs have filtered it. So after n point clouds, each point cloud will be filtered by n*2 view frustums.

- `filter_by_all_previous_poses=false`

  Filters point clouds by all previous view frustums.

- `allow_overlap_on_older_previous_poses=false`

  If filtering by all previous poses points seen in previous poses will get filtered out if the poses' respective point cloud is not in memory anymore. Enabling this keeps points which are closer to the new pose's camera position, leading to overlaps.

- `rotation_threshold=90.0`

  Rotational threshold for 2 view frustums to filter each other. Should be around Azure Kinect's FoV.

6.2.9 Point Cloud Output Config

`[Output]`

- `save_point_clouds=true`

  Enables saving of point clouds to `.ply` files.

- `nr_of_threads=4`

  Number of threads concurrently saving. Saving might not be computationally expensive, but can still take a long time, so for an extraction rate of 1.0 per second, without any point cloud reduction, 4 threads would be recommended. When using the *overlap filter* and especially when using the *classification filter*, point clouds may also be small enough s.t. one thread suffices. Write speed may also be affected by storage hardware used.

- `output_folder=..\..\..\results`

  Specifies the output folder relative to the `.exe`'s location.

- `output_filename_suffix=_`

  Will be printed between `record_name` and `[point_cloud_index].ply`. For example, if set to "_OL_filtered_" will output the first point cloud as `input_OL_filtered_0.ply`.

6.2.10 Memory Freeing Config

```
[Memory]
```

- `enable_memory_freeing=true`

  Should always be set to true, except if one wants to debug all point clouds after processing in the debug visualizer.

- `max_number_of_point_clouds_in_pipeline=25`

  Sets the maximum number of point cloud's inside the pipeline at any point. Number should match the amount of point clouds concurrently being worked on by all threads. Can be easily calculated by: 2 if PnP is activated + 1 if *point cloud classification* is activated, + 1 for *transformation*, + x for the number of *plane detection* threads, + y for the number of input point clouds in the *overlap filter*. A little bit more can serve as a buffer, since extraction of captures from a recording is stopped if there are already enough point clouds inside the pipeline.

6.2.11 Visualization Config

```
[Visualization]
```

- `live_visualization_window=transformation`

  Active window for real-time visualization. Options include: *transformation*, *object_detection_filter plane_segmentation_filter*, and *overlap_filter*. Putting anything else into it disables it.

- `live_max_number_of_point_clouds=20`

  Specifies the number of point clouds seen in the real-time visualization. If activated at all, uses the same CPU power no matter if 1 or 100 point clouds are concurrently visualized. But memory usage will be affected by this.

6.3 Debug Visualization

This is an additional guide for the debug visualization that opens up after the program finished processing **if** `enable_memory_freeing` is set to *false*. The interaction with the visualization window that pops up during recording should be self-explanatory: Clicking and dragging with the left mouse button rotates the camera, scrolling zooms in and out, and the middle mouse button moves the center of the viewing camera around the scene. Like any other 3D viewing software, zooming will only take one so far, so instead of scrolling to a point of interest, one should first move the camera position towards the desired target using the middle mouse button, only then zoom by scrolling.

When the debug visualization window first shows up, it will most likely show the combined point cloud. Depending on how many point clouds are in memory, it may take quite a while until the camera can be moved. After which the following keyboard commands are available to the user:

- C: Switch from combined view to single view and vice versa
- →: In single view: Move 1 point cloud forwards in the array
- ←: In single view: Move 1 point cloud backwads in the array
- M: In single view: Mark (toggle) inclusion for combined point cloud
- A: In single view: Mark all point Clouds for combined point cloud
- U: In single view: Unmark all point clouds for combined point cloud
- S: Save currently displayed point cloud to single `.ply` file
- W: Save **all** marked point clouds as separate `.ply` files
- P: Compute and show plane segmentation for currently displayed point cloud
- I: Show information for current point cloud. Most likely empty. To fill it, one may add debug strings to a `colored_point_cloud`'s `debug_info` attribute anywhere in the code before visualization.

If one alters and re-compiles the code, one may also include `Visualization.hpp` and start a debug visualization anywhere in the software using either `visualize_single_point_cloud` or `visualize_multiple_point_clouds`. There are function overloads both for using our `colored_point_cloud` struct, as well as PCL's own `pcl::PointCloud<pcl::PointXYZRGBL>::Ptr`.

CHAPTER 7

Conclusion and Future Work

In this thesis, we designed and developed a real-time system for capturing dense and colored point cloud using an Azure Kinect RGB-D camera. We implemented the system as part of the BIMCheck project, and use the pose estimation of its LiDAR SLAM implementation to align our point clouds.

In order to improve on the LiDAR pose estimations, KinFu and PnP were introduced to the system. The system further utilizes many pre- and post-processing methods to improve on the results. A pre-processing sharpness filter reduces the amount of blurred data, while three distinct filters may reduce the point cloud data before sending the dense point cloud data to other components of the project. Two of these filters try to minimize the amount of redundant data, as only details of the geometry that are too small for the LiDAR to resolve are kept. Additionally, a real-time open vocabulary neural network was implemented, which can be used to detect objects in the point clouds based on text prompts, without the need for any model training.

We evaluated our system in a hallway full of small objects as well as inside of a larger office complex. Compared to a state-of-the-art LiDAR point cloud, our registration accuracy only shows a mean error distance of 3.69cm. Our implemented overlap filter can avoid capturing the same area twice, even if the old point cloud reconstructions are no longer in memory. Our two other filters can further reduce the point cloud data to only represent small objects of interest, such as power plugs, light switches etc., reaching reduction rates of 80-90%.

Just like the project it is a part of, our proposed *Azure Kinect data pipeline* is a modular system as well, and a real-time set-up can be configured for any machine running it. Though for the highest of quality settings, we recommend running this on recent desktop PC or state-of-the-art laptop.

There are still a number of ways how the proposed system can be further improved. In order to make the higher quality setting more real-time applicable on less performant hardware configurations, we propose running the FFT sharpness filter on GPU instead of CPU. This will be especially important should a system like this ever be used with a more detailed depth camera. A registration refinement using PnP did not show any significant improvements. However, the literature suggests that a more profound implementation using PnP strategies can make the system match state-of-the-art LiDAR systems. Such an improvement could then be used by our own LiDAR pose estimation to help in the detection of loop closure.

Even though we did not detect any errors when scaling our room size from a hallway to a larger office complex, the system does not react to a loop closure detected by the BIMCheck component. Since such a component might alter the positions of previous point clouds no longer in memory of our *Azure Kinect data pipeline*, we recommend adjusting for the loop closure inside the system responsible for showing this component's results.

We believe our system to be fit for the task of recording and reconstructing buildings for the comparison of smaller installations with a building's BIM twin. Its runtime performance and implemented filters can allow the BIMCheck project to present a detailed visualization of the differences to the user in real-time.

List of Figures

List of Tables

List of Algorithms

Acronyms

AMCW Amplitude Modulated Continuous Wave. 10

API Application Programming Interface. 45

BIM Building Information Modeling. xv, 1–3, 6, 9, 18, 19, 22, 24, 26, 62–68, 94, 95

CPU Central Processing Unit. 14, 33, 37, 64, 74, 79, 80, 87, 89, 91, 94

CSPC Cross-Source Point Cloud. 15, 17, 18

DoF Degrees of Freedom. 22

FFT Fast Fourier Transform. 33, 49, 74, 75, 78, 79, 81, 86, 94

FoV Field of View. 34, 43, 81, 90, 95

fps frames per second. 5, 51, 62, 75, 81, 85

GAN Generative Adversarial Network. 17

GMM Gaussian Mixture Model. 17

GPU Graphics Processing Unit. 14, 37, 52, 64, 74, 79, 88, 94

Hz Hertz. 5, 51, 75

ICP Iterative Closest Point. 12–14, 16–19, 65

IMU Inertial Measurement Unit. 3, 5

KinFu Kinect Fusion. 20, 27–29, 31, 32, 36, 46, 50–52, 63, 68, 74, 75, 87, 88, 93, 95

LiDAR Light Detection and Ranging. 3–6, 15–20, 22–24, 26–32, 36, 37, 50–52, 63, 65, 66, 68, 72, 74, 75, 79, 81, 83–85, 87, 88, 93–95

LUT Look-Up Table. 51

ONNX Open Neural Network Exchange. 46, 52, 53

PCA Principal Component Analysis. 17

PCL Point Cloud Library. 13, 39, 46, 50, 52, 54, 92

PnP Perspective-n-Point. 14, 20, 29, 30, 36, 37, 46, 48, 50–52, 63, 68, 75, 79, 87, 88, 91, 93–95

RAM Random Access Memory. 25, 31, 41, 43, 47, 48, 64, 74, 84

RANSAC Random Sample Consensus. 14, 19

SDK Software Development Kit. 45, 46, 49, 50, 75, 83, 85, 86

SfM Structure from Motion. 15, 16, 95

SLAM Simultaneous Localization and Mapping. 3–5, 14, 24, 32, 63, 66, 93

Slerp Spherical Linear Interpolation. 51

SURF Speeded Up Robust Features. 14

ToF Time-of-Flight. 10, 11

TSDF Truncated Signed Distance Function. 12, 13, 15, 56

WiFi Wireless Fidelity. 26

YOLO You Only Look Once. 29, 35, 46, 52, 53, 69, 70, 80, 88, 95

Bibliography

[ALP20] Ahmed Khairadeen Ali, One Jae Lee, and Chansik Park. Near real-time monitoring of construction progress: Integration of extended reality and kinect v2. In *Proceedings of the 37th International Symposium on Automation and Robotics in Construction (ISARC)*, pages 24–31, Kitakyushu, Japan, October 2020. International Association for Automation and Robotics in Construction (IAARC).

[AO19] Abner Araujo and Manuel Oliveira. A robust statistics approach for plane detection in unorganized point clouds. *Pattern Recognition*, 100:107115, 11 2019.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, sep 1975.

[BIM24] United BIM. Electrical bim services. `https://www.united-bim.com/electrical-bim-services/`, 2024. Accessed: February 15, 2024.

[BM92] P.J. Besl and Neil D. McKay. A method for registration of 3-d shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, 1992.

[Bra00] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[BTVG06] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In Aleš Leonardis, Horst Bischof, and Axel Pinz, editors, *Computer Vision – ECCV 2006*, pages 404–417, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[CL96] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, page 303–312, New York, NY, USA, 1996. Association for Computing Machinery.

[Com23] D. R. Commander. libjpeg-turbo. https://libjpeg-turbo.org, 2023.

[CSG+24] Tianheng Cheng, Lin Song, Yixiao Ge, Wenyu Liu, Xinggang Wang, and Ying Shan. Yolo-world: Real-time open-vocabulary object detection, 2024.

[CSK05] Dmitry Chetverikov, Dmitry Stepanov, and Pavel Krsek. Robust euclidean alignment of 3d point sets: the trimmed iterative closest point algorithm. *Image and Vision Computing*, 23(3):299–309, 2005.

[dev21] ONNX Runtime developers. Onnx runtime. `https://onnxruntime.ai/`, 2021. Version: 1.17.1.

[DGGB+23] Luca Di Giammarino, Emanuele Giacomini, Leonardo Brizi, Omar Salem, and Giorgio Grisetti. Photometric lidar and rgb-d bundle adjustment. *IEEE Robotics and Automation Letters*, 8(7):4362–4369, 2023.

[DNZ+16] Angela Dai, Matthias Nießner, Michael Zollhöfer, Shahram Izadi, and Christian Theobalt. Bundlefusion: Real-time globally consistent 3d reconstruction using on-the-fly surface re-integration. *CoRR*, abs/1604.01093, 2016.

[EHMR+22] N. El Haouss, R. Makhloufi, I. Rached, R. Hajji, and T. Landes. Use of kinect azure for bim reconstruction: Establishment of an acquisition protocol, segmentation and 3d modeling. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLVIII-2/W1-2022:87–94, 2022.

[FB81] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, jun 1981.

[GJ+10] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[GM24] Daniel Girardeau-Montaut. Cloudcompare (version 2.12.1) [gpl software]. http://www.cloudcompare.org/, 2024.

[HFW+19] Xiaoshui Huang, Lixin Fan, Qiang Wu, Jian Zhang, and Chun Yuan. Fast registration for cross-source point clouds by using weak regional affinity and pixel-wise refinement. *CoRR*, abs/1903.04630, 2019.

[HLZ+22] Xiaoshui Huang, Sheng Li, Yifan Zuo, Yuming Fang, Jian Zhang, and Xiaowei Zhao. Unsupervised point cloud registration by learning unified gaussian mixture models. *IEEE Robotics and Automation Letters*, 7(3):7028–7035, 2022.

[HMZ23] Xiaoshui Huang, Guofeng Mei, and Jian Zhang. Cross-source point cloud registration: Challenges, progress and prospects, 2023.

[HMZA21] Xiaoshui Huang, Guofeng Mei, Jian Zhang, and Rana Abbas. A comprehensive survey on point cloud registration, 2021.

[HWF+22] Fan Huang, Niannian Wang, Hongyuan Fang, Hai Liu, and Gaozhao Pang. Research on 3d defect information management of drainage pipeline based on bim. *Buildings*, 12(2), 2022.

[HZF+17] Xiaoshui Huang, Jian Zhang, Lixin Fan, Qiang Wu, and Chun Yuan. A systematic approach for cross-source point cloud registration by preserving macro and micro structures. *IEEE Transactions on Image Processing*, 26(7):3261–3276, July 2017.

[HZW+16] Xiaoshui Huang, Jian Zhang, Qiang Wu, Lixin Fan, and Chun Yuan. A coarse-to-fine algorithm for registration in 3d street-view cross-source point clouds. In *2016 International Conference on Digital Image Computing: Techniques and Applications (DICTA)*, pages 1–6, 2016.

[HZW+18] Xiaoshui Huang, Jian Zhang, Qiang Wu, Lixin Fan, and Chun Yuan. A coarse-to-fine algorithm for matching and registration in 3d cross-source point clouds. *IEEE Transactions on Circuits and Systems for Video Technology*, 28(10):2965–2977, 2018.

[Kud22] Kudan Inc. Kudan 3D-LiDAR SLAM (KdLidar) in Action: Vehicle-Based Mapping in an Urban Area. `https://www.kudan.io/blog/kdlidar-in-action-vehicle-based-mapping-in-an-urban-area/`, 2022. Last accessed: 2022-11-03.

[KVB88] Nick Kanopoulos, Nagesh Vasanthavada, and Robert L Baker. Design of an image edge detection filter using the sobel operator. *IEEE Journal of solid-state circuits*, 23(2):358–367, 1988.

[LGC+18] M. Lamine Tazir, Tawsif Gokhool, Paul Checchin, Laurent Malaterre, and Laurent Trassoudaine. Cicp: Cluster iterative closest point for sparse–dense point cloud registration. *Robotics and Autonomous Systems*, 108:66–86, 2018.

[LJQ20] Nan Luo, Yuanyuan Jiang, and Wang Quan. Supervoxel based region growing segmentation for point cloud data. *International Journal of Pattern Recognition and Artificial Intelligence*, 35, 10 2020.

[LLW+21] W. Liu, B. Lai, C. Wang, X. Bian, C. Wen, M. Cheng, Y. Zang, Y. Xia, and J. Li. Matching 2d image patches and 3d point cloud volumes by learning local cross-domain feature descriptors. In *2021 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW)*, pages 516–517, Los Alamitos, CA, USA, apr 2021. IEEE Computer Society.

[LMNF09] Vincent Lepetit, Francesc Moreno-Noguer, and Pascal Fua. Epnp: An accurate o(n) solution to the pnp problem. *International Journal of Computer Vision*, 81, 02 2009.

[LQ22] Xiao Ling and Rongjun Qin. A graph-matching approach for cross-view registration of over-view 2 and street-view based point clouds, 2022.

[MAT17] Raul Mur-Artal and Juan D. Tardos. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, October 2017.

[MCC+20] Niall O' Mahony, Sean Campbell, Anderson Carvalho, Suman Harapanahalli, Gustavo Velasco-Hernandez, Lenka Krpalkova, Daniel Riordan, and Joseph Walsh. *Advances in Computer Vision: Proceedings of the 2019 Computer Vision Conference (CVC), Volume 1*. Springer International Publishing, 2020.

[Mea82] Donald Meagher. Geometric modeling using octree-encoding. *Computer Graphics and Image Processing*, 19:129–147, 06 1982.

[Mic22] Microsoft Inc. Use Azure Kinect Sensor SDK image transformations. `https://learn.microsoft.com/en-us/azure/kinect-dk/use-image-transformation`, 2022. Last accessed: 2024-16-02.

[NIH+11] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pages 127–136, 2011.

[PHY22] Zihao Pan, Junyi Hou, and Lei Yu. Optimization algorithm for high precision rgb-d dense point cloud 3d reconstruction in indoor unbounded extension area. *Measurement Science and Technology*, 33, 01 2022.

[PLLZ23] Lei Pang, Dayuan Liu, Conghua Li, and Fengli Zhang. Automatic registration of homogeneous and cross-source tomosar point clouds in urban areas. *Sensors*, 23(2), 2023.

[PWF+14] Furong Peng, Qiang Wu, Lixin Fan, Jian Zhang, Yu You, Jianfeng Lu, and Jing-Yu Yang. Street view cross-sourced point cloud matching and registration. In *2014 IEEE International Conference on Image Processing (ICIP)*, pages 2026–2030, 2014.

[RC11] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[Rob] Robert McNeel & Associates. Rhinoceros 3D.

[RRKB11] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *2011 International Conference on Computer Vision*, pages 2564–2571, 2011.

[Rus09] Radu Bogdan Rusu. *Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments*. PhD thesis, Computer Science department, Technische Universitaet Muenchen, Germany, October 2009.

[Sch22] Linus Schaub. Point cloud registration in bim models for robot localization. Master's thesis, Technische Universität Wien, 2022.

[SGS$^+$18] Tyson Swetnam, Jeffrey Gillan, Temuulen Sankey, Mitchel McClaran, Mary Nichols, Philip Heilman, and Jason McVay. Considerations for achieving cross-platform point cloud data fusion across different dryland ecosystem structural states. *Frontiers in Plant Science*, 8, 01 2018.

[SSP07] Robert Sumner, Johannes Schmid, and Mark Pauly. Embedded deformation for shape manipulation. *ACM Transactions on Graphics*, 26, 07 2007.

[TMHF00] Bill Triggs, Philip F. McLauchlan, Richard I. Hartley, and Andrew W. Fitzgibbon. Bundle adjustment — a modern synthesis. In Bill Triggs, Andrew Zisserman, and Richard Szeliski, editors, *Vision Algorithms: Theory and Practice*, pages 298–372, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[WKJ$^+$15] Thomas Whelan, Michael Kaess, Hordur Johannsson, Maurice Fallon, John J. Leonard, and John McDonald. Real-time large-scale dense rgb-d slam with volumetric fusion. *The International Journal of Robotics Research*, 34(4-5):598–626, 2015.

[WMK$^+$12] Thomas Whelan, John McDonald, Michael Kaess, Maurice Fallon, Hordur Johannsson, and John J. Leonard. Kintinuous: Spatially extended kinectfusion. In *Proceedings of RSS '12 Workshop on RGB-D: Advanced Reasoning with Depth Cameras*, July 2012.

[YEK$^+$20] Wentao Yuan, Ben Eckart, Kihwan Kim, Varun Jampani, Dieter Fox, and Jan Kautz. Deepgmr: Learning latent gaussian mixture models for registration, 2020.

[ZSK17] Yasin Zamani, Hamed Shirzad, and Shohreh Kasaei. Similarity measures for intersection of camera view frustums. In *2017 10th Iranian Conference on Machine Vision and Image Processing (MVIP)*, pages 171–175, 2017.

[ZY19] Guangtang ZHU and Minl YE. Research on the method of point cloud denoising based on curvature characteristics and quantitative evaluation. *Bulletin of Surveying and Mapping*, 0(6):105, 2019.