The History of the Walls

Update of indoor IFC models using point cloud geometry

van der Meer, Jean

Award date:
2024

The History of the Walls

Update of indoor IFC models using point cloud geometry

Master Thesis

Student: Jean van der Meer
1760602
+31 613020531
j.v.d.meer1@student.tue.nl
jeanvdmeer@gmail.com

Graduation committee:
1st supervisor: Dr. Ir. Pieter Pauwels (BE)
2nd supervisor: Dr. Ekaterina Petrova (BE)
3rd supervisor: Dr. Elena Torta (ME)
Company supervisor: Ir. Mohammad Kafaei – BIM-Connected
Chairman: Dr. Ir. Pieter Pauwels (BE)

Course: 7CC40 - Graduation project Construction Management and Engineering
Study Load: 40 ECTS
Academic Year of graduation: 2024-2025

Eindhoven University of Technology
Master Construction management and Engineering (CME)

Date: November 25, 2024

TU/e

This page is intentionally left blank

Colophon

Full title	**The History of the Walls** - Update of indoor IFC models using point cloud geometry
Date of presentation	**25-11-2024**
Academic Year	**2024-2025**
Program	**Construction Management and Engineering (CME)** *Eindhoven University of Technology (TU/e)* *Groene Loper 6, 5612AZ Eindhoven, The Netherlands*
Course	7CC40 **- Graduation project Construction Management and Engineering** – 40 ECTS
Chairman of committee	**Dr. Ir. Pieter Pauwels**
Graduation committee	**Dr. Ir. Pieter Pauwels** – 1st supervisor *Associate Professor at TU/e's departments Built Environment (BE) and Eindhoven Artificial Intelligence Systems Institute (EAISI)* **Dr. Ekaterina Petrova –** 2nd supervisor *Assistant Professor at TU/e's departments Built Environment (BE) and Eindhoven Artificial Intelligence Systems Institute (EAISI)* **Dr. Elena Torta** – 3rd supervisor *Assistant Professor at TU/e's departments Mechanical Engineering (ME) and Eindhoven Artificial Intelligence Systems Institute (EAISI)*
Company advisor	**Ir. Mohammad Kafaei** *PDEng – consultant at BIM-Connected*
Cooperating company	**BIM-Connected** Torenallee 110, 5617 BE Eindhoven
Name	**Jean van der Meer**
Student number	**1760602**
Email	jeanvdmeer@gmail.com

This master thesis has been carried out in accordance with the rules of the TU/e Code of Scientific Integrity.

Preface

Dear reader,

This work marks the end of my formal academic education, even though learning shall always remain enjoyable, maybe all the more after finishing my formal studies. But it was a nice journey, and I have a large number of people to thank for all the valuable knowledge and moments in this long time studying. The history of my Master studies starts when I decided to learn some more, beyond my Bachelor studies in Civil Engineering. I was not yet living in The Netherlands at the time, but discovered Construction Management and Engineering, where I felt I had the chance to learn a lot more about how new technologies are being applied into the world of construction. CME is offered in three Dutch Universities, so when trying to know in which one I wanted to study, I tried to contact them. I was impressed by the kindness and readiness that Prof. Dr. Bauke de Vries showed when I sent an email – I only had a few questions for which I would have been happy by a written response, but busy as he was, chair of an important research group, he took the time to propose a videocall and answer my questions in good detail. So Prof. Bauke is the first person I have to thank.

The study program seemed really interesting, the courses available were what I wanted to study, with many nice free-electives, so I was sold – and embarked to start my CME journey. Upon starting the study program, I enrolled into the Fundamentals of BIM course, taught by Pieter Pauwels, thinking I would learn some advanced BIM modelling… To my surprise I learnt much more than that, with much valuable knowledge about data behind BIM, how it works, and programming applied to the Built Environment. Another course also taught by Pieter is Parametric Design, also tremendously enjoyable and fun, with a high density of knowledge taught. With most of my favorite courses being taught by Pieter, I thought the most reasonable decision would be to also have him as my supervisor. I am glad and thankful he accepted. Initially I wanted to work on a different topic, but he suggested another one – this one – which I thought was near impossible to do. I took my chances, and even though it took me a lot of effort and time, I think it was worth it, and I learned a lot working on something I felt is meaningful, and that allowed me to study many fun things. So thanks, Pieter!

At many crucial moments of my thesis I also had help from the company BIM-Connected, so I want to thank the BIM-Connected team for their support. Through BIM-Connected I got in contact with Mike Zitter and Oscar Miranda from the Erasmus MC, that helped me to understand how they use point clouds in the practice of their facility management activities, and were kind enough to provide me with IFC/ point cloud data and allow me to scan data. Thanks Mike and Oscar! I would also like to thank Simple BIM, specially Tomi, that supplied me with a student license to work with some IFC models, and Mark and Tiemen from the Qonic team, that were kind enough to show me and tell me more about how their online tool works, that updates IFC files online in real time, with collaboration in teams and fast geometric processing. I would furthermore like to thank Prof. André Borrmann, that took time to talk to me and give me valuable tips for my thesis at an early stage of my thesis. Special thanks also go to all the people that supported and had good moments with me during my education and my thesis, specially my friends from my study association (of CoUrsE!) and my sports association (Odin). You made my student years all the more special and fun, and many of you are friends that I expect and hope to keep for life! Last, but as the most important of all, I want to thank my family, whose support in all this process and in all my life was crucial, and I can always rely on.

I wish you will enjoy reading this thesis, and may it inspire you with new ideas,

Jean van der Meer

Table of Contents

Summary

Building Information Modelling (BIM) has changed much of the construction world in the last decades. However, BIM is still mostly applied only for the design of new buildings, and much of the building stock is kept in old paper-based archives. This prompted research in the area of Scan-to-BIM, where point cloud geometry is used to produce BIM models of buildings that did not have a BIM model yet. Scanning buildings is also used in the Scan-vs-BIM paradigm, where a BIM model is available, but it is necessary to check whether the real building matches this available model. It can be that the available BIM model is an as-designed file and the building was built with deviations from the project, or the building was renovated without updating the BIM model, but Scan-vs-BIM comparisons are also used to keep track of construction progress.

As construction companies, specially the ones that also work with the operation phase of buildings, start to use BIM models more and more in the management of existing buildings, the importance of keeping BIM models updated becomes increasingly clear. The push for circularity and sustainability also points towards using and reusing buildings increasingly more, and renovating them to make them more sustainable, by extent increasing the need for updating information and models of a building. IFC is the most popular vendor-neutral standard for files of BIM models, but only a small portion of current Scan-to-BIM and Scan-vs-BIM research covers direct generation or update of IFC models based on point cloud geometry. The importance of having updated models and reliable data prompted researching the structure of a procedure that can update an outdated IFC file based on point cloud geometry. The general structure of the procedure is divided in six steps, where, starting with an outdated IFC file, (1) data is collected, (2a) segmentation standards are defined, (2b) elements are segmented and labeled, (3) IFC and Point Cloud geometry are overlaid so their coordinates can match, (4) geometric properties of elements are extracted from the segmented point cloud elements, (5) IFC elements and segmented point cloud elements are matched to each other and (6) the IFC model is updated.

As a proof-of-concept, a method is proposed to automate steps 4 to 6, with an emphasis on the update of IfcWallStandardCase elements in a Manhattan-World multi-room and multi-storey paradigm. The implementation also demonstrates how the height of ceilings (IfcCovering entities) can be updated based on point cloud data, and how IfcColumn entities can be updated. The update of walls is based on matching point cloud walls by their starting point and ending points, wall-by-wall, to IFC walls present at the outdated IFC file. IFC walls at the outdated file that get matched to point cloud data are preserved, and IFC walls that are not found at the as-is, current state of the building scanned, are deleted from the file, with their decompositions such as doors, windows and openings, and relations to other elements, and mentions of them in e.g. list of elements in a storey. Point cloud walls that are not matched to any IFC wall of the outdated file, prompt the generation of new IfcWallStandardCase instances in the IFC file. Each new IFC wall added is made based on a “template wall”, a pre-existing IFC wall in the file that matches the profile of the new wall as well as possible, and preferably from the same region of the building. In this way, the semantics of the model are kept in the new wall, but the geometric properties of that wall and of the layout of walls are updated. To aid the comparison of IFC walls and point cloud walls, a standard for modelling IFC walls and for segmenting point cloud walls is proposed, that helps in comparing and matching walls in the most efficient way possible. The layout of walls is also correctly updated if that standard is not followed, but then more walls are deleted and reconstructed by the tool, if the starting and end points did not match.

The geometry of walls, such as their start, is also differently defined in a volumetric representation, when compared to a surface representation. That plus errors found when aligning IFC and point cloud models, and errors from sensors, can generate geometry with slight, but quite noticeable, misalignments when walls are added to the model based on point cloud geometry. The connection points often have indentations at the junctions with other walls. To compensate for and correct those inaccuracies, an algorithm is created to refine the geometry of the new walls added to the model, after their initial creation. Their starting points are updated to match e.g. horizontal alignments with walls around it if they are horizontal walls (on a floorplan view), or vertical alignments with walls around it if they are vertical walls. Another idea proposed to help matching walls, in a earlier stage, or locate which other walls are around a wall are connected to it, when correcting its geometry and aligning it, is using dynamic thresholds. Dynamic thresholds are thresholds that define if a wall is matched or not, that change depending on the thickness of a wall. Thicker walls get a proportionally thicker threshold, but very thin walls also have a minimum value that guarantees geometry will be matched. Finally, a proposed idea that helps the update of the layout of walls, but can also be used when updating other IFC elements, is that of creating a scope volume around only the scanned area, and only performing checks and matches in this scanned area. This avoids deleting elements in areas that were not even scanned, as it could be implied that they did not match the point cloud data. This is named “Room Mode”, and allows, for instance, using cheaper sensors such as those contained in smartphones, that do not work when scanning a large area. It is often not even desirable to scan a large area, as changes in a building take often place in a limited section of it, and scanning a smaller area saves time and money.

The update of ceiling heights is demonstrated using the procedure developed to update both ceilings defined by IfcRectangleProfileDef profiles and ceilings defined by IfcArbitraryClosedProfileDef profiles. Lastly, the update of columns is demonstrated, as they can pose complex situations where some columns are embedded in walls and others are not, or yet, some are partially embedded. Partially embedded columns may or may not be detected by scanning process, and fully embedded columns are often not detected. Being not scanned, even if they are in the same scanned region of the building, they should not be deleted. It is important however to make a report that notes how many columns are present in the as-designed project, and how many were verified by the scanning process, and how many of those that perhaps were not found should have been found and should have been visible. Columns that should be visible might have their position updated based on a threshold, or be created, if there is no IfcColumn nearby a point cloud column, or might be deleted in some cases.

To implement the methodology created, a python tool was made, called “Point Cloud to IFC Updater”, based on IfcOpenShell to parse and edit IFC data based on the IFC schema, and a OpenCascade-based STEP visualizer, for which IFC files are converted into STEP for visualization. The OCC viewer has some PyQt5 functionalities to create a user interface, that allows to see models and point clouds to visualize the updating process and configuration of the as-designed and as-is statuses. The tool created in python is made open source and available in an online repository.[1]

[1] https://github.com/jeanvdmeer/The-history-of-the-walls

Abstract

Building Information Modelling (BIM) is a methodology that has changed many paradigms in the construction world in the last decades, bringing more efficiency, reliability of data, and solving design issues early on the building process. In the BIM world, IFC is a schema that allows vendor neutral collaboration with open standards, and is therefore the most popular file format for exchanging building models among stakeholders. Despite the large and ever increasing implementation of BIM into new projects, much of the current building stock is represented only by paper drawings or 2D CAD files. Furthermore, for facility management processes, it is important to have up to date data from buildings, that is also reliable. The most popular methodology to capture geometry data from a building site is by performing a laser scan and generating a point cloud of it. In the recent years, methods have been developed to create BIM models from existing buildings. The process is often manual and laborious, but current research focuses on using machine learning models that segment building elements out of the point cloud, label them, and construct building data in BIM models based on that geometry, aiming to automate this process. A limited amount of research however is focused on the update of existing BIM models, and even more scarce research in the update of IFC files, which approaches a more commercial implementation. A research gap was identified, as the need of a procedure that allows the automated update of IFC files based on point cloud data. This was done taking into account the importance of real-time reliable data, and the increased demand to use buildings for a longer period of time and repurpose them, or renovate them, for sustainability reasons. The process was identified as starting with an outdated IFC file, LiDAR data is collected, and based on pre-determined segmentation standards the building elements are segmented from the point cloud. The segmented elements, and indeed the point cloud, need to have their location aligned to the coordinate system of the IFC file and the building geometry in IFC, a process whose automation is covered in other research of the literature, just as the segmentation and labelling. This research focused on the automation of the last steps of the updating process – the geometry extraction from building elements segmented from the point cloud, the comparison and matching of that geometry to that of IFC elements present in the outdated file, and the update based on the comparison. The procedure generates new elements based on point cloud geometry, can remove elements depending on their absence in the as-is state of the building, and can update an element, such as updating the position of a column if a match is found but there is a deviation in position. The demonstration of the process of IFC update based on point cloud geometry is implemented for Manhattan-World walls, ceilings and columns, and is implemented in python using IfcOpenShell to access the IFC schema and OpenCascade to visualize point clouds, building models and the update. The building element type whose update is most extensively discussed and implemented is the IfcWallStandardCase, for which new walls generated take a pre-existing IFC wall as template for semantics, looking for the best fit based on thickness and area of the building, and algorithms are developed to improve the geometry based on the context (e.g. wall connections). A concept named dynamic thresholds is also used, in projects where walls of many thicknesses are used, to create flexibility with a large threshold to match thick walls and a smaller threshold to match thinner walls. Another concept named Room Mode was created to allow performing a check and update with a scan that represents only a part of the model. The code is made publicly available.

Key-words: IFC update, BIM update, point clouds, Scan-to-BIM, Scan-vs-BIM, as-is

List of figures

List of Tables

List of Listings:

List of Algorithms:

1 Introduction

1.1 Background

Building Information Modelling (BIM) has been improving the construction industry and its workflows in many, if not all of its branches. It is a methodology where interconnected information – including geometry – is brought together in a meaningful (*semantic*) way to compose a digital project. Construction projects and operations involving it become much more efficient, by being able to automate the extraction of floor plans, quantities, and details. Notably, improvements can be seen in coordination of the different disciplines involved in the construction process, improved technical visualization and technical drawing extraction, better collaboration between stakeholders, better simulations, cost estimations, progress monitoring, issue and clash management, and Facility Management (FM) (Borrmann et al., 2018).

Whereas the methodology of BIM gained traction by changing the industry of new buildings, much of the housing and real estate stock was built at a time where 2D paper or CAD representations were prevalent. Even for the cases where BIM models were available, in situations where renovations at a section of a building are performed, such as the removal, replacement or addition of walls and doors, the contractors may not share their drawings of the new section, or drawings can still be done in 2D (Beetz et al., 2013; Thomson, 2018; Pas, 2022). Much attention is necessary to existing buildings, and they also present an important market opportunity, given that a high amount of the total ownership cost of a building is spent on maintenance and renovations. In the UK, for instance, at least half of all nationwide construction expenses goes into existing assets (Cabinet Office, 2011).

Another problematic scenario is when deviations occur between the available as-designed BIM model, and the way the building ended up being constructed (as-built). This can happen by natural deviations in measurements, or, for instance, last minute changes in the structural design that are not updated in the IFC files available. Then, the original BIM model of the building will contain obsolete information about the real building. In this scenario, the as-is situation of the building is unknown and the available information on material quantities, measurements in the floorplan layout, and information on building elements becomes unreliable.

Incongruencies in BIM models can also be the result of poor in-situ surveying to produce a BIM model of a building that had no such model available, or human mistakes when creating a BIM model from available 2D drawings (that are often incomplete). To execute a major renovation, or a building disassembly to reuse its materials, or other FM purposes such as installing new systems on a building, or allocating space use in large buildings, or making cleaning contracts for areas of the building, a key step is having precise information of the as-is state of the building. Correct parametric BIM model generation and update can help in solving one of the largest current problems the construction industry faces: namely inadequate reliability of data. Another problem is the lack of data synchronization between BIM and on-site processes, for which BIM model updating based on site conditions can be an important solution (Hamledari et al., 2018).

1.2 Current gaps in the field

Even though much progress has been achieved by the construction industry in the last decades with the advent of BIM, the development of BIM has been focused on planning new buildings (as-designed models), and little attention has been given to BIM tools that allow management of existing (as-is) buildings (Volk et al., 2014; Macher et al., 2017; Scherer & Katranuschkov, 2018; Werbrouck et al., 2020). This might have happened due to the previous (still existing) building stock having its data stored in sparse and often incomplete 2D data, so to avoid all the rework of creating and assembling this data, the BIM process was started with its focus on the new building development market.

A second challenge that hampers the development of as-is BIM models for the existing building stock, that was built at the pre-BIM era, is the burdensome and expensive process of conducting surveying of the spatial information, to then pass this information onto BIM models, in a partially if not completely manual process. The elevated costs to perform geometry surveying in high definition, the specialized staff required to do this survey, and uncertainties on information about the building are seen as some of the limiting factors to achieve broader as-is BIM model development for Facility Management (Volk et al., 2014; El-Din Fawzy, 2019; Kavaliauskas et al., 2022). Nonetheless, point cloud acquisition is seen as the best resource and industry standard in digitalizing geometries of real world objects (Werbrouck et al., 2020). Furthermore, Thomson and Boehm (2015), and Werbrouck et al. (2020) consider the upcoming advancements in BIM tools to soon become as important to the management of existing buildings as they currently are for the market of new buildings.

With the recent efforts of society and institutions to make the built environment and industry more sustainable and energy efficient, an increased demand of work in renovation and facility management is expected for two reasons: Buildings are expected to be used for a longer lifespan in order to reduce waste and raw material use, and many existing buildings are also expected to undergo renovations and retrofitting to make them more sustainable (better insulation, heat pump installation, double or triple glass installation on windows, etc). In the Netherlands, for instance, 7 million homes and an extra 1 million non-residential buildings are expected to be made free from the gas-network by 2050 (Ministerie van Economische Zaken, 2020). Scherer and Katranuschkov (2018) consider the improvement of BIM applications to retrofitting of buildings a process of utmost importance to achieve the necessary upcoming sustainable and energy efficiency goals. They also consider the creation and maintenance of BIM models of existing buildings as an important challenge to be overcome, which is also endorsed by Thomson and Boehm (2015) and Valero et al. (2021).

Looking into the future, the new buildings that are built using BIM, and older buildings for which a BIM model was created after completion, will also need methods that can keep their information (geometric and semantic) up to date, to help facility management. The current moment could therefore pose a significant opportunity to look for tools and workflows that collect the as-is condition of a building and update available BIM models.

In this context, a third problem in the implementation of BIM into the management of existing buildings is that, when renovations are done by a contractor, there is often no guarantee that changes will be updated into the as-designed BIM model. If they are, they can stay with the contractor and not be made available to the facility manager. When the updated BIM model is needed in the future, for management reasons or a renovation, the models may not be available with the contractor anymore.

For the case of “do-it-yourself” renovations performed by owners, the odds of updated BIM models are even lower. Those small renovations can nonetheless include relevant layout and model changes, such as changes in interior walls. BIM models that are made based on 2D drawings, can also have wrong information, either by lacking information at the 2D model that was "guessed" when creating the BIM file, or by human error. The “Tsunami” of data mentioned by Salazar et al. (2019) illustrates a fourth problem, common in facility management: there is a high amount of data produced in the design phase, but it is often not clear what of this data is useful to management, and some of the needed data can be missing in the model. A better collaboration between design and management stages and processes is thus necessary.

Relevant problems found in the literature are thus that: **1.** BIM is applied mostly for the design phase, the methodology is not fully applied into the lifespan of a building and its management. **2.** Surveying the geometry and data of a building is expensive and a specialized activity, which hampers the generation and update of BIM models for existing buildings, which is often slowly and manually made. **3.** Even when building models are available they are often outdated and don’t represent the current state of the building, and improper methodologies and data exchange between the parties involved keep those models outdated. **4.** A new BIM paradigm that meets the needs of Facility Management and provides facility managers with the reliable and up-to-date data they need is necessary.

Facility management (FM) can therefore be aided by the creation of better tools, and applications of existing technologies to keep the as-is version of the building model updated for better management. Tools that capture the as-is state and overlay it with the as-planned project to update the BIM model can also be useful for renovations, apart from its usefulness for daily facility management reasons.

1.3 Research problem

In the literature review, a knowledge gap for a methodology that uses both a Scan-vs-BIM and a Scan-to-BIM approach to update BIM indoor models is identified. Bosché et al. (2015), for instance, claim that there is a significant gap in the field, but see promising advances in integrating Scan-vs-BIM and Scan-to-BIM approaches to process 3D as-built data more efficiently in buildings that already have an as-designed BIM model.

There is a growing demand on the market for more as-is data on the condition of buildings, for preservation of historical buildings, or renovation of modern buildings that have outdated BIM models. Trends such as sustainable retrofitting of buildings, circular disassembly, or management focused on using buildings for a longer time, also increase the importance of knowing the as-is state of buildings. Renovations require an up to date model, in order to be executed in the first place. Therefore, a methodology is desired to save time and money by means of automation or semi-automation, in the process of updating an outdated BIM model into its as-is configuration. The use of more accessible tools to collect data can further improve the cost-benefit ratio.

With the advent of web and linked data technologies and frameworks, the advantages become increasingly clear of keeping information systems that are the most updated as possible (and easily updatable), instead of a file-based perspective. To connect this building information in a meaningful way, standards as IFC and structuring ontologies for it have been developed. Some methodologies

even go from point cloud models directly into graph databases (Werbrouck et al., 2020), to enhance the current Scan-to-BIM processes.

Gourgechon et al. (2022), in their review of the state-of-the art on the creation of BIM models from point clouds for building interiors, explain that much of the research in reconstruction from point clouds is focused only on the 3D reconstruction, ignoring further information and semantics. Furthermore, they found that very few authors have focused on producing results in IFC or other proprietary format, which points to a lot of further research needed in this area that is more directly applicable to the market.

A step that will help to keep information and geometry of building assets up to date, is the creation of a procedure that collects as-is point cloud data on site, compares this data to the available BIM model, and adjusts the BIM model based on deviations found. This procedure should also keep as much as possible from the semantics of the original BIM file. To make it more applicable and follow recognized standards, it is good that this procedure is developed for IFC files. This research aims to develop this procedure in a way that is as automated as possible, in a proof of concept, and test it. The **research problem** is thus summed up as describing and testing **the procedure that updates IFC files of indoor environments based on geometry extracted from point clouds, with an emphasis on the update of walls.**

1.3.1 Problem analysis and research questions

Even though there is a high need in the market for the creation of as-is BIM models (Thomson, 2018; Werbrouck et al., 2020), the return on investment is still too low, with expensive professional surveying machines, and expert staff needed to execute surveying as a limitation (Volk et al., 2014; El-Din Fawzy, 2019; Kavaliauskas et al., 2022), and due to the laborious process of manual model updating (Volk et al., 2014; Esfahani et al., 2021; Gourgechon et al., 2022; Tang et al., 2022). Therefore, workflows and tools that reduce labour and time spent in this process pose an opportunity of research. The current high costs can be reduced by using more accessible means and processes to collect data, increasing adoption by the market, and by bringing automation into the model update.

The main research question is defined as: “how does the procedure that automatically updates IFC files of indoor environments based on geometry extracted from segmented and registered point clouds, with an emphasis on the update of walls, and reuse of model semantics, look like?”. The following two sub-questions are formulated:

1. How can the implementation of this procedure be made more accessible to users?
2. How can the new IFC geometry generated, based on point cloud geometry, be improved and harmonized to the rest of the model?

1.3.2 Research objectives and limitations

The research takes place between a Scan-vs-BIM and a Scan-to-BIM paradigm: the BIM model is assumed as existing, which asks for a comparison – Scan-vs-BIM –, but outdated, which asks for changes or creation of elements, based on scan information – Scan-to-BIM. The workflow studied shall generate an updated model based on identified discrepancies. Those discrepancies, found through LiDAR scanning of the interior of the building, will be used to update the model and obtain a sound "as-is" IFC model. Identifying the best tools of the state-of-the-art in Point Cloud-to-BIM applications is therefore part of this research. Among the tools and methodologies available, the study

identifies which ones can be integrated in an as seamless and automated process as possible, to have a procedure that will save as many working hours as possible. **Describing how this tool works, its operations, steps taken, and data files used as input, used during the process, and which files and information can be retrieved as output, and developing a user interface to integrate those steps and make the procedure clear with visualizations for stakeholders, are goals of this research.** To validate this procedure, tests with different datasets will be performed in a case study.

Developing on the research questions, the following objectives are also established:

1. Prove the feasibility of automated IFC model update based on geometry extracted from segmented point clouds, with emphasis on the update of IfcWallStandardCase elements that follow a Manhattan-World Assumption, and further demonstrations with IfcCovering and IfcColumn elements, using semantics present at the model;
2. Make the implementation of Scan-to-BIM and IFC update more accessible by using data collected from smartphone LiDAR sensors, and reduce the time of surveying by allowing the use of a scanned area of only the section that needs to be updated, instead of a scan of the entire building or floor;
3. Create algorithms that allow the improvement of the initially generated new IFC walls, to harmonize the generated geometry to the previously existing geometry around it e.g. in corners;

Even though all steps involved in the necessary procedure will be discussed and researched, this work will be focused on the steps that update the IFC file when segmentation, registration and labelling of a point cloud is already done. That means that the focus lies on how to extract geometry of point cloud segmented elements and based on this data and heuristics compare those segmented elements to IFC elements in the as-designed IFC file and remove or add elements if necessary. The semantics of the model is taken into account when adding new elements.

Possible limitations are limitations in measurement from the LiDAR scan itself – the IFC model available might not reflect the real building perfectly, but a point cloud also has a limited measurement precision. In the current state-of-the-art elements such as e.g. walls, columns and ceilings can be identified by semantic segmentation, and even wall types might be inferred from heuristics (and will, in this research). However, even with this degree of automation, the engineer responsible for the update might still manually check if the right type of material or other properties were assigned to the newly created element. That happens because sensing technologies can mostly only retrieve geometry information, and even in methods that try to infer e.g. materials based on color information, this is not fully reliable, and other internal properties of the element cannot be assessed.

The current research goes more in depth specifically in the updates of walls, as it is one of the most important components of a building. The focus lies in the update of IfcWallStandardCase elements, and even though the ideas should be applicable into non-Manhattan World scenarios, the application of the procedure was developed for the Manhattan World assumption, for simplicity and demonstration purposes and due to the scarcity of datasets that do not follow the assumption. Instructions for the application of the procedure into the so called “diagonal walls” will however be discussed. Several types of connections between wall elements are possible and the procedure is tested both in a simple experimental model as in a large multi-storey multi-room dataset, which proves its robustness into wider applications. Furthermore the procedure is flexible enough to also work with partial scans and test and update only a limited section of a building, which is much more

realistic in cases of renovations and saves costs, also making the use of cheaper tools more accessible, as they are less precise for large scan areas. The tool can compare and update walls with any type of thickness, correct inaccuracies in wall connections and dimensions, and look for the most fit wall type available in the model, when creating a new wall element to update the model, helping to preserve semantics. The update of ceilings and columns in IFC is also discussed and tested, though not as extensively as walls. Columns are updated both in their position and quantity – columns might be deleted or added into the model to match the as-is state, and ceilings have their height updated.

1.4 Field data and field site

The thesis is developed in collaboration with the company BIM-Connected (based in Eindhoven) and their client Erasmus Medisch Centrum in Rotterdam. The Erasmus MC has made available IFC models of their *Ca* building, for validation of some of the tests performed in this thesis. The building premises can be accessed for collection of data (e.g. Point Clouds) to further test the framework developed. Data is also used from the Atlas building at the Eindhoven University of Technology, for which an IFC model is available for floors 8 and 9, and point cloud data is collected by the author for the development of this research. The Haus30 dataset from the DURAARK European project, available in their GitHub repository, is also used as a large dataset to test the procedure developed. The datasets used, together with the implementation of the procedure created in python are made available on GitHub[2]. In the case of the EMC data only a small IFC file made by the author, based on the layout of a section of the hospital, is made publicly available.

1.5 Reading guide

At the beginning of this work, the literature review will structure the knowledge basis on which the research is developed, clarifying some of the research questions, explaining how point clouds work and are applied in the Architecture, Engineering and Construction (AEC) industry, and how the different types of data collection are performed, as well as when each one is recommended and their advantages and limitations. New applications in the area, and their current and future potential in scan-to-BIM and scan-vs-BIM procedures are also discussed. The current workflows and limitations in generating BIM models based on point clouds are investigated, as well as the state-of-the-art in matching building elements from a point cloud with elements from an as-designed project. Application of BIM models into Facility Management, limitations and prospects in the area, and FM principles relevant to this work are finally discussed, to close the literature review.

The next and third chapter discusses the Methods deployed in this research, using tools and principles found in the literature and discussing tools needed for each step of the proposed workflow. The steps taken are further explained, from data collection and file formats that are used in the procedure, to standards in creating building models that might aid keeping them updated. The general functioning of the procedure is outlined, with a brief discussion of how the procedure, that is implemented in python, could also be applied in a web-based paradigm. The algorithms and principles used in the updates of IFC walls, columns and ceilings are then discussed, besides the introduction to "Room Mode", the checking paradigm that allows the scan of only a small section of the building to be used

[2] https://github.com/jeanvdmeer/The-history-of-the-walls

in the update. Lastly, the chapter discusses the implementation of those principles in a python tool and user interface, called "*Python Scan to Updated IFC",* that allows visualizing the process.

Section 4 explains the results of the process and application into a case study, where different datasets are used to demonstrate the update of ceilings, columns and walls. The chapter starts with a brief discussion of the results of the scanning process with some recommendations of scanning methods based on the results of data collection. After that, the results of the update of walls are shown, with demonstrations of simulations performed that allow testing how the procedure works in several different cases that are important when walls need to be updated. Some problems found during the research are shown to exemplify how they prompted improvements and the addition of new steps into the procedure that make the end-result more accurate. After that, the data used in the update of ceilings and columns, and their tests and results obtained are outlined.

Chapter 5 ends the report by outlining the conclusions of the research, discussing how the research questions and objectives were solved, the most important findings that were developed along the research, contributions to the field and recommendations to apply this procedure on a larger scale.

2 Literature Review

In this chapter, the theoretical basis is outlined, elaborating on how data collection and processing is done. Furthermore, the existing methods and solutions for the application of point clouds into the Architecture, Engineering and Construction (AEC) industry, with their respective state of the art, are described, helping to frame the context of this research and the pertinent research questions.

A review is done on the devices commonly used for point cloud collection, the different modalities of point cloud collection and methods involved therein, ways of storing this data, and which type of data is applied into which context, shown in Section 2.1 The review follows to describe in Section 2.2 the different types and stages of application of point clouds in the Built Environment, explaining the concepts of Scan-to-BIM, Scan-vs-BIM and contextualizing Scan-to-as-is. The state-of-the-art of those technologies is outlined, supplemented by new methodologies and technologies that might become relevant to Scan-to-BIM and model update. A further breakdown of Scan-to-BIM is made in Section 2.3, describing the recognition of walls, floors, ceilings and openings from point cloud geometry. The chapter ends with a discussion of the application of the approached methodologies into BIM management and Facility Management in Section 2.4.

2.1 Data Collection and Point clouds

Surveying of a building site is the process by which information is collected on its condition or construction progress. 3D laser scanning is considered the primary geometry measurement method for the BIM process, for new buildings, and for the generation of BIM models for old buildings (BIM Industry Working Group, 2011; Thomson, 2016; Werbrouk et al., 2020) . This section will detail the collection of point clouds from 3D laser scanners, problems that are often found, and their characteristics and use for the built environment.

2.1.1 What is a Point Cloud?

A point cloud is a digital representation of the surface of a real world object or environment. This representation is stored as a large number of points in space. The most basic form of a point cloud is that of storing the coordinates of the 3 axes (x, y and z), relative to an origin point, and often RGB information on the color of each point is also stored. This (colorizing) process is aided by the information of a camera sensor: the point cloud (depth and location of each projected point relative to the laser sensor) is overlapped with the color information in each estimated point from a camera sensor. The files of a point cloud can be simple ASCII text files where each line contains the information of each point, in formats such as .xyz or .pcd.

Because a point cloud is scanned by emitting laser pulses in an oriented pattern, and measuring their return time to recover the 3D location of points around the scanning position, the distances are, at first, relative to the scanning position. If the scanner position is known, as in many terrestrial workflows that use a total station, then the position of the points can be estimated based on global coordinates, often aided by precise GPS data. This structured type of data collection, with panoramic views of each collection point, depth maps, and precise positioning location for each scanning point, information on which points were collected at which place, as well as information on how one point of data collection is related to the others, is called structured data collection. One clear distinction of

structured point clouds is that their points are indexed by proximity, and the data collection also happens in a structured way with predetermined collection points. Unstructured point clouds can have points that are spatially very close to each other being collected at different points in time and with very different indexes. This could happen e.g. with a mobile scanner, where the person performing a scan goes over the same area at different points in time. In unstructured data collection point cloud data can thus be collected without storing panoramic images, and locations of where each collection point is at. Unstructured data collection will generate file formats that store data in the format mentioned at the beginning of this section, with only x, y and z coordinates, and possibly RGB information and/ or normal planes of the points.

A problem faced when collecting a point cloud from a given position, is that because some objects can be in front of others, objects can block the collection of geometry of other things behind them. This process is called occlusion, and to avoid it, scanning points from several collection points is recommended and a standard practice. In Figure 1 an example is given of how occlusions can occur.

Figure 1: A Terrestrial Laser Scanner (TLS) is collecting data from a specific collection point, but the person in front of the wall impedes the projection/collection of some laser points on the wall, blocking the "sight" of the sensor from that angle, creating an occlusion (shown in grey)

Given that in an unstructured point cloud approach several frames (or placements of data collection) will be used to form a complete point cloud with as little occlusions as possible, several point clouds will have to be merged. Each one of those frames, or smaller point clouds, will have points stored with their coordinates relative to the distance to the laser sensor. The different point clouds can be merged and re-structure the coordinates of their points, to be relative to a new global base point of their coordinate system (0,0,0). These steps can be done using several algorithms, where one of the most famous is the ICP (Iterative Closest Point, described first by Besl and McKay in 1992) algorithm, that efficiently automates the matching of points that are shared between point clouds, moving and merging them into a single one (Thomson, 2016). Determining the spatial transformations that align two point clouds and sets them in the same reference system is a process called registration. SLAM (Simultaneous location and mapping) algorithms help to locate the position of the sensor collecting the point cloud, across time, and generate the complete point cloud as the sensor moves, which can aid the working of ICP, for instance, making its seed values for translation and rotation converge faster.

2.1.2 Methods of point Cloud collection and files

Frydlewicz (2018) gives a broad overview of how methods and technologies of 3D ranging and divided and connected to each other, which is shown in Figure 2.

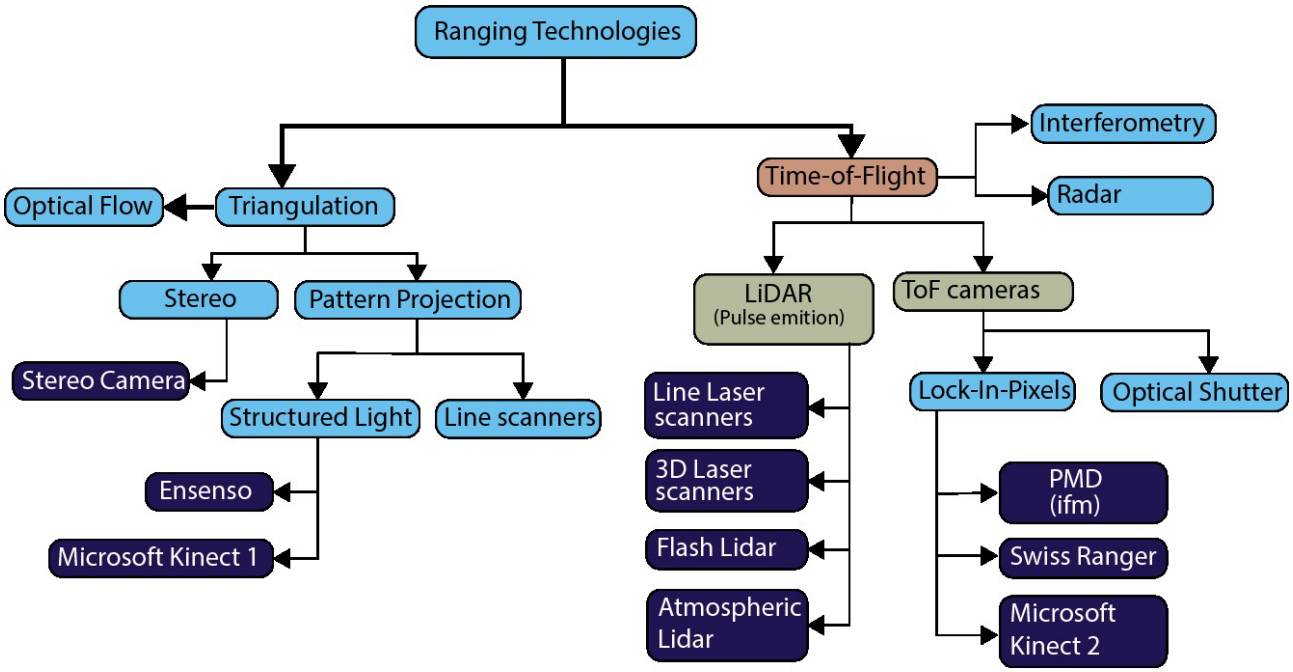

Figure 2: Overview of many of the main ranging technologies available on the market and how they relate to each other. Based on Frydlewicz (2018).

Photogrammetry

Photogrammetry is a method of passive acquisition of 3D geometry (Pereira, 2023) which means that the reconstruction is based on light obtained from the environment, and not a light source connected to the system of sensors. The sensor used is often a simple camera. This makes the technology cheap, and well suited for outdoor environments, where natural light is abundant. Picture-based reconstruction also performs well in real-time assessment of depth, which makes it suited for applications such as autonomous navigation (Pereira, 2023).

The development of algorithms to identify common features in a large array of pictures, and in this way reconstruct depth information is more complex than other applications, such as laser scanning. Furthermore, the extraction of features in regular surfaces such as walls, with changes in illumination, makes this methodology especially vulnerable for applications that focus on the interior of buildings. Photogrammetry methods of point cloud and geometry reconstruction often adopt the Structure from motion (SfM) framework, which reconstructs images based on common features spotted at different angles of an object. To execute this process, it is recommended to take as many pictures around the object as possible, from as many different angles as possible, and try to include the entire object in each picture. This is clearly easier to achieve for buildings exteriors, when compared to interiors, which makes this methodology widely adopted for outdoor building or infrastructure elements. It can also be cheaper than laser scanning methods, as it only depends on a camera sensor, and can be equipped on drones.

Disadvantages of photogrammetry for the collection of point clouds in indoor environments are pointed by Furukawa et al. (2009) and can be that the complex algorithms that merge different images to reconstruct perspectives and shapes, have a difficult time handling the repeated textures of indoor environments in walls. Another problem is that it is desirable to keep as much of an object in view as possible, when performing geometry and point cloud reconstruction from pictures. The nature of indoor environments, however, keeps most of the other rooms out of view in most pictures, which creates problems for the application of photogrammetry into the scanning of interiors.

Laser/LiDAR scanners and total stations

A total [positioning] station is a highly precise surveying equipment that has, however, a lesser density of point projection. That makes it suitable for cases where extremely precise measurements are desired, but the geometry of the measured object is simple, or at least the information required is only a few distances between elements. Total stations can also be used to track the positioning of laser scanners with high accuracy, which can help in the later process of registration and georeferencing (Thomson, 2016).

Laser scanners, on the other hand have good accuracy, but lower compared to that of a total station, but can collect a much higher density of points, allowing for complex geometry to be extracted and digitally represented. They are also often used in surveying complex element arrangements, such as refineries or other pipe or plumbing systems (Thomson, 2016). Laser and LiDAR (Light Detection And Ranging) scanners are the standard and recommended method to obtain point clouds in interior environments (BIM Industry Working Group, 2011; BIM Task Group, 2013).

Spring (2020a) and Thomson (2016) point the beginnings of Terrestrial Laser Scanning (TLS) as military and defense-driven, whose application found its way into the industrial and commercial world around 1987-1998 (Spring, 2020a). The first applications were used as a computer vision mechanism for planetary rovers, docking space-shuttles, and satellites. Even though this history of laser sensing is often overlooked, the capability of laser sensors to locate themselves, and reconstruct their position in relative space, is inherited from the space-navigation framework they originated in (Spring, 2020a).

In his comprehensive review of TLS systems' history, Spring (2020a, 2020b) divides their history in four stages: at the first stage, development was mostly done by military, defense, and government institutions such as DARPA and NASA; at the second stage the technology expanded into the industry, and at the third stage the traditional tripod based systems started to appear. The fourth stage relates to the present where laser scanning is used in remote navigation, mixed reality, sensors become cheaper and are applied even in smartphones and automobiles.

There are three main methods to use laser systems for the collection of point clouds: Optical triangulation, Time of Flight (ToF), and Phase Shift (PS). Optical triangulation found many applications in the beginning of the laser scanning era, while ToF and PS later dominated the market due to their higher accuracy (Spring, 2020b). Optical triangulation works with a camera and a laser placed on the sensing device, whose distance is known, and allows the acquisition of geometry by triangulation. It can be done with a projected laser point (Figure 3), or with a projected line, that sweeps a laser projection along an object, and reconstructs the profile of the surface at that section, for each position of the projected line. Some technology applications like the first version of Xbox

Kinect 360 used this line projection technology, and helped to introduce point cloud technology into the wider market. This period of commoditization of laser sensors helped to make the technology more accessible, which characterizes the current era of autonomous driving and more accessible scanners (Spring, 2020b).

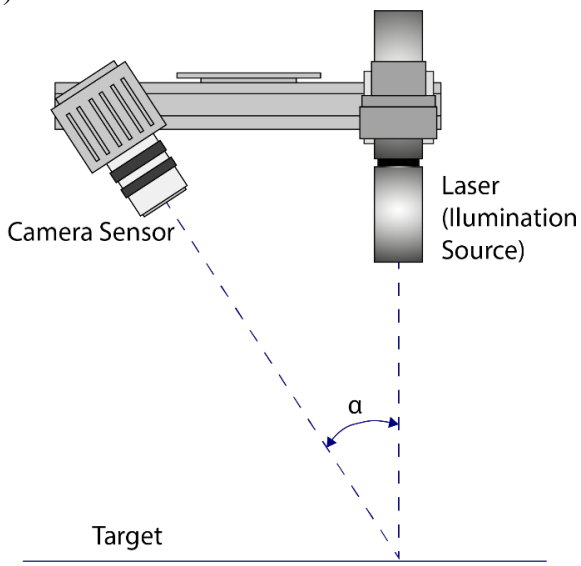

Figure 3: Camera sensor and laser placed on the same device, with a distance and angle position between the two of them known, allowing triangulation

Phase Shift sensors, as well as Time of Flight sensors, work based on the speed of light and the time a projected beam takes to come back. The difference for Phase Shift sensors is that the light is modulated, controlling the frequency of the beam in a continuous wave pattern, and measuring the difference in phase for the reflected beam to compute travel time, and therefore distance. Several different frequency waveforms can be used simultaneously to process more data and avoid ambiguity in decoding the data (Thomson, 2016). Figure 4 shows the Phase Shift method exemplified.

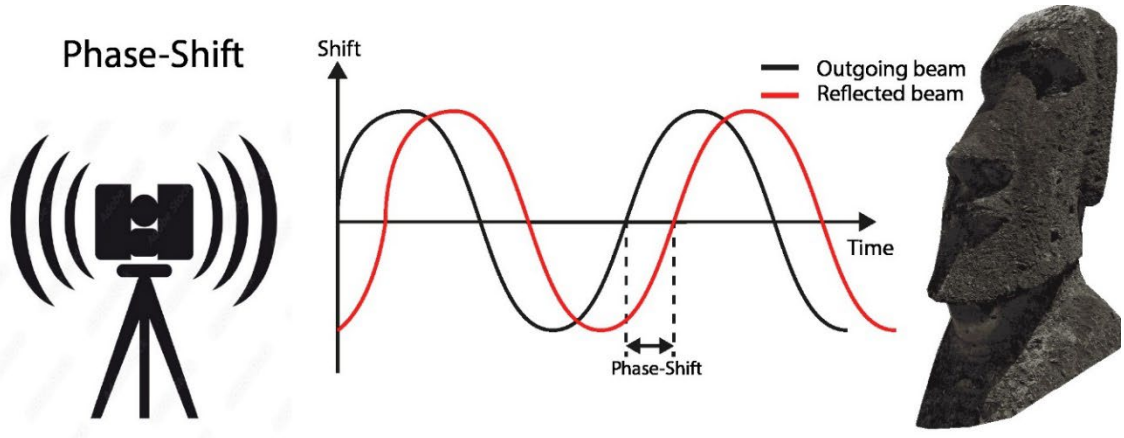

Figure 4: Phase Shift sensing, based on Spring, 2020b

Time of Travel (ToF) is a general methodology where sensors obtain distance measurements by emitting pulses of light (or, in a few cases, a continuous stream of light, (Pereira, 2023)) and measuring the time taken by them to return. The next pulse beam can only be sent (by the same laser) after the signals from the previous pulse is received back and processed (Thomson, 2016), which can be quite fast, taking into account the speed of light. The light bounces in the object ahead and the photons that come directly back (Figure 5) and those that come back via an alternative path (Figure

6) are received and decoded to assess the distance measurement. Figure 5 shows the process of distance assessment using a ToF methodology.

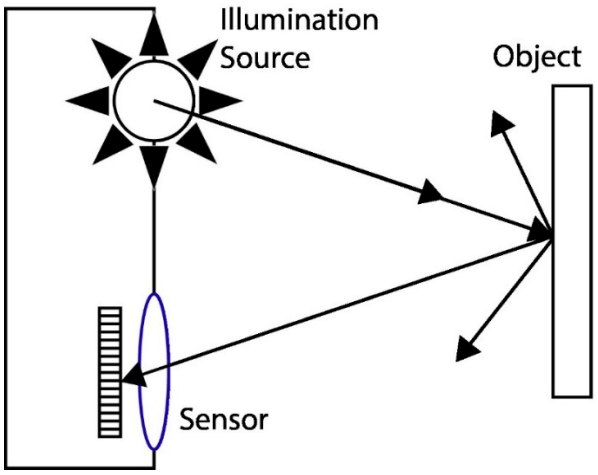

Figure 5: Simplified functioning of a ToF sensor. A controlled illumination source present on the device sends luminous pulses to an object, and based on the time to travel back to the sensor, and the distance of the sensor to the illuminator, the object's distance is estimated. Based on: Teizer, J., & Kahlmann, T. (2007)

In their article released in 2022, Gyongy et al. explain the principles behind dToF (direct time of flight) single photon imaging. One big advantage is that by collecting photons returned by the environment, and taking into account the emitted laser beam(s), it is possible to create an histogram with all the returned signal and separate the peaks on the histogram graph as returns from the main object, compared to background noise (which is more or less continuous). It is even possible to identify photons that take alternative paths, as not all photons reflect directly back in the exact direction of the receptor, and many take an alternative path, before finding their way back to the sensor, as also shown in Figure 6. This method also allows recognizing semi-opaque surfaces and objects behind them. The general scheme is shown in Figure 6.

A LiDAR sensor works using the same principles of ToF, but has more projected laser beams and operates at a higher frequency, having usually a larger range, whereas ordinary ToFs sometimes only use infrared beams. Non-LiDAR ToF sensors often use a flash of continuous [infrared] light onto an area, and reconstruct its depth by analyzing the reflected light from the different parts of the projection (Frydlewicz, 2018; Pereira, 2023). A LiDAR sensor, on the other hand, uses lasers in the form of focused beams that project a point, with often an array of multiple lasers, that have more power for a longer range and higher precision.

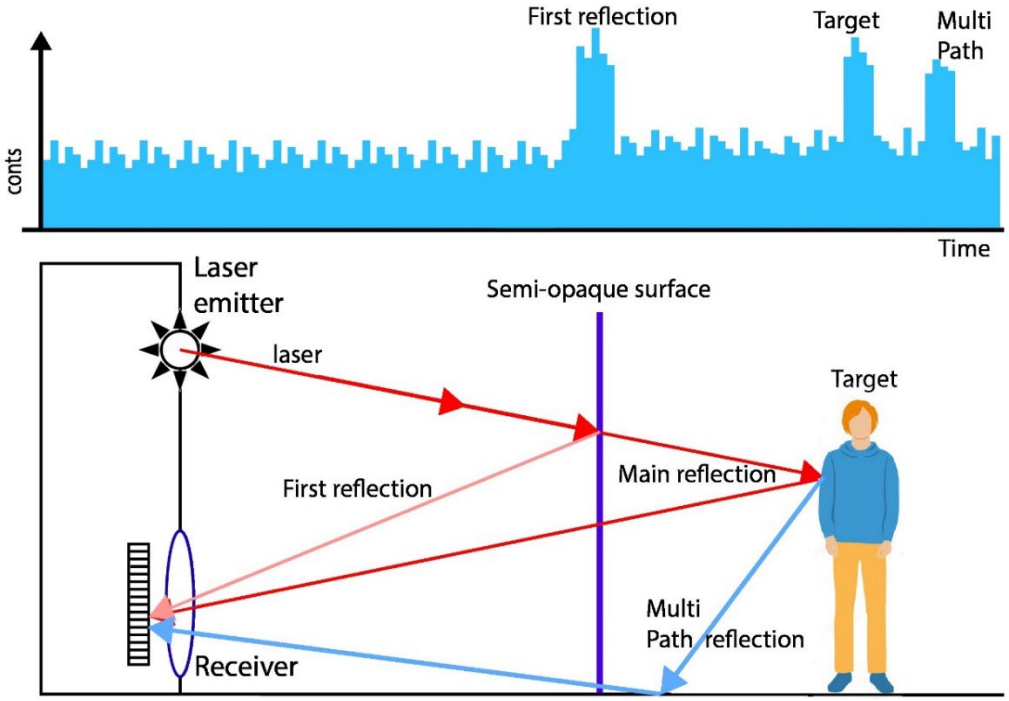

Figure 6: Distinction of an object behind a semi-opaque surface, with the histogram on top showing the general noise, and the peaks in signal from the reflection at the glass surface, and then the peaks from the target itself (used to assess the distance), and the peaks in signal from multipath reflections (take longer to return to the receiver). Based on: Gyongy et al. (2022).

Because of the high speed of light, many pulses can be emitted per second to collect several points and make a highly dense point cloud. To compose a LiDAR system, a few elements are required (Domi, 2023):

1. A laser emitter, to produce the pulsed laser beam;
2. An internal clock of high precision, to compute the time taken by light to come back, and compute the distances. Also, for Mobile Mapping systems, that will be approached soon in the text, the clock is important to connect the different locations in time of a moving sensor;
3. A Scanning system, that structures how the points will search the environment. In a stationary (e.g. tripod) scanner, this can be a tilting mirror that has its movement tracked, which, with several laser pulses per second, and a tinting mirror and a rotating base, can cover a large environment with laser points. Figure 7 exemplifies this system. Increasing the amount of laser emitters can improve point cloud density, but makes the system more complex and expensive, which was for a long time a reason that made LiDAR expensive;
4. A receiver/photodetector, to capture the returning laser pulses reflected from the environment;
5. A processing unit, to process all the parallel stages and build the point cloud that is being collected based on the data surveyed.

The differences between normal ToF and LiDAR make LiDAR more accurate, producing a higher resolution in scans, and performing better in a wider range of lighting conditions. Compared to a Photogrammetric approach, ToF-based systems have better performance in a wider range of lighting conditions, because they do not depend on an external lighting source (Pereira, 2023). Some natural outdoor conditions such as heavy fog can be a challenge for some LiDAR systems, however, for applications such as autonomous driving. Reflective or transparent surfaces, such as glass, can in turn pose a challenge for indoor applications, due to reflections or projections of beams into outdoor areas (Macher et al., 2017; De Geyter et al., 2022; Roman et al., 2023).

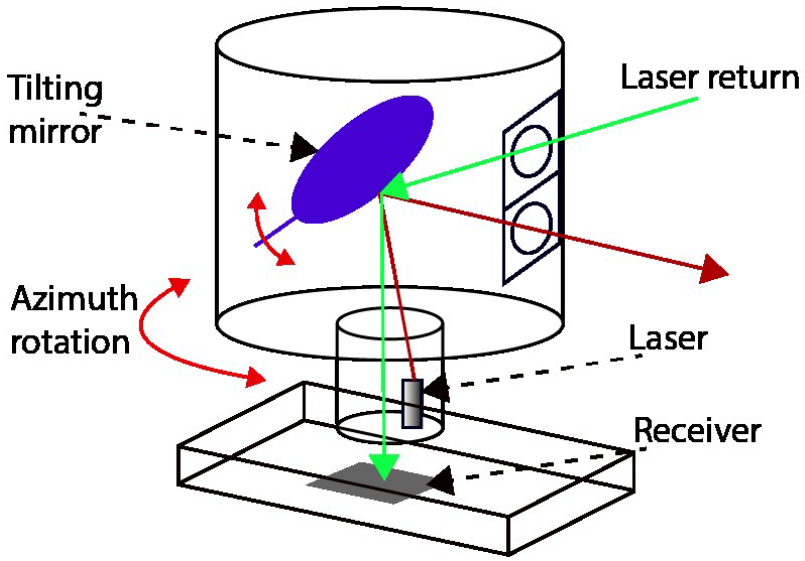

Figure 7: Simplified scheme of a LiDAR scanner with a moving mirror

Mobile Mapping and new technologies

Terrestrial static laser scanning has a high precision of measurement due to the absence of the need of corrections of measurement due to staying static all through the process of point collection. The sensor is moved to a new static place on a tripod and the distance between measuring points is known, and height is kept the same all along. Less transformations and rotations need to be done to merge the different point clouds obtained, which confers high accuracy to the measured points. The process of static measuring is often done with the help of a total station, so each measuring point has its position known with high accuracy. One of the main disadvantages of the terrestrial static approach is, on the other hand, the increased amount of occlusions, when compared to mobile collection (Previtali et al., 2018). This happens due to the limited amount of collection points with a static, tripod based surveying method, collected at a constant height. Objects that are in a low or high position can be partially covered by other objects, or not have enough geometry collected, for example a beam blocking the view to installations close to the ceiling, or complex pipe systems, where some pipes only have some sides of their profiles' geometry captured. The latter problem could, for instance, create problems in recognizing the profile of a pipe to automate the labeling and identification of a pipe geometry in a routine that automates BIM generation from point clouds.

Another problem of surveying with a static terrestrial scanner and a total station, aiming for high accuracy, is the amount of time taken to setup all the machines and measurement (Thomson, 2016). As an alternative to situation where the problems mentioned above need to be avoided, technologies of Mobile Mapping Systems (MMS), Mobile Laser Scanning (MLS), and Indoor Mobile Mapping Systems (IMMS) were developed to scan continuously while moving, whether carried by a person (handheld, in a gimble, or as a "backpack") or attached to a car/trolley or a robot. This continuous scan means continuously producing point cloud frames of the collection points at each given moment, and merging them together. Current advantages of systems carried by a person are their accessibility, where surfaces of difficult access to wheels of trolleys or robots have some limitations of places they can reach (Thomson, 2016). In the future this limitation for robots could be circumvented by the use of flying drones, for instance. Limitations of IMMS, however, pointed out by Previtali et al. (2018), are that they can have more noise in their data, when laser beams pass through openings, generating

sparse data in those areas. Point density can change considerably on different sections of a point cloud. If a mobile scan is being performed, reduced point cloud density can be found in areas that were scanned more quickly, or that had too many outliers due to drifting, for cases where the point cloud is collected and filtered at the moment of collection. Static laser scanners also produce different point densities. Because the beams of the lasers are emitted within a certain angle of each other, objects that are closer to the scanner have higher densities, as the same small angle implies a smaller distance between points, for points close to the scanner.

To aid the registration of those moving systems and track their position over time to merge the point clouds collected at each frame as effectively as possible, a few methodologies are applied. One of them is on the hardware side: sensors that allow 6 degrees of freedom (movement and rotation along the 3 coordinate axes), to track the positioning and change of positioning along time of the device. Some sensors found in devices that help checking those movements, are, for instance, an altimeter for height change, a compass for rotations along the Z axis (Thomson, 2016), and gyroscopes and accelerometers also help in tracking movement along the axes. This hardware input can be coupled with algorithms such as ICP (Iterative Closest Point, described in 1992 by Besl and McKay), to automate registration and eliminate the use of targets. Targets are target shaped elements that are placed vertically and horizontally in areas scanned by TLS or photogrammetry systems to help scaling and merging point clouds. ICP iterates two point clouds among the 6 degrees of freedom to find the best fit (Thomson, 2016). Figure 8 shows a scheme of the 6 degrees of freedom. A sensor that measures movement in the 6DOF is often called an IMU, an Inertial Measurement Unit.

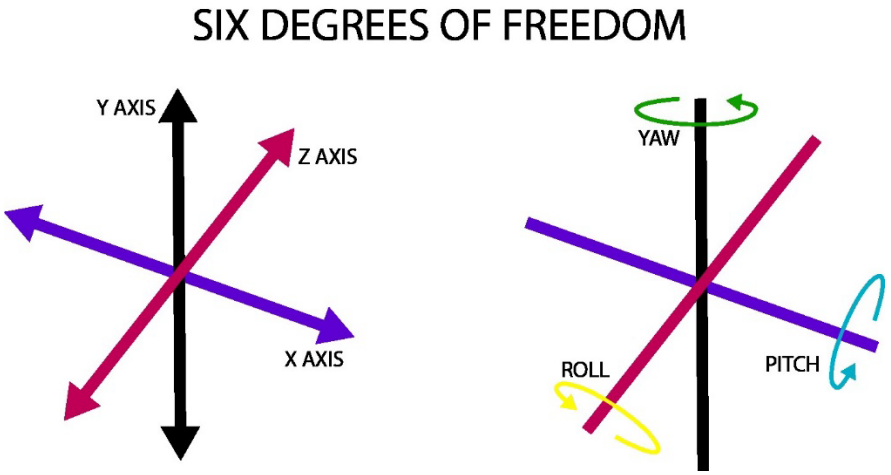

Figure 8: The 6 degrees of freedom. Movement up and down is also alternatively called Heave, left and right Sway, and forward and backward named Surge.

Besides the advancements brought about by IMMS, the application of LiDAR sensor into smartphones is a recent phenomenon that presents promising chances for the Built Environment. The technological advancements in the industry of semiconductors made the rise of solid-state LiDAR sensors possible by use of vertical cavity surface-emitting lasers (VCSELs) as emitters, and single-photon avalanche diodes (SPADs) as receivers. First applied in the field of autonomous driving, this combination allowed the price of LiDAR systems to go down substantially, and now VCSELs and SPADs entered the smartphone market via the Pro line of iPhones and iPads produced by Apple, with the aim of increasing the possibilities of AR applications and image capturing (Lee, 2020).

The big advantage of VCSELs and SPADs in popularizing and commoditizing LiDAR systems is that because they are based on semiconductor technologies, they can be produced with semiconductor wafer machines which drastically increases the scalability of production and reduction of price (Frome, 2020; Lee, 2020), unlike big systems with rotating mirrors and complex laser arrangements that made assembly and production very expensive. For comparison, the first LiDAR scanners were around USD75.000,00 against new sensors, which can be incorporated in smartphones. Furthermore, because their efficiency follows Moore's law, their market has a promising growth future (Frome, 2020). The smaller size of sensors also allows them to be applied into all kinds of popular portable devices (headsets, drones, etc), and the lifespan of VCSELs is also increased (Frome, 2020).

SPADs, as the name indicates, can detect individual photons, which allows good precision in lasers with lower power, but all of that sensitivity generates noise. This noise on data is solved through a complex post-processing stage. One limitation is that a VCSEL's laser is less bright than that of traditional systems, which makes a challenge to achieve a high range (200m and above), which can be a challenge in autonomous driving applications. Apple's sensor itself is an example of that: so far the range is limited to 5 meters. The lower power on the laser's side can also bring challenges to decode the returning signal from environment noise, but advancements in out-of-band light detection have made the current advancements possible (Lee, 2020).

Another factor to make this VCSEL + SPAD pipeline work well and increase precision and quality of the data captured, is merging pipelines of decision processes that are done across different sensors, and sharing information along the several decision steps in the process. This increases the total quality of decisions and geometry reconstruction. This was pointed out by Yang (2021), who noted that the fusion of sensors in their pipelines makes the process continue even when one sensor is covered. The general methodology of this sensor fusion technique is laid out in the patent proposed by Xu et al. (2020). If the LiDAR sensor is covered, depth information continues to be assessed using the camera and other movement tracking sensors (with less precision, of course). Likewise, the addition of information from the camera sensor, based on Machine Learning models, adds information in the depth decision process mainly based on LiDAR, and increases accuracy of point clouds. Other movement sensors, and their improvement over time probably collaborate in this process too. This points to a future where the use of fusion between several sensors and real time processing of the data might increase the accuracy of the data collected beyond the mere capabilities of the hardware used.

While some interiors' surveying applications have their minimal required accuracy regulated, in countries such as the UK (Thomson, 2016), applications focused on Facility Management often require less precision (Thomson, 2016; explained in higher detail at section 2.3). Tamimi (2022), doing tests with an iPhone 13 Pro, and devices such as the viDoc RTK GNSS and a DJI Osmo 3-Axis Gimbal, found that the results allowed the application of smartphone data collection in smaller practical applications, focused on outdoor terrain, in this case. Chase et al. (2022) also found good results, in their case for interiors, especially with the aid of control points to make the model more precise. Both Chase et al. (2022) and Tamimi (2022) emphasized the promising future of the use of smartphones for laser scanning of indoor environments.

Google, responsible for Android, has also developed APIs focused on Augmented Reality. One of the first was project Tango, and currently ARCore, with platforms such as the Depth API. Because of the multitude of devices serviced by Android, and the diverse price ranges those devices cover,

Google opted out of optimizing their algorithms for LiDAR sensors, and chose instead for camera based depth perception, such as in its Depth API. This has limitations for indoor applications of regular surfaces such as walls, however. Nonetheless, Google had, already in 2014, a pilot within its Tango project, with a device that contained a high resolution ToF laser sensor, and ultrawide cameras, that was sent to researchers, institutions and universities to testing and development (Goldheart, 2014). At the end of 2016 Google released in the consumer market the Phab 2 Pro, in a partnership with Lenovo, as a development of its Tango project. The device was designed to have enhanced AR applications, and had a ToF sensor and 6 degrees of freedom sensors (Wokke, 2017). The consumer reception was moderate, as AR was not too developed or common in the market back then, but research on smartphone-based point clouds was done using this device (Liu et al., 2018).

SLAM (Simultaneous Location and Mapping) is a type of algorithm by which the position of a device is tracked to reconstruct angles and placements in which pictures or 3D scans were collected, and help in the reconstruction of the 3D environment – and to possibly help the location of a robot in this environment. Nocerino et al. (2017) point out that in an AR framework, the application of SLAM into a smartphone 3D scan of an environment can help in giving real time feedback on areas that were already scanned and areas that still need to be scanned. Apple's RoomPlan API is an example of the application of this concept in the Built Environment: the indoor reconstruction is tracked in real time on the screen, exemplified in Figure 9.

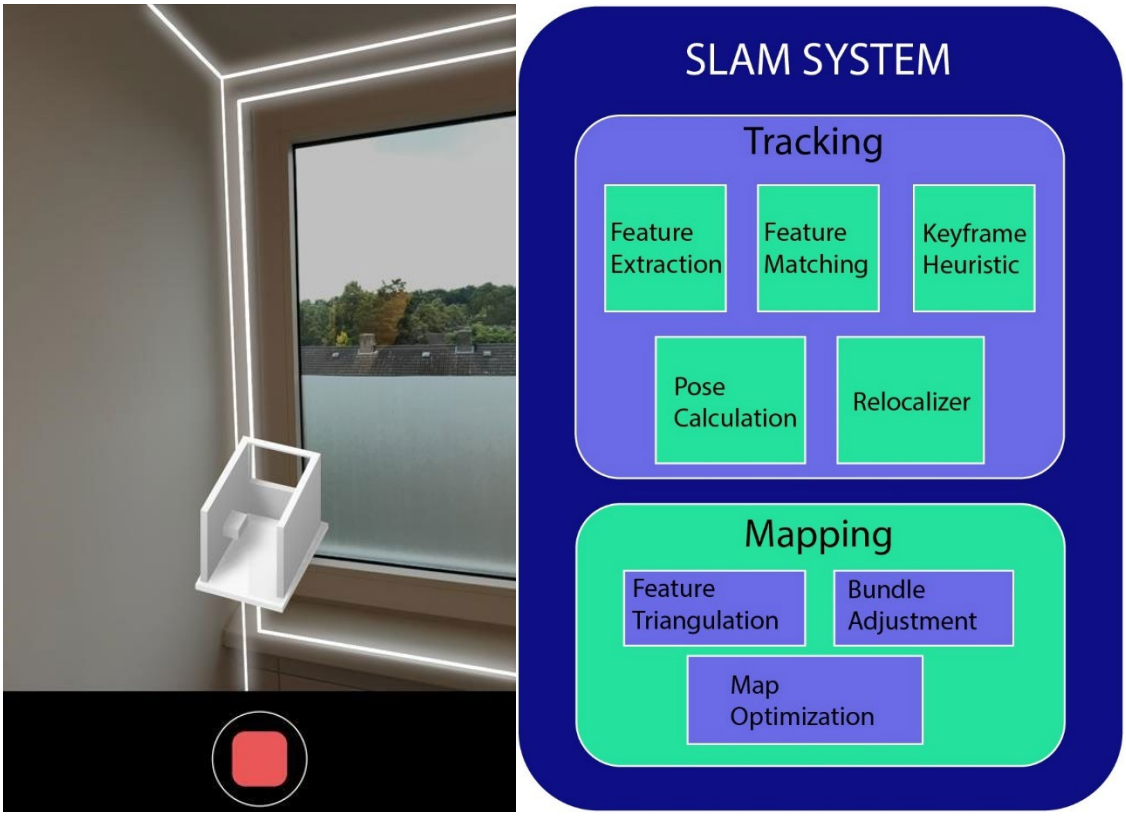

Figure 9: To the left, real time tracking of scanned walls (and visualization of missing walls), using SLAM by RoomPlan's API. Source: The Author. To the right: suggestion of necessary elements for a SLAM system in smartphone-based 3D mapping. Based on Nocerino et al., 2017.

File formats

Some of the most important format files for point cloud storage are ASCII, LAS and E57 (Khalsa et al., 2022; Pereira, 2023). ASCII files are characterized by text files that may or may not contain a header with metadata, and after that each line of the text file represents one point in space, with its x, y and z coordinates separated by a character such as a space, comma, or pipe. An example can be seen in Figure 10, with x, y and z coordinates and then RGB values separated by spaces.

Figure 10: Simple xyz point cloud with RGB values and no header

Additional information, such as intensity, normal of the point, and color information in RGB can be available (values ranging from 0 to 255 in red, green and blue). This additional information is stored in the same line of the point. This way of storing data makes it easy to be read by humans, and provides a straightforward way of parsing data, but comes at the cost of occupying more space in memory compared to machine readable formats such as binary formats, and reduced working performance (Khalsa et al., 2022; Pereira, 2023). A number of file formats are based on ASCII, such as .pcd, .xyz, .pts, .ptx, and .txt.

LAS (or LAZ, in its compressed form), is a format developed and maintained by the American Society for Photogrammetry and Remote Sensing (ASPRS). Those files are often used by vendors of laser/LiDAR scanners, and are encoded in binary format. Those files can contain a lot of information pertaining to the recording of data and the proprietary equipment used to obtain it. Points are often stored by chronological order of acquisition (NASA, 2022). The binary encoding of the data makes it consume less memory and perform faster in many applications (Khalsa et al., 2022; Pereira, 2023).

E57 is a format file that can hold entire projects and has many terrestrial scan positions stored, while keeping all the translation vectors and rotation matrices that align the different scan positions into a coherent global solution. It can also store 2D images relating to each point cloud scan and metadata related to each image, and can be very useful for spherical scans. The file has typically a header, an XML section that structures the hierarchical tree of the file, with references to the data that is stored in binary formats in a different section of the file.

Depth maps are one methodology used by some devices to store 3D information and later reconstruct an environment. Each movement in space of the device, as well as positional information of the cameras, is stored. With one image representing de depth of each pixel (or region of pixels, as depth

maps are often of low resolution compared to the color image), and one color image from a common camera, a combination of the information contained in both images is done in a process called back-projection to create 3D coordinates of the environment. This can be done using open-source libraries such as Point Cloud Library (PCL), Open3D and OpenCV (Pereira, 2023). Because algorithms of 2D image recognition are quite advanced, this depth map framework allows easier segmentation of elements in a 3D environment, compared to 3D segmentation directly out of point clouds. Figure 11 shows an example of depth map being used in an indoor context to reconstruct building elements.

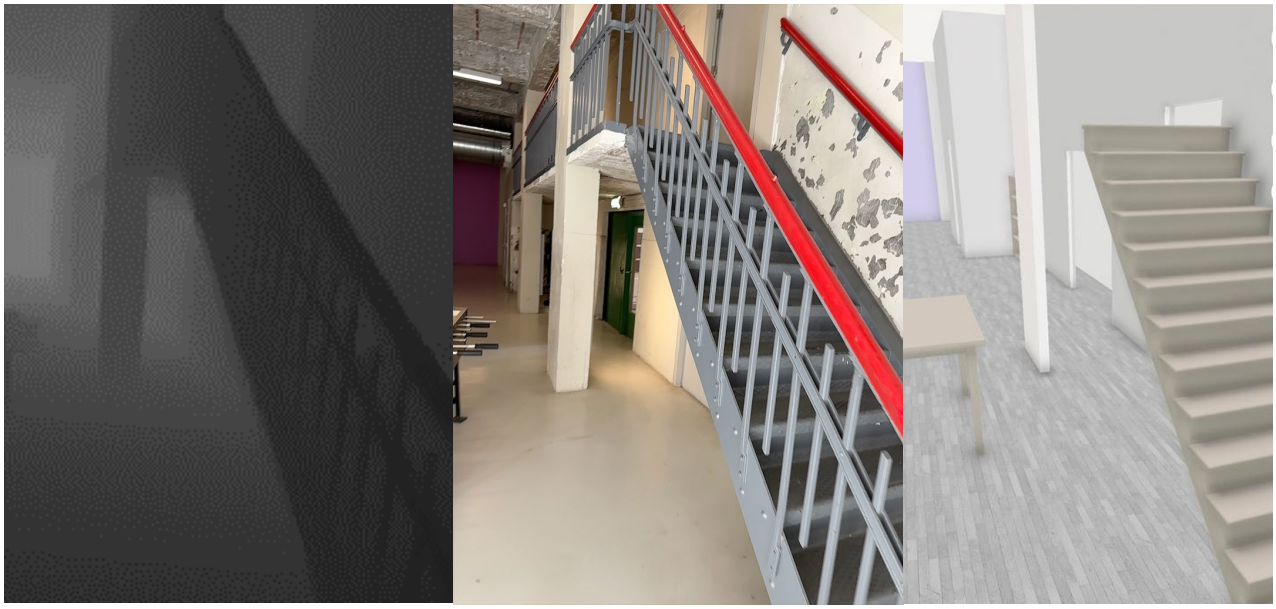

Figure 11: 11a, 11b, 11c: 11a, A depth map with every pixel as the Z coordinate value on greyscale. The depth sensor collects the depth of the center pixel with high precision, and the depth of the other pixels is estimated by tone. 11b: A picture collected by a color camera at the same moment in time and same camera position. With both images, the depth can be reconstructed and segmentation by machine learning algorithms is made easier, and point clouds can also be made by sequences of those "keyframes". 11c: A 3D CAD reconstruction with some semantic segmentation made by the RoomPlan API: walls, floors, openings, doors and the stair are recognized and reconstructed from the 3D information collected, but as the API is limited to 1 floor of reconstruction, the mezzanine was not reconstructed.

2.2 Moving from Point Cloud to BIM

Point cloud collection is considered the best method of digitally capturing real world geometry. Often a Scan-to-BIM or Scan-vs-BIM approach is used. Scan-to-BIM refers to the process of creating BIM models based on the scan of an environment, and Scan-vs-BIM implies the comparison of an existing BIM model with the as-is point cloud retrieved from building. A Scan-to-BIM approach is often used in older buildings that do not have an available BIM model (Werbrouck et al., 2020), whereas a Scan-vs-BIM approach is most often used to track construction progress (Golparvar-Fard et al. 2009; Kim et al. 2013; Chen & Cho, 2018; Dülger, 2020). However, Scan-vs-BIM approaches can also be used in facility management (Son et al., 2015; Chen & Cho, 2018), and to compare differences between as-built and as-planned models (Bosché et al., 2015, Chen & Cho, 2018). Scan-vs-BIM and Scan-to-BIM are also a promising solution to update outdated models, which can occur after a renovation is done and insufficient documentation is available (Thomson, 2018).

Even though Point Clouds are the most indicated way to collect geometric data from real world geometry, the information they provide alone about the object being scanned is quite limited, outside of geometric information, and sometimes RGB color information on each point, or normal vectors and sensor trajectory. Semantic information of a building model can be said to be any information that does not pertain its geometry (Thomson, 2016). The advancements in BIM helped to surpass a stage where only geometry of a drawing is seen, but to indeed be able to access meaningful properties and parameters of objects and models, whether they are material type, physical properties such as fire resistance, thermal conductivity, cost, layers, properties of glass, or relationships to other elements in the project such as containment or structural connections. There is no information in point clouds about boundaries and containment relationships between objects (Werbrouck et al., 2020), but fortunately, in the last years, large progress has been made by tools that use processes such as Machine Learning (ML) to identify building elements within a point cloud (Bruno et al., 2018), with interesting tools developed for instance in the wall modeling processes described by Ochmann et al. (2016, 2019) and that of Bassier and Vergauwen (2020). This points towards a future where sensing technologies can see their application becoming more and more useful to the update or creation of BIM models.

2.2.1 Scan-vs-BIM

Scan-vs-BIM is a process often used to compare the as-designed characteristics of the building to the as-built characteristics of the building. This is often done under a paradigm of following the construction process' progress by overlaying the geometry of designed (BIM) and built (scanned) elements. If a given building element is expected to be placed on site, on a specific day being analyzed, based on the building schedule, but no volume is found in its place (by assessing the point cloud's as-built geometry), then a construction delay can be identified. The basic idea of Scan-vs-BIM – comparing a designed BIM configuration to that found on site – can also be used to update BIM models of buildings that were already completed, but have project deviations or were renovated.

Regularly monitoring the building site helps to keep track of construction delays, which helps in purchasing the right materials and deploying the workforce at the right time, increasing construction quality with the improved documentation (Hamledari et al., 2017). The monitoring process is however time consuming, expensive and prone to error (Hamledari et al., 2017), but the use of imaging and geospatial technologies to automate or semi-automate progress checking has been found to be cheaper, make the process faster, and is more accurate and reliable (Turkan et al., 2012). Turkan et al. (2012), for instance, introduced a method to track construction progress based on a BIM 4D-model (with construction schedule data) that improved the recognition of building elements found at scans. By acquiring a point cloud and manually aligning it to the BIM model using 3 or more points, an automated process starts that outputs the current status of each activity and element.

The Scan-vs-BIM methodology of Bosché et al. (2015), focused on MEP systems' pipe reconstruction, for instance, consists of 4 steps. In the first step every point in the point cloud is assessed to assign it to the building element, at the BIM model, it is closest to. A threshold, in their case of 50mm, is assigned to filter points that are too far away from any BIM object (e.g. clutter). After that, a check is done to compare the normals of the BIM element's surface's normal to the normal of the point, and a threshold is applied, in their case only points that deviate in less than 45 degrees are accepted. In this way, every building element should have a group of points assigned to it, and

there will be a remaining group of points that were rejected by the thresholds. The second step is to find which points are occluding building elements, obstructing the sensor view. This helps to calculate the percentage recognized, and the confidence of the segmentation by knowing the total area recognizable, instead of using the entire area of the element. The third step is the production of a synthetic as-planned point cloud: a virtual point cloud is made based on the surface of the BIM model, with similar resolution to that of the as-built point cloud. The fourth step is the object recognition: a match is created of recognized surface areas of the point cloud that match to a BIM element.

Dülger (2018) developed a method to track construction progress of prefabricated structural elements using point clouds in a Scan-vs-BIM paradigm. Collecting point clouds from an automated camera at a construction crane, the point cloud of each floor is compared to an IFC file of the expected progress at a given moment of the day. Using IfcOpenShell to parse an IFC file, and linking its elements to an equivalent STEP file that corresponded to the IFC, the faces of building elements are divided into grids, that are voxelized and get a bounding box (around each building element in the STEP file). This grid formation on the building elements' faces is based on the method suggested by Tuttas et al. (2014), helping to increase accuracy compared to other voxelizing processes. Around the bounding box of each voxel grid, a test is done to see if there is enough point density (at the point cloud) to classify the as-designed volume as fully constructed.

Thinking of comparing a BIM model , or previously known status of a building, to its current state (including construction progress tracking), Meyer et al. (2022) created a method to detect changes in buildings using point clouds. The method differentiates and detects dynamic elements (such as doors), or clutter that is covering a building element, by voxelizing an environment and thereafter classifying voxels in voxels that are empty, occupied, or unknown. Unknown voxels have incoherent results from different collection points (epochs), such as a person that could be standing in a place at one point of data collection, and be out of the scanning range in the next point of data collection. Or yet, an object that covers a wall from one angle and not from another. This special attention to voxels that are ambiguous or unknown, instead of empty or occupied, is also seen at some ray tracing algorithms used in opening detection, discussed in Section 2.3.3.

While most Scan-vs-BIM methods simply take the as-designed geometry, and check the presence of point cloud points around building elements to track construction progress, Hu and Brilakis (2023) proposed a method based on an unsupervised clustering algorithm to have more accuracy. Taking input from an IFC model, the method is robust against on-site deviations in position, orientation, and scale compared to the as-designed model. The IFC elements are used by the unsupervised clustering algorithm to segment elements from the point cloud and identify completed elements even when their configuration at the building site deviates from the as-designed model. The segmentation of BIM elements for the Scan-vs-BIM comparison also helps in filtering out clutter.

Collins et al. (2022) developed a methodology based on Graph Neural Networks (GNNs) that unites both Scan-vs-BIM and Scan-to-BIM concepts, aiming at matching as-designed elements to as-built elements. The method is based on the cosine similarity of their feature space (both point cloud and BIM model), taking geometry and connections into account.

2.2.2 Scan-to-BIM

In contrast to the current paradigm of Scan-vs-BIM, that is mostly based on volumetric comparison of BIM elements and point clouds, Scan-to-BIM is a process that aims to generate complete BIM information aided by remote sensing data, such as 3D laser scans. The process is still largely manual and time consuming (Xiong et al., 2013; Macher et al., 2017; Jung et al., 2018; De Geyter et al., 2022; Tang et al., 2022; Roman et al., 2023), and much research has been done in this area for more than 25 years, aiming for a highly automated process, but with limited results (Nagel et al., 2009; Thomson, 2016). Furthermore, the research community is quite fragmented in their approach, and many different researchers have widely diverging approaches in attempting reconstruction (Pintore et al., 2020).

In recent years, however, important breakthroughs have been made with the aid of Deep Learning for segmentation and classification of point clouds, while automating the reconstruction of BIM files still remains a large challenge in the area (Tran et al., 2019; Gourguechon et al., 2022; Roman et al., 2023). Automation in the generation of BIM models is nonetheless very desirable, as it might increase standardization (different manual modelers tend to produce different BIM models out of the same data source) (Esfahani et al., 2021), making data and processes more reliable. Esfahani et al. (2021) also found that semi-automated generation of BIM models from point clouds delivers more accurate and precise models, when compared to both manual procedures by untrained and trained professionals.

Whereas the process of building reconstruction is at more advanced stage for building exteriors, what is discussed in works such as (Boulaassal et al., 2010; Dore & Murphy, 2014), the reconstruction of the geometry, and the segmentation of elements for *interiors*, is still at a comparatively initial stage (Previtali et al., 2018). This happens because this area is more complex, containing systems, more clutter and occlusions making accurate data collection difficult (Macher et al., 2017; Jung et al., 2018; Tran et al., 2019; De Geyter et al., 2022), and requiring more refined algorithms and methods to identify some indoor building elements (Díaz-Vilariño et al., 2017, Previtali et al., 2018). Another problem found in the Scan-to-BIM process in general is the lack of semantics obtained from point clouds, which makes the Scan-to-BIM limited (Pătrăucean et al., 2015; Tran et al., 2019), and with less information than the as-designed BIM model (Macher et al., 2017). Tang et al. (2022) point out the current incompleteness of data, such as occlusions and noise, as one of the main limiting factors of Scan-to-BIM automation. As mentioned in Section 2.1.2, mobile point cloud data collection could partially help in solving those problems, as it reduces occlusions. To help in the poor semantics often obtained by simple collection of point clouds, Valero et al. (2012) proposed a method where RFID (Radio-Frequency Identification) technologies are used together with laser scanners to enhance semantics of produced models. Those RFID tags can contain information about the building or interior element, and help building reconstruction in collecting material information or segmenting elements whose identification is more challenging for current algorithms. Those technologies are also useful for facility management, where furniture items can be checked and registered when managing large buildings.

Another limitation often found in many models is the Manhattan assumption. Originally proposed in 1999 by Coughlan and Yuille, it assumes buildings wherein rooms are composed exclusively of

orthogonal planes. Even though that is mostly the case for many types of buildings, such as office buildings, that also leaves out many buildings that have slanted walls, curved walls, or even chamfered walls or walls at a different angle. Non-conventional ceilings, dormers and such typologies cannot be represented under this assumption either. The Manhattan assumption has, however, brought computationally efficient methods to reconstruct buildings that fall under its prerequisites.

LiDAR technologies started as a means to the reconstruction of urban surfaces, executed by sensors attached to planes, and this characteristic and its limitations still plays a role in the Scan-to-BIM limitations and challenges found today. Whereas the mapping of urban spaces (in formats such as CityGML) is done by representing the visible surfaces, and acquiring these surfaces with imaging technologies, BIM standards such as IFC often have volumetric representations of the complete design of the building, which includes encased elements that are not visible after construction (Nagel et al., 2009). An example would be pipes embedded in walls. Sometimes an element can also have only some of its faces visible, a visible beam that crosses multiple rooms: its upper face is not visible, and even though it is connected to the ceiling, the line or plane that represents their interface in reality means that there are two planes, one for the beam and one for the ceiling, and the solid objects are distinct entities. Across different rooms the beam also stays the same entity, even though this information is not accessible looking at the visible surfaces. Building knowledge, and assessing information such as position and adjacency is thus necessary in the step of moving from a surface based model into a Constructive Solid Geometry model (CSG), such as IFC (Nagel et al., 2009).

Macher et al. (2017) divide the Scan-to-BIM process in three steps: geometry modelling of components, assigning a category to an object and material properties, and establishing relationships between components. Bassier et al. (2017), on the other hand, divide the Scan-to-BIM process in four stages: point cloud generation, segmentation phase, classification phase and reconstruction and modelling phase. *Point cloud generation* is covered in Section 2.1 of this report. For Roman et al. (2023) *segmentation* is the division of a point cloud in point groups, named *segments*, based on similarities and the acquisition of local features. This can be done by finding geometric primitives (cubes, planes, etc), by using edge detection algorithms, or region growing algorithms – where a starting (seed) point is used and points of similar features are found around it, or yet by other machine learning algorithms. A combination of those is also possible. *Classification* consists of labeling the segments found, and can be done with ML in a *supervised* process, where the model is trained with many segments of similar type manually labeled, and the process can also be done without training data, based on user parameters (*unsupervised*). The supervised approach is more common for labeling of building point clouds (Roman et al., 2023). Macher et al. (2017) did an approach based on parameters to detect primitive elements, such as planes constituting ceilings, after that making 2D binary pixel maps of ceiling-level cross sections to find room layouts, and from there reconstruct walls. Collins et al. (2022) went all the way to classify a segmented element not only e.g. as a column, but to also track it by element ID. Assuming an existing as-designed IFC file for the building, this method finds which as-designed element has the highest chance of being the particular segmented group of points being analyzed. This method could be essential in updating existing as-design IFC files that have discrepancies to the real (as-is) building. The *reconstruction* step involves the final acquisition of geometry and BIM model generation.

Thomson (2016) points out that the wider application of point clouds into the AEC industry was heavily aided by the decision by two major BIM authoring tools of natively allowing users to work

with point clouds, without third-party plugins. Bentley systems pioneered by, in 2009, integrating the engine of Pointools into microstation, and later in the same year Autodesk followed by incorporating point cloud engines related to aerial LiDAR for Civil and Aero 3D, and running point clouds in AutoCAD. Later, Autodesk worked on a dedicated point cloud platform named ReCap, after acquiring AliceLabs.

Much of the current commercial tools, and methods developed in academia, only generate primitive shapes and meshes that cannot be directly used by BIM authoring tools or converted into BIM files (Macher et al., 2017; Gourgechon et al., 2022). Some tools as EdgeWise and plugins such as ImagineIT Scan-to-BIM, however, make semi-automation of building reconstruction possible, for some building elements, and are integrated to tools like Revit. Other research such as those of Agnostopoulos et al. (2016), Bassier and Vergauwen (2020), and that of Valero et al. (2021) have made it possible to reconstruct BIM elements from point clouds directly into the IFC format.

Agnostopoulos et al. (2016) start with the assumption of already detected and segmented building elements, and based on their borders, go on to create walls, floors, ceilings and spaces in IFC. In their process of producing Ifc Walls and spaces based on point clouds, Anagnostopoulos et al. (2016) first detected boundaries, using both object boundaries and also constraints, such as a wall has to be bounded by two orthogonal walls, and after that enriched the IFC model with room and space labeling. IfcWallStandardCase entities are used to describe walls, and IfcSlab entities are used to describe floors and ceilings. Evidently, based on the first step, the Manhattan assumption was adopted.

In their method, Agnastopoulos et al. (2016) first construct bounding boxes with Cartesian coordinates to generate walls, ceilings, and floors. The borders however have to be often manually adjusted. This is aided by a script that checks and classifies whether the coordinates of the corners of the walls overlap or have a gap in between them. Once the walls form airtight rooms, the IfcSpaces are detected. For this, a check is done based on the center point of each wall, and comparing it to the center point of adjacent walls, and the expected center point for a Manhattan-World situation. This is done for each wall, and if all four walls are connected in the expected way, a space is created.

As an alternative of BIM model generation from point clouds, Roman et al. (2023) suggests pre-built modifiable models, that can adapt the instantiation of each parameter and adapt to the point cloud being studied. Rausch and Haas (2021) suggest a similar methodology where a “proto-BIM” model has many dynamic parametric possibilities and can adapt its location and shape into point cloud elements. This latter approach is discussed further in Section 2.2.3.

Tang et al. (2022) applied shape grammar rules to aid their BIM model reconstruction. Those shape grammars are constraints that help making the reconstruction process more accurate and give it semantic value. One common approach they found in the literature for indoor reconstruction is to first separate planes and geometric primitives, to detect main structures like walls, and after that do a watertight room and space partition, and segmenting openings as a last step. In their research, they described the data generated in GML (Geography Markup Language) format, to store semantics and relationships between components, and make the use of the files reconstructed from the point cloud more applicable in BIM authoring tools.

Whereas the largest part of current Scan-to-BIM research has been conducted mainly on the reconstruction of basic and structural building elements, such as walls, floors, ceilings and openings (Pintore et al., 2020; Valero et al., 2021; Gourguechon et al., 2022), a few authors also made efforts to apply Scan-to-BIM into elements such as MEP (Mechanical, Electrical and Plumbing) components. Valero et al. (2021), as part of the EU-backed BIMERR program, created an extensive framework where, by means of TLS point clouds and pictures of a building interior, an IFC model is semi-automatically reconstructed of both structural and MEP elements. On top of that, the framework also has a Scan-to-BIM editor that is connected to a BIMERR database of materials and component information, that allows the user to semantically enrich the IFC file to produce a final IFC for energy analysis. This is an important step, as much information cannot be retrieved from the geometry+RGB colours information contained in a point cloud.

The Scan-to-BIM tool proposed by Valero et al. (2021) allows the user to insert information such as the type of window glazing or door and wall properties for thermal and energy analysis. The methodology is based on first generating an IFC model for the structural elements (walls, ceilings, floors, openings), which is done almost fully automatically (there are three automated steps and one feeds data that is used by the next tool), and after that the MEP elements are inserted into the IFC using recognition and allocation based on the 2D images collected on site – a second point cloud, this time photogrammetric, based on the indoor pictures, is made, and has its coordinates matched to the TLS point cloud to aid in the process. Some possibilities like grouping ifcSpaces into ifcZones are also included in the tool, to help in per-zone energy analysis. It is worth noting that the proposed structural element IFC generation step, proposed in the paper, is the most automated step of their proposed framework, which also makes it one of the most automated scan-to-IFC tools available in current research. The authors of that paper point out, however, that sometimes inconsistencies may arise in e.g. the IFC model's geometry, so the produced files need to be checked by a modeler.

Another interesting research on Scan-to-BIM applied to MEP is that of Bosché et al. (2015). This work focuses on identifying discrepancies between as-built and as-planned pipes, due to changes done in the field that were not updated into the BIM model. For that, use is made Scan-to-BIM, Scan-vs-BIM and updating frameworks, identifying and reconstructing pipes based on the Hough transform (which would be a Scan-to-BIM approach), and aligning laser scans and as-designed BIM models to check, with thresholds, overlapping geometry (Scan-vs-BIM).

Another interesting advantage of uniting both paradigms is that elements are identified as specific elements by ID within a list. This often happens in Scan-vs-BIM paradigms – whereas a Scan-to-BIM paradigm is most often concerned in identifying whether a segment of points is e.g. a column or not, in Scan-vs-BIM the idea is knowing whether *that* segment of points is *a specific* column of a certain *ID*. Most Scan-vs-BIM approaches, however, are more limited in their semantic recognition and labeling, doing simple geometry overlapping tests, so uniting both perspectives makes discrepancy detection much more efficient. Due to their geometric complexity, MEP elements often deviate from the as-planned layout due to unforeseen clashes or changes. Knowing accurately the as-built condition can help remodeling, repairs and extensions of e.g. HVAC systems, and help Facility Management in general (Bosché et al., 2015).

Whereas a further breakdown of the Scan-vs-BIM steps done by Bosché et al. (2015) e covered in Section 2.2.1, his approach of first comparing BIM model and point cloud, identifying the as-designed

elements found, to then use a Hough transform to find other elements that were not in the as-designed version, expands the possibilities of element recognition procedure. By allying Scan-to-BIM and Scan-vs-BIM it is possible to identify elements that are extremely displaced, and would otherwise not be detected with only a threshold, and detect new elements that were not present in the as-designed project in the first place.

Most of the research found in the literature does not assess the precision of the results obtained by reconstruction, compared to a ground truth model. In their study, Jung et al. (2018) did an analysis of the average deviation of coordinates of some corners, in automatically reconstructed projects, such as walls or door corners, compared to a manually conducted Scan-to-BIM reconstruction. The results for their method were of around 3cm of deviation in each coordinate, or around 9cm of total displacement, on average.

2.2.3 Scan to as-is BIM and new methodologies in the market

Many bottlenecks in the development of the BIM reconstruction and update's industry, and its state-of-the-art, have been mitigated in the recent years by developments in academic research, technology, and commercial solutions. Terrestrial Laser Scanners, that were limited to static tripod-based platforms, started being implemented in mobile platforms, which allowed many more points of capture and less occlusions. Drones became cheaper and more accessible for aerial laser scanning, mostly based on photogrammetry, even though solutions with both photogrammetric and laser sensors are starting to show up as well. A lot of research has been done on applying LiDAR into robotics and computer vision, for autonomous driving and indoor navigation, which points to a future where 3D scanning and reconstruction might be automated not only on a software level, but on a surveying level too. The development of solid state LiDAR sensors has made their costs, now scalable and silicon-based, plummet, making them increasingly more accessible. Many kinds of gadgets, notably smartphones, are starting to be equipped with those sensors, bringing computer vision and accurate measurements into mainstream use.

Application of smartphones into LiDAR scanning

Making use of the smartphones *Lenovo Phab 2 Pro* and *Asus ZenFone AR*, that work based on Google's former 'Tango' platform, Liu et al. (2018) conducted research to form floorplans based on RGB+depth video information, obtained by the smartphones' sensors. While point cloud information can be very useful for accurate geometry measurement and segmentation, the state of the art in 2D image object recognition is much more advanced than other forms. Due to 2D RGB images containing condensed and, storage-wise, inexpensive information about objects' features, they are more efficient when analyzing environments and recognize objects in it, compared to 3D representations of objects. Therefore, the method of Liu et al. (2018) uses Deep Neural Networks (DNNs) to estimate the geometry and layout of a room based on depth information from the sensors, but also uses Convolutional Neural Networks (CNNs) to match objects detected from 2D images into the room layout, in bounding boxes or squares. Liu et al. (2018) detect opposing corners, connects them with a line, and in this way produce a vectorial representation of a wall. Thereafter, line segments are identified to represent openings (doors and windows), and segmented indoor elements such as tables, sofas, cabinets, and beds are aligned with a rectangular bounding polygon. The Manhattan assumption was followed in this research, which limits the application but has helped them to reconstruct complete 2D floor plans of houses and apartments.

Those advantages of using 2D images to identify objects with CNNs were later used by Apple's API RoomPlan, that allows the reconstruction of 3D models of interiors using an iPhone or iPad. The API leverages fusion of point clouds, depth maps made using point cloud information, and data from movement sensors and cameras, tracking in real time the environment and objects such as furniture contained in it. Instead of merely relying on point clouds, the framework uses images as well to enhance object recognition (that so far works better for 2D images). Reconstructing furniture based on image recognition and classification based on a furniture library (as RoomPlan does) also allows furniture occlusions at the point cloud to be overcome (Pintore et al., 2020). It can also help with problems in point cloud density of thin elements that make their reconstruction difficult, e.g. the legs of a chair often have too few points. The API works simultaneously with point clouds to measure the interior of a building, and images to recognize furniture and indoor objects, resulting in an enhanced detection of building elements. This happens because objects such as shelves or closets block the view of much of a wall, and because they may go up to the ceiling, they confound wall recognition when it is dependent on vertical plane recognition. If objects that can be confused as walls are first recognized for what they are, the identification of the wall behind them becomes more accurate and the heuristic is enhanced. RoomPlan divides a room in up to 512x512 voxels (cuboids) of 3 cm in the x and y axes, by a height of up to 12 voxels of 30 cm in the z direction. In this way, the API can maintain good measurement accuracy up to rooms of 15x15m and 3,6m of height (Apple Inc., 2022).

RoomPlan can reconstruct floors, walls (including diagonal and curved), doors, windows, inclined ceilings (or upper wall boundaries, in any case), stairs, diverse fixtures such as toilets and sinks, and many types of furniture. In their research into the application of the iPhone and iPad's LiDAR sensor into surveying activities, Losè et al. (2022) found out that it can be used for medium-sized areas with satisfactory resolution in the order of a few centimeters and good level of detail. Losè et al. (2022) also point out that the choice of app used for scanning environments makes a large difference in the quality of results, which could indicate different approaches to algorithms like SLAM and different use of movement sensors and noise filtering per application. Even though furniture, kitchen and toilet fixtures, and overall indoor clutter are often not the object of BIM reconstruction, tracking and understanding their location can be useful for security, energy management, evacuation plans (Pintore et al., 2020). It can also help autonomous navigation – being useful for a robot that is updating the available scan of an environment but needs to navigate in a cluttered environment, for instance. Furthermore, some branches of Facility Management (for very large assets, e.g. governments) also concern themselves in tracking and managing how many pieces of furniture, and of which type, every room, every floor, and every building has.

Even though new mobile solutions often have lower resolution and accuracy when compared to static professional scanners, their implementation has already shown some advantages. Fusing information from cameras and LiDAR/ToF sensors, and processing it in real time with the help of ML methods and user input filters and perfects the processes in many fronts, while the point cloud or geometry collected is still being produced. By instructing the user collecting data to point where the first collected wall is, and point to the ceiling and the floor, RoomPlan, one prominent example adopting this technique, helps to validate data collection with a human check early on, and make the process more precise. Pintore et al. (2020), for instance, elaborate on the advantage of early clutter removal (or identification) in the accuracy and stability of a Scan-to-BIM process. The fact that data collection is interactive, and the user can see how the floor plan is turning out in real time, also helps to produce a more accurate model: if the user sees that the model is misidentifying part of the building geometry

or an object, the user can scan that problematic section using more angles to correct the model. Other advantages of RoomPlan are the reconstruction, in its newer version, of curved walls and sloped ceilings (Apple Inc., 2023), which very few methodologies in literature do, and none does both simultaneously. The recognition of objects and furniture also allows the export of a semi-synthetic point cloud, that is based on depth and image sensing, but is completely regularized and has occlusions removed, which can help in the development of point cloud algorithms. One disadvantage of the current API versions is that the geometry generated is surface based, and not based on solid primitive shapes such as CSG. Exports can be done in formats such as .DAE/ COLLADA or .USDZ. Direct interaction with IFC, that is mostly CSG based, becomes therefore more difficult. Some semantic segmentation of the surfaces is available, and has been enhanced in recent updates (Apple Inc., 2023), but the representation of geometry is still far from the IFC standard.

Future of alternatives that overcome the visibility problem of point clouds

One of the biggest limitations of the current Scan-to-BIM methodologies is that point clouds only detect information from visible surfaces. BIM models have much embedded geometry, systems, and information that could not be seen on a finished building without opening up walls and ceilings. Some of the main systems that can often not be surveyed by laser scanners to make as-built models are MEP systems - Mechanical, Electrical and Plumbing components. But technologies have also been developed that help to survey those elements to produce a sound as-built model including, for instance, the electric network of a building, using a wire detector.

Dehbi et al. (2022) proposed a method to optimize data collection when sensing the wire network of walls, solving a Mixed Integer Linear Program (MILP) to find the minimum collection point, based on standards that determine the recommended locations to install electric lines, and the external location of sockets and switches. The order of the measurements, and where to measure next, is outputted by the method until there is no ambiguity in the electric network, using a highly optimized data collection time to produce the as-built model. Those types of methodologies could in the future be integrated into the procedures that generate and update BIM models and now are based on only point clouds. With the addition of surveying of wire position some of the embedded elements, that normally would not be created or updated in a Scan-to-BIM process can also be integrated into the as-is version of the model.

Distinction between as-is and as-built

In recent years, many authors such as Xiong et al. (2013), Macher et al. (2017), Hamledari et al. (2018) and Tran et al. (2019) have started using the terminology *as-is* instead of just *as-built,* for updated BIM models of existing buildings being managed. The distinction can be useful because it signals that the available model is up to date, and is not limited to the state of the building when it was finished by the contractor. Changes like renovations and retrofits can happen after the completion of a building, or, often times, the available as-designed model does not match the as-built state due to changes in situ, or miscommunicated changes in the project by a stakeholder. Areas such as HBIM – Heritage Building Information Modelling – work with models that deviate from the original building, by changes over the years, renovations, or even decay. For all these cases, it is important to keep an updated version of the BIM model available, to dispose of reliable information.

Methodologies concerned with the update of BIM models

Hamledari et al. (2018) proposed a methodology and an algorithm to update IFC models based on building site inspection. Using python and its IfcOpenShell library they updated properties and element types, and annotated information on the condition of a given building element into an IFC file. This methodology could start from data captured by imagining technologies and subsequent image based-detection of element type, or selection of type by user input, where the type and characteristics of a building element are inserted by the inspector into the algorithm. The method shows the power of imaging technologies and programming in automating the update of BIM models, but depends on manual (or external) matching of inspected objects to an object in the IFC model, and can only replace an element type by another element type that is already in the model (e.g. one type of light fixture for another one). The geometry updates are limited to a Shape Representation update to that of a different building element present in the IFC project, but the algorithm is not able to update elements like walls and slabs, as a wall type or slab type might have virtually infinite possibilities of shape representations. The elements replaced by this methodology where only elements for which every element type only has one shape representation, such as a power outlet being replaced by another power outlet, or a door type of given dimensions being replaced by another doo type of the same dimensions. Each door type only has one shape representation, and the methodology is not able to update the dimensions of e.g. the opening that hosts the door. As an alternative for elements that should be updated but their geometry cannot be changed by the algorithm, the algorithm does create property sets to report results of an inspection of the element. **Different IfcOwnerHistory entities are added to report the changed properties and keep track of unchanged properties.**

To overcome the challenge of updating the geometry of BIM models, Rausch and Haas (2021) suggested the use of Dyna-BIMs. Those are dynamic BIM models, that are highly parametrized and can change their shapes and poses to match point cloud elements. The model starts with a "Proto-BIM" model, and making use of genetic algorithm (GA) and simulated annealing (SA), a best fit is found between the BIM model and the built object. A study case was done and demonstrated that the method can be quite useful to update the geometry of pre-fabricated and/or structural elements, in concrete and steel, to obtain an as-built with high accuracy. A comparison is also done between representations of geometry in CSG geometry and NURBS (Non-Uniform Rational B-Spline). CSG geometry is often used in IFC elements, for instance, and can represent most elements with basic geometric primitives such as blocks, spheres, cylinders. NURBS represent surfaces with high flexibility using control points, but demand more processing power due to their complexity (Rausch and Haas, 2021). A conclusion of their research was that when a deviation in the order of centimeters is acceptable, CSG geometry is recommended, but for a millimeter accurate geometry NURBS are the best option**. Another contribution of the research was presenting the idea of having access to changes among several versions of a project, keeping track of the history of a building in a Digital Twin-like framework, in a way that is not accessible through simple Scan-to-BIM.**

Helping the question of matching building elements at the as-is state with building elements from the as-designed BIM model, Collins et al. (2022) developed a method to match building elements from a point cloud to those of a BIM model even with large geometrical or positional differences. This was done using cosine similarity to compare the feature space found for elements in a point cloud, to the features of the BIM elements, in order to allow model update. This was done by formulating a graph for the point cloud elements and a graph for the BIM model, and then enriching them with featurization using GCNs (Graph Convolutional Networks).

2.3 Scan-to-BIM applied to specific building elements

2.3.1 Recognition of Walls

Valero et al. (2021), after extracting the ceilings of the building, which is discussed further in the subtopic about Ceiling Recognition, used the locations of the extremities of the ceiling to find planes that represent walls more accurately. Furthermore, the planes that have another parallel plane in a neighboring room, within a certain threshold distance, and have opposite normal directions, can be filtered to constitute the walls of the building.

One of the most interesting methods currently available for wall reconstruction is that of Bassier and Vergauwen (2020). The method reconstructs many wall types, such as curved walls, that most other methods do not reconstruct, reconstructs many possible types of wall connections, is compatible with multi-storey processing, outputs the generated walls as IfcWallStandardCase entities, and has a Level Of Accuracy (LOA) higher than manual reconstruction. The higher LOA is due to the method being focused on finding the center line of a wall, and reconstructing the wall centered around it, while modelers that reconstruct walls manually tend to pick one face of the wall in the point cloud to align the wall entity, as it is visually hard to estimate the centerline. The source code is made available by the authors.

The method of Bassier and Vergauwen (2020) starts with an unstructured point cloud that is processed into classified meshes using Grasshopper plugins in the software Rhino. Then, the algorithm clusters the geometry to identify walls, determines the wall axes and parameters, and can feed them into Revit using the plugin “Rhino.Inside”, or export the wall parameters from Rhinoceros into OpenBIM tools such as VisualArq and GeometryGym. The transfer of data between plugins and software is done in such a way that the modeler using the tool may manually adjust some wall axes if it is necessary, such as in areas with poor meshing around the floor or ceiling, that affect wall generation.

While the method of Bassier and Vergauwen (2020) brought a very effective framework of wall reconstruction going all the way to export them in the IFC standard, Tang et al. (2022) point out that their methodology of seeking the centerline and expecting walls of continuous thickness can have shortcomings. In the case of columns partially embedded in walls, the column will not be detected and the reconstruction will be incomplete, for which they try to create a solution with their method.

To find the correct wall planes, Tang et al. (2022) check the plane’s proximity to the already detected ceiling, whether or not the plane has a height superior to 1,5m, and an area superior to 3m² (between 1 and 3m² it is considered a column). To merge different planes into the same plane a threshold of 10% of deviation in inclination, and 0,2m in displacement is adopted. After those initial planes are segmented, a 2D binary image is made, with dark points representing areas with points close to ceiling height (walls), and white points the remaining “empty” areas. An expansion algorithm is used to make those lines and points that compose the walls thicker. This 2D map is then used to segment the area of each room, after filtering outlying points, and after using a watershed transformation method to estimate the distance of each pixel to the room’s borders. After that, this space segmentation and the initial grid of extracted planes are merged in a space shape regularization process using a Markov Random Field approach. In this approach, the grid planes from plane segmentation are identified, and

the areas found for each room on the 2D binary map colour the areas that represent each room, helping to eliminate extensions of planes that are not parts of walls. An application of the principle of grid creation by extensions of walls, to help segmenting spaces is seen in Figure 12. Based on the ceiling height then, the walls are reconstructed. Previtali et al. (2018) also uses an angle threshold to detect suitable planes, and the boundaries of the ceilings to find suitable planes for walls, and does extraction of shape primitives by using a RANSAC-based algorithm and a region growing algorithm. Normals of planes are also used in the method just mentioned, and to obtain them a plane is fitted into each point, by analysis of the surrounding points, and the normal is detected as the normal of that plane.

By finding maximum and minimum coordinates along a continuous plane in both x and y local coordinates, Thomson and Boehm (2015) obtain the full extent of a wall, even in areas where occlusions cover the lower or upper parts of a wall's geometry. Spatial reasoning is used to obtain more cohesive geometry. Walls whose length is inferior to 1/3 of the floor-ceiling distance, for instance, are excluded. If a plane is large enough (above a given threshold), but it does not extend from ceiling to floor due to occlusions, this extension can be done automatically, as well as merging into walls close to planes that have parallel normal planes.

Macher et al. (2017) generate walls after finding ceilings, floors, and segmenting rooms. The layout of the ceiling is established using z-coordinate density histograms to find ceiling height, a region growing algorithm to find room area, and MLESAC (Maximum Likelihood Estimation Sample Consensus) to segment the horizontal plane that composes the ceiling. After that, lines are found around this ceiling that should represent the outline of the walls bounding the room, and some filters are used to remove clutter, such as removing planes that do not go above 1,8m in height. After that, vertical planes are sought around the lines that bound the room, with a buffer of 5cm of tolerance and one degree of deviation, and a minimum of points per plane, based on the sampling of the point cloud. Moreover, volumetric walls are obtained finding parallel walls on adjacent rooms, with a maximum deviation of 5 degrees and a maximum thickness of 50cm. After that, the geometry is converted to an .obj file and taken into the software FreeCAD, where IfcWalls and IfcSlabs are generated by selecting elements, resulting in an IFC file.

Jung et al. (2018) start the process of wall reconstruction with ceiling segmentation by z-coordinate density histograms, and after that a small offset below the ceiling is used to encounter the points that compose the layout of the walls. Instead of using a 3D point cloud approach, they identify planes that intersect the ceiling and create wall layout in a x,y plane. Each room is segmented, and different point clouds are made, one for each room, to have more independent and effective processing. Noise in data, such as extra points added outside the scanned area through windows or doors, can hinder the correct identification of room layout, so a threshold of distance around the ceiling is adopted when detecting wall.

Later, Jung et al. (2018) create a 2D bitmap, based on a x and y grid of the floor plan, close to ceiling height, where pixels are classified as occupied or empty (with or without points). To help segmenting the rooms, they use a circular-shaped detection window that outlines an area slightly smaller than that of the actual room. Because the algorithm has to be sure that the room area segmented has no points in it, it goes with a circle around the room's outline and selects all pixels where a circumference can be placed with its center at their center, and no pixel inside the circle's area is occupied by wall points. The radius of this circle has to be larger than the width of all doors, to respect room boundaries, so

the maximum door width has to be manually collected for this process. After the layout of the area of the interior of the room is obtained in the methodology of Jung et al. (2018), the planes forming walls around it are sought, and an algorithm closes the access doors to detect help choosing only the planes inside of the room. To remove clutter and reconstruct walls, it is assumed that most clutter do not assume a vertical plane behavior that follows up to the ceiling. To this extent, clutter is defined as having an offset of a given threshold from the ceiling. Problems with tall cabinets can be a disadvantage of this methodology. To account for areas that have occlusions or openings, a histogram is made above the excluding threshold of clutter, and if points are found in one of the bins, the area is detected as part of a wall.

Because their methodology is based on interior scans for wall reconstruction, Jung et al. (2018) start with interior walls being reconstructed, beginning the process with a line (on floorplan view), for which a close by parallel line is sought. If a parallel line is found, representing the other side of the wall, a wall is reconstructed, and the ends of the initial line are chosen as corners to find the orthogonal walls, and the limits of the wall volume. If no parallel line is found within a reasonable distance, the algorithm detects the wall as an external wall, what is dealt with at the second stage of the process. Later, when detecting exterior walls, a user-defined standard wall thickness is used (0,20m, in this case), as the thickness of the wall cannot be assessed without a point cloud of the building's exterior, registered together with a point cloud of the interior. The technique of studying the layout of the walls in a room, and then reconstructing walls based on parallel lines at the other side, has helped this methodology to deal with walls of varying thicknesses, provided they just change their section thickness but remain rectangular and keep parallel faces.

Nikoohemat et al. (2018) develop a comprehensive approach to wall reconstruction, where not only vertical walls are detected, but slanted walls and sloped ceilings are also detected. Adopting a criteria to differentiate wall and ceiling plans that is based on “almost vertical” for walls and “almost horizontal” for ceilings, it becomes possible to deal with configurations such as those of complex attic spaces where walls are highly sloped and would not be recognized using other methods. The parameters can also be changed by the user if, looking at the visualization, an element of a given inclination was wrongfully classified as wall or ceiling (e.g. a ceiling element is so vertical that it was classified as a wall).

Making use of MLS (Mobile Laser Scanners) for their method, Nikoohemat et al. (2018) also found a way of reducing some of the problems faced with reconstruction in environments full of reflective surfaces. Because MLS systems store which points were collected at each moment in time, it is possible to detect and filter out “ghost” mirrored walls by analyzing the time of collection of their points. This time does not match that of walls around them – mirrored walls were made at the same time as walls from a different region.

The method of Ochmann et al. (2019) takes as input a point cloud that has normals assigned to each point. After that, planes are detected using a RANSAC algorithm. Thereafter, rooms are segmented using an unsupervised clustering approach aided by raycasting analysis. This is done by analyzing mutual visibility between point patches, that is, on top of an area where a plane is identified, a grid-like patch is made to assume that light does not cross that area, and analyzing other patches around it that are visible or not, it is possible to deduce which ones are in the same room (more visible). At the next step, planes are labelled as ceilings or floors (horizontal) or walls (vertical), with heuristics to

eliminate false candidates. Some walls are dilated in their length to merge their end points with other walls and ease the identification of wall connections. After that, all planes are expanded up to the limits of the building area, and intersected in a 3D cell complex. An example similar to this methodology and that found in other research available can be seen in Figure 12. Each cell needs to be classified as outside area, part of a room, or wall segment, which is solved as a cost minimization problem. At last, room and wall volumes are obtained using an integer linear program. The method does not assume the Manhattan model, supports multi-storey processing, and most importantly produces volumetric walls. Even though it does not go all the way to output IFC walls and slabs, it creates geometry that is highly compatible with it, compared to other methodologies. Bassier and Vergauwen (2020) point out, however, that methods based on cell decomposition may struggle with large buildings.

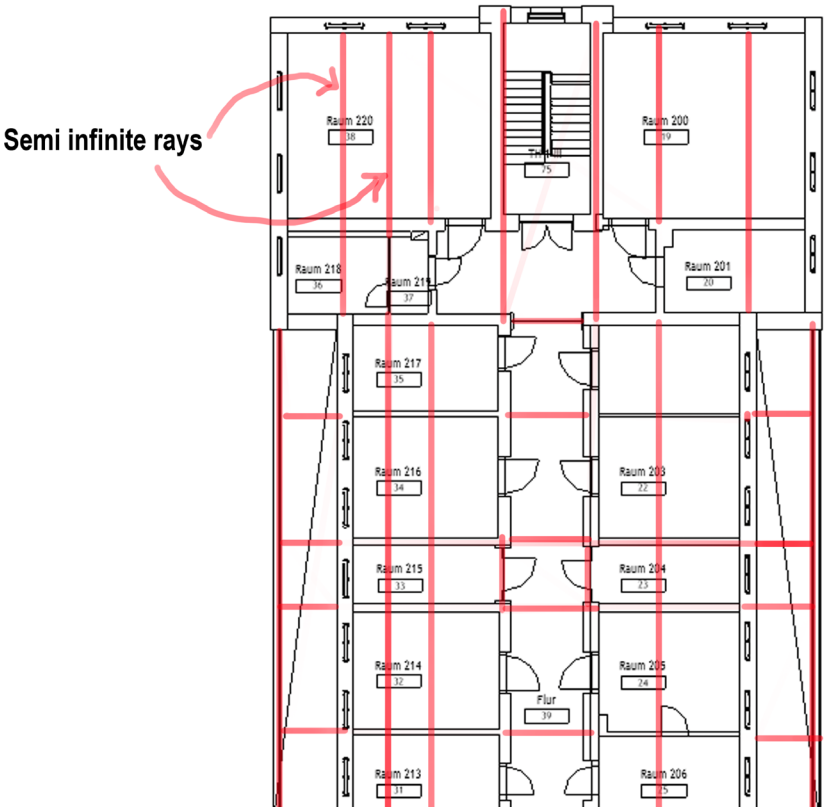

Figure 12: The principle behind semi-infinite rays, strategy used by many authors to connect wall planes that were previously unconnected, making all wall planes have the extent of the building, and trimming them later to compose watertight spaces and later walls

The work of Tran et al. (2019) approached BIM space and wall reconstruction by use of a shape grammar, where a large model is produced based on primitive shapes and grammar rules that govern relations, topology and hierarchy between elements. The method uses the Manhattan-World assumption, and alignment of the main directions of the building, in its point cloud, is done to the x and y axis. This can be automated by finding the normal vectors of the points expected to represent walls, clustering them to find the main axes of the point cloud, and then align those axes to the x and y global axes. This research had a strong emphasis on allowing indoor navigation, so topological relations were prioritized, but besides the spaces and their hierarchies and connections, wall geometry was also reconstructed.

The method of Tran et al. (2019) focuses on being robust against missing or incomplete data, as it reconstructs spaces seeking a cuboid shape of the space, instead of looking for each plane individually. The floors and ceiling heights are found by use of z-coordinate density histograms, but the placement of the cuboids is done between peaks at the x and y-coordinate density histograms. This generates an overly-segmented initial shape, and based on the grammar rules some cuboids will be identified as walls, others as interior spaces, and others as exterior spaces. If points are found at two parallel sides of a cuboid, and they are separated within a certain maximum threshold, the cuboid will be identified as a wall. If no point is found in two close parallel planes, and there is a plane with points in the ceiling, this was an overly segmented area and the spaces on both sides are merged. The geometry generated is based on stored vertices and faces, which are then converted into the Wavefront obj format. The obj file can be opened in the open-source tool FreeCAD, and be used to generate an IFC file.

2.3.2 Recognition of Floors and Ceilings

One technique commonly adopted in the reconstruction of floors and ceilings is to study and parse the z coordinates (vertical) of the points present in the point cloud, analyzing their distribution by means of an histogram. Works such as that of Huber et al. (2011), Khoshelham and Díaz-Vilariño (2014), Oesau et al. (2014), Macher et al. (2017), Jung et al. (2018), Tran et al. (2019), and Valero et al. (2021) use this methodology to derive ceiling and floor positions. The values can be divided in so called “bins”, equally spaced divisions, and the heights that tend to cluster z values will likely be parts of the environment that are horizontal planes or surfaces – many points with about the same vertical coordinate. Figure 13 exemplifies what such an histogram looks like.

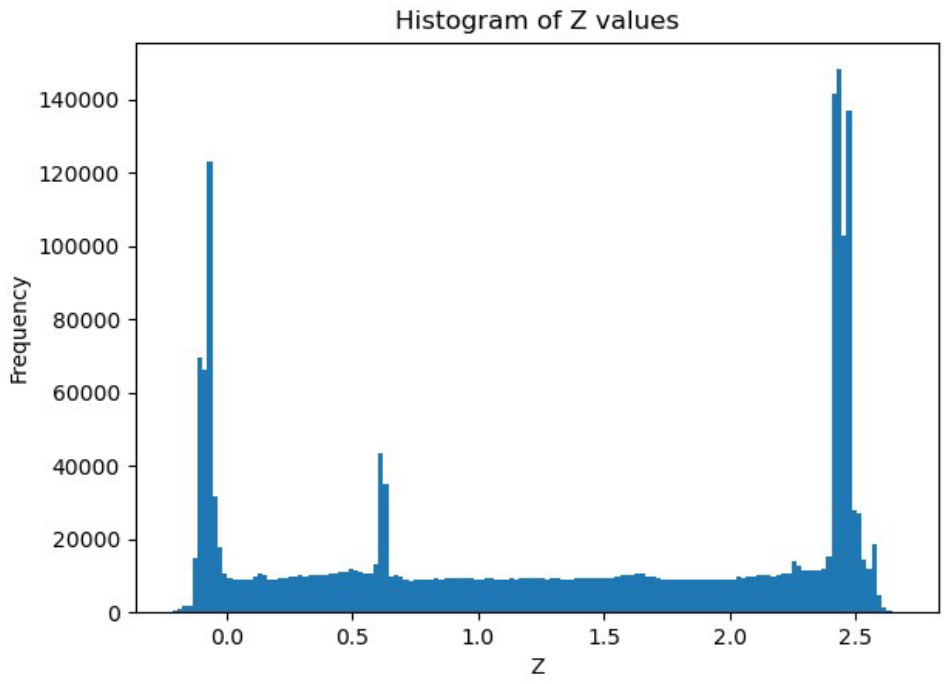

Figure 13: Histogram of z-coordinate values of a room, with the highest peaks indicating floor and ceiling, the peak around the height of 0,7m represents furniture items such as a table, that also have horizontal planes

Even though some noise in the peaks in z values is caused by horizontal surfaces which are not floors or ceilings – such as furniture like tables, that can occupy a decent amount of area, and occlude the

collection of points on the floor in some cases – usually the highest peaks are formed by ceilings and floors. Furthermore, the process of recognition can be aided by algorithms that determine a minimal floor height, maximal distance from the ceiling that belongs to the floor below (for buildings with several floors), or other such conditions that can be checked to make the automated detection more certain and accurate. Once ceilings and floors are segmented, this information can also be used to determine other parameters that can be used to reconstruct other building elements, such as the bounding height of the walls, and done by Thomson and Boehm (2015). One clear downside of this technique is that inclined ceilings (e.g. in an attic) will not be properly reconstructed.

A similar approach is followed by Valero et al. (2021), where first floors are segmented using z-coordinate density histograms, to then later voxelize those points and group them in clusters. Because walls' interiors do not have points, those clusters can delimit the spaces and give semantic information of room boundaries and layout. Tang et al. (2022) used a z-coordinate density histogram to segment ceilings, and applied a threshold to filter planes with less than 4m².

After using z-coordinate densities to find floor and ceiling heights, Macher et al. (2017) segments the point clouds into rooms. This is done by selecting a slice of 30cm of thickness right below the ceiling (as there are less occlusions on top, due to most furniture and clutter being on the floor), and making a 2D binary image of this section as a floor plan. Each pixel that has at least one point in it is assigned as white, and empty pixels are black. Then a region growing algorithm is used to find the area of the room, and the pixels forming a horizontal plane at ceiling and floor levels are selected as ceiling and floor, using a strategy similar to RANSAC, with a tolerance of 2 cm.

Nikoohemat et al. (2018) took a different approach in segmenting ceilings and did not use z-coordinate histograms. The reason for that was the aim to have a method that can detect floors with changing ceiling height, e.g. an Auditorium in a 3-storey building might be an area where there is a double-height ceiling, while the rest of that floor has a lower ceiling. Because the data collection using a mobile sensor is done at approximately a regular height, this height can serve as a reference to determine which floor is being measured, and then merge all spaces with different ceiling heights into the same floor. Their method also allows the detection of sloped ceilings, which most methods cannot do, and uses adjacency relations between detected wall and ceiling elements to distinguish wall and ceiling planes from clutter. Besides the methodology of Nikoohemat et al. (2018), the 2023 version of the RoomPlan API from Apple is also able to create sloped ceilings of complex geometry in real time (Apple Inc., 2023).

When rebuilding surface-based BIM elements based on a surface model, such as point clouds, general building knowledge can help in making interpretations when creating new elements. This is the case, for instance, when the ceiling of a space is separated from the floor level of the next building storey by a distance higher than the thickness of most slabs – then it can be assumed that it is in fact not only a thick slab present, but rather: at the floor level there is the slab, and the ceiling below it is a suspended ceiling (Nagel et al., 2009).

2.3.3 Recognition of Openings

De Geyter et al. (2022) divide the process of reconstruction of BIM elements from point clouds in primary elements (spaces, floors, walls), and secondary elements (walls, doors, and other elements like systems' components). Secondary elements pose an extra challenge in being properly segmented and reconstructed, because their geometry is more complex and they can contain several sub-elements, like door handles and details in bars of a doorframe. On top of that, some of those secondary elements can be dynamic, like a door that can be open or closed, or partially opened. A window can be covered or partially covered by curtains. The lack of points in windows, due to laser beams crossing them, can be considered data in itself, but that makes the assessment of properties such as glass thickness almost impossible. All of this makes it more difficult to accurately describe and recognize those complex elements. Primary elements, on the other hand, often have a simple geometry based on planes, as is the case for walls. One of the possible solutions to problems caused by occlusions is the adoption of mobile scanners, that capture data from more angles (De Geyter et al., 2022).

Once ceilings, floors and walls are reconstructed, Valero et al. (2021) use a hole detection strategy to find empty areas on walls and create bounding boxes around them. Then, based on the size of the bounding boxes, and on the dimensions (and presence) of the bounding box found at the other side of the wall, an opening element is detected and reconstructed. In this way it is also possible to use exterior and interior point clouds together to help detecting some openings such as windows. Quintana et al. (2018) proposed an opening recognition method that uses both the geometry and the RGB color information of a point cloud to detect doors, whether open, semi-open, or closed.

Xiong et al. (2013) and Previtali et al. (2018) proposed approaches to recognize and reconstruct openings based on ray-tracing algorithms. Both methods divide the environment in voxels to discover which ones are being *observed* – if a dot is projected in them, they represent part of a surface –, *empty* – if a beam crosses them, as in an opening such as a window –, or *occluded* – another surface blocked the path of the laser beams so they could not reach the region of the voxel. Figure 14 shows a scheme of how those voxels are classified.

Xiong et al. (2013) starts with this voxel labelling, and after that detects openings, based on a learning method that also helps detecting partially occluded openings to reconstruct their complete layout. After the openings are detected, occluded regions that are not related to the openings are reconstructed using a 3D hole filling algorithm, such as incomplete regions of walls. Whereas Xiong et al. (2013) makes the analysis of one room at a time, and later all rooms are merged, Previtali et al. (2018) aim to reconstruct the complete floor plan at once, using the path of the mobile sensor as an input that helped detecting openings, but under the Manhattan World assumption. Previtali et al. (2018) also used a Hough Transform to help detecting closed doors, classifying door candidates as opened or closed. Openings are considered as doors when they intersect the floor. Díaz-Vilariño et al. (2017) also uses the trajectory of a MLS sensor to detect openings, in specific doors, and recognizes them as the local minima of vertical distances, that is, compared to the ceilings, doors usually represent lower profiles. This detection of doors and use of trajectories also makes it possible for them to segment and recognize rooms.

An example of why detection of openings by analysis of occluded and empty voxels can be effective, is that often times radiators are located right below windows (Previtali et al., 2018). Being really close to the walls, above the floor, and of large rectangular shape, many models could recognize them as windows. But when occlusion analysis is performed, it is discovered that the points at the wall plane, behind the radiator, are absent, due to being covered, and are not actually empty points crossed by the laser beams. This analysis is done by what Previtali et al. (2018) call Occupancy Maps (OM), and the voxel size is set to 1,5 times the size of average point separation, not too small that it contains no points or irregular point presence, and not too large that accuracy would be lost.

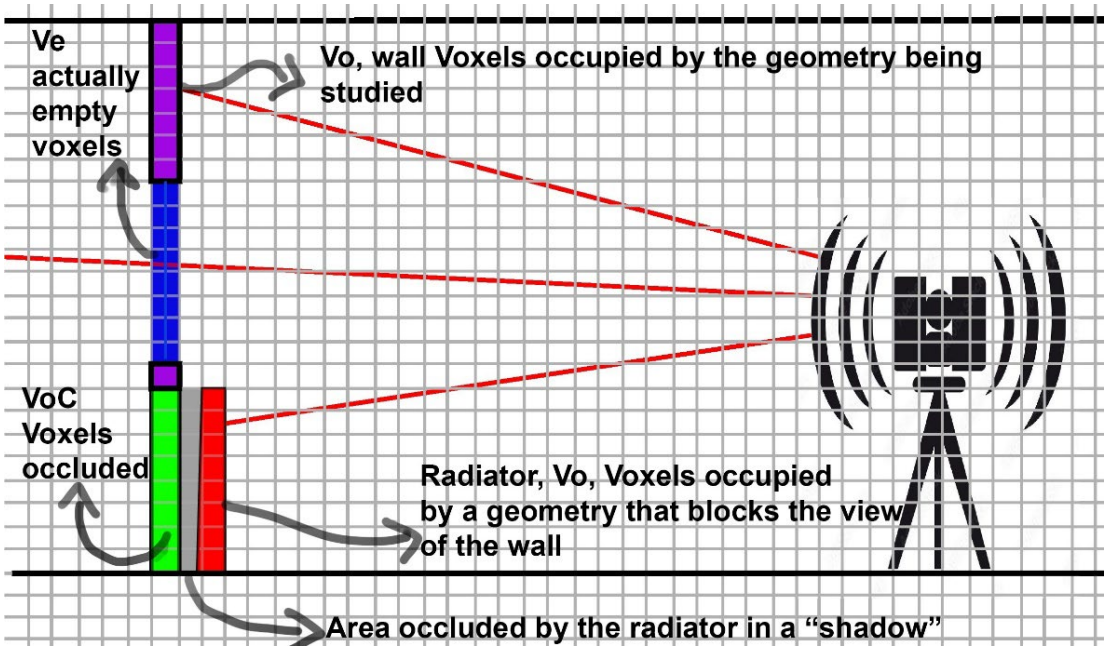

Figure 14: Analysis per voxels similar to that done by Xiong et al (2013). Voxels (volume pixels), or cubic divisions of space, (here seen as a grey grid) are used to detect, by raytracing algorithms, whether a ray is crosses a voxel (Ve), or whether the local lack of points in a plane is due to an actual occlusion (the rays cannot reach the object). Here we have exemplified the section of a wall being scanned, with a radiator under the window. The radiator is represented in red, and it has occupied voxels (Vo), which generates an occlusion behind it (in green). In the green area there are no points collected, and algorithms that look for square openings in walls could think that is a window. The analysis of voxels, that determines that it is empty because the geometry was occluded at time of collection, allows to avoid false positives in e.g. window detection.

Another way of detecting openings was suggested by Macher et al. (2019), which proposes leveraging data from thermal cameras, to merge this data with point cloud data and aid the segmentation of windows and doors. Thermal data is already often collected after a building is completed, or before a renovation, to assess its energy performance and insulation of façade elements. Point clouds are also often collected after finishing a building, but the stakeholders responsible for those two distinct surveys are often not the same, and an opportunity of merging this data to obtain more insights is missed.

For opening detection, Jung et al. (2018) made a binary map of each wall and looked for hollow areas. Use was made of comparing holes in the point cloud's representing walls, and matching openings present on two sides of a wall. If at one side the opening is smaller, the side with a larger opening is considered, as occlusions and noise in data are more likely to reduce than increasing the opening area detected. In comparison with other methodologies that use ray-tracing to detect real openings and distinguish them from occlusions, Jung et al. (2018) did a perimeter to area ratio filtering. Because

occluded areas have a very erratic boundary, they often have a much higher perimeter, which skews their perimeter/area ratio. This is an interesting approach, but one curious drawback was that in one instance, where a door was opened and blocking part of a wall, the occlusion had a regular rectangular shape, and two doors were then detected. This points to the importance of scanning from as many angles as possible, which is made easier with mobile scanning. As a last step, the real openings filtered are then refined using Douglas-Peuker segmentation and constrained least-squares regularization, to simplify and regularize the lines.

With a terrestrial laser scanner (TLS) and a thermal camera that captures simultaneously infrared and RGB images, Macher et al. (2019) fused RGB, thermal, and point cloud-based geometry into a textured mesh, for indoor environments. Some challenges found were that a number of manual or complex steps were required, for instance: the IR thermal camera had a low resolution, and the color sensor of the device produces images that span a larger area than the thermal sensor images, so the matching of the IR heatmap image has to be done on top of the color image. On top of that, the image center is not fully aligned between both pictures, as there is a displacement between both sensors. A further problem is that the color scale can change from an image can change from one image to the next image collected, so e.g. 25°C can have one color in one image and another color in another image. Another step to fix those colors to single values across multiple images is thus necessary. Several targets are placed at the room of collection to make matching the TLS point cloud and the thermal data. Results at this research were limited by the mentioned factors and current quality of available sensors, but the application of thermal data in opening detection shows itself to be promising. A further advantage of the method could be help to recognize doors in both states, as, in many cases, both an opened and a closed door let more heat go through than a wall.

Most methodologies focus on merely trying to identify the locations of objects that could be classified as a door or as a window. One of the most time-consuming activities of reconstructing openings (using up to 60% of modelling time), however, is finding the adequate BIM objects in object libraries, that can represent the best each window and each door (De Geyter et al., 2022). To solve this, De Geyter et al. (2022) developed an approach to cluster together similar windows and similar doors in a point cloud, using machine learning, and then assign each cluster to an object type. Because most openings are repeated, the majority of the openings in a building can be automated, and the few unique instances are done manually.

In the study area of opening recognition, De Geyter et al. (2022) pointed the current limitations of grammar-based approaches, that still often need rules adapted to each building in order to be effective enough. Other possible approaches are pattern recognition (e.g. grid-like structures) and ML models. As Section 2.2.3 explains, their literature review shows that multi-view image representations are computationally more efficient than 3D representations due to how robust current 2D Deep Learning networks are. To detect the object types of windows, they first suggest clustering them into similar windows by images, and then crop a point cloud around each window. After that, a histogram is made of the geometries of the windows on the 3 axis and compared to the same histogram of the BIM model of the window. Some problems matching are still found due to clutter and noise, and the fact that some faces of a window frame are encased within the wall, which makes comparison harder, but the technique is promising.

2.4 Applications of Scan data for Facility Management

Volk et al. (2014), Macher et al. (2017) and Salazar et al. (2019) point out that the application of BIM principles is still far from its potential at the maintenance, deconstruction or renovation of buildings. The use of BIM presents however great opportunities for areas such as maintenance, quality control and space management (Juan & Hsing, 2017; Macher et al. 2017), and can help to significantly reduce the elevated annual costs incurred by poor interoperability in Facility Management (Becerik-Gerber et al., 2011). Furthermore, in the last years, we notice that, as BIM becomes more consolidated in the market, companies are starting to leverage the advantage of openBIM tools for management of real state and facilities as well. One of the big advantages of using openBIM is that the standardization and interoperability of the formats used guarantees that data stays accessible and reliable over the lifespan of a building, independent of the specific tools used whose type might change over time.

Werbrouck et al. (2020) say that the fine-tuning of openBIM helps to streamline the construction process and the design of new buildings as effectively as possible between all stakeholders. However, this almost exclusive emphasis on the design of new buildings makes it more difficult to integrate the generated data to other data from sensors, GIS, FM, historical data, and multiple geometrical representations of the building and its elements, such as point clouds or meshes. Those alternative representations can be useful to keep data about the most recent condition of a building (Werbrouck et al., 2020).

Becerik-Gerber et al. already pointed in 2011 to the importance of using a BIM-complying database to make FM more efficient, solving problems in space allocation and management, maintenance, commissioning, visualization and marketing, quality control and, very importantly, real-time data access, both numerical and for locating building elements. Werbrouck et al. (2020), in turn, proposed a framework for Scan-to-Graph, where a robust linked-data database can be established to connect various representations of the building to the IFC semantic data that is transformed into RDF. Based on this approach, data generated by 3D scans can be used to assess up-to-date geometry connected to building elements, and can be queried by building managers, increasing reliability of data in the FM and real estate management.

Harv et al. (2022) and Kloet et al. (2022) discuss how the use of a Common Data Environment (CDE) is essential in the good functioning of Building Management Systems (BMS), as they help connecting and managing all documentation, models and data of projects. If such a CDE is well applied, different tools can work and exchange information in an optimized way that avoids duplicated data and enhances collaboration between the involved parties. With the use of open standards such as openBIM the data exchange becomes vendor neutral and reliable. To take the most out of the application of BIM into Building Management Systems, it is ideal if the facility manager and the team responsible for system integration are already involved at the conception stage of the building (Harv et al., 2022; Kloet et al., 2022). Harv et al. (2022) and Kloet et al. (2022) also point out the essential role that CDEs play in coordinating the linking of the data between the BMS and the model or 3D data in a Digital Twin (DT) perspective, where information is collected in real time using sensors from a building. The CDE can thus serve as the architecture in which the different real time data of a building, as sensors in HVAC systems that aids maintenance, or current geometry of the building in the form of point clouds, is brought together and tracked whenever necessary.

Required ranges of point cloud and model accuracy for different applications

For different purposes of data usage, different accuracy of point cloud data is necessary, with applications such as historical building detail reconstruction requiring a higher accuracy, and management, cleaning and space planning requiring a relatively lower degree of accuracy (Thomson, 2016; Macher et al., 2017). Thomson (2016) gives the required range of accuracy for Facility Management as 3 to 10 cm, for Space Planning (often part of FM) as 5 to 20 cm, and architectural surveying and as-built documentation as 2 to 5 cm. Historic Detail reconstruction, on the other hand, would fall within 0,1 to 1 cm of accuracy. Some of those ranges are estimated based on how accurate scaled models should be on paper, however, so, evidently, it is adequate that computer geometric models should follow those ranges of accuracies or better ones.

Importance of data accuracy in practice

Salazar et al. (2019) point out the importance of real-time and reliable information about the asset inventory and condition, in order to achieve success and efficiency in Facility Management. Reliable information is necessary, but information comes at a price. Therefore, methodologies that make the acquisition of up-to-date and accurate information faster and cheaper are promising and desirable for the market. Another consequence of information acquisition, handling and storage coming at a price, is that it is important to know and focus on which information is relevant to the desired application, and not necessarily obtain information for its own sake. Therefore it is essential to have Information Requirements and Exchange Requirements that make clear from the beginning what data is necessary for the execution and management of a building.

One problem that has occurred historically is that sometimes designers and contractors would not transmit appropriately to facility managers the accurate information that they need at the maintenance stage. To aid operators and managers in bridging this gap, the American National Institute of Building Sciences made a manual called National Building Information Modelling Guide for Owners, which helps facility managers in setting their goals and requirements for internal management, or when dealing with other stakeholders.

Having up-to-date models and reliable information can make a large difference in FM costs, and speed up many processes. At a conversation with the BIM Director of the real state department of the Erasmus Medical Centre (EMC) in Rotterdam, where part of the data of this report was collected, it was emphasized how important this reliability of information can be: the larger the assets managed, the more important quick and precise information becomes for decision taking. Space management becomes an important task in such cases of large real estate management, such as hospitals and large offices. Buildings are large and expensive, and change of use of an area is often necessary, which sometimes asks for renovations in the building, but sometimes only implies in repurposing existing areas. Having all this information in BIM and up to date makes the process a lot better: if for example it is known that a given amount of square meters is needed for a new activity, a quick check can be made among selected candidate areas, which would be much more laborious with physical files and hard copies.

Knowing with reliability and accuracy the areas of each division of a building is also very important for contracts with third parties such as cleaning contracts, which make a substantial part of management costs. One way of surveying this data and uploading models is by scanning the relevant

area and producing a point cloud of it. If information is inaccurate the payer may have much more by an area they did not knew they had, or will not be able to prove their area is in fact smaller. Another use of BIM for FM pointed out by the EMC is that, knowing with high reliability all building elements containing in a building, it is possible to make an inventory of materials when the building is demolished (disassembled) to reuse them in a circular way when constructing another building. This was done by EMC, which donated materials of a demolished building to compose most of the building materials of a company's new office in Overijssel (The Netherlands). The BIM management team of EMC also started incorporating point cloud collections in their facility management activities, like keeping the inventory of building elements just mentioned, helping in the update of as-built BIM models, and measuring deviations of BIM models and real condition of the building. The point clouds were collected with an iPad's LiDAR scanner, using the app Polycam.

2.5 Conclusions

The literature starts with an introduction to point clouds, how they can be collected, how their files are structured, and how they are currently the most reliable method for collection of geometry of objects in the real world. Some problems found in point cloud collection such as occlusions and challenges with reflexive surfaces are also outlined. The main algorithms and methods to register point clouds such as ICP and SLAM are explained, as well as the differences between point clouds collected in fixed collection points (TLS) or moving sensors, for which SLAM is often used.

The new advances in solid-state LiDAR production are briefly discussed, which make the scalability and drastic decrease of costs in surveying sensors possible. Some applications of smartphones into the collection of point clouds for interiors and the state-of-the-art of this type of use is outlined. After that, the application of point clouds into the validation, generation and update of BIM models is discussed. Scan-vs-BIM is described as a methodology that compares an existing BIM model with the condition found on site captured by a point cloud. This is often used in tracking construction progress, where the comparison is often done under a simple volumetric check paradigm – if a volume of points is found in an area that, in the BIM model, should be built, that area is checked as built. Scan-vs-BIM can however be used in the update of models as well, where discrepancies of an outdated as-designed BIM model are identified and the building can be updated into its as-is state. Some of the first authors to use Scan-vs-BIM for the update of outdated models were Bosché et al. (2015), that coupled a check of proximity of elements with the recognition and reconstruction of elements (the latter more common in Scan-to-BIM). The check was performed to detect elements that were not present at the as-designed project, e.g. a new pipe system, and make them ready to be updated into the model, and as well as tracking element deviations. The accuracy of model update was largely improved by adding the recognition based on heuristics and element geometry of specific element types, usually found in Scan-to-BIM, compared to a simple Scan-vs-BIM volumetric comparison.

Scan-to-BIM is the methodology that creates building models or building elements based on point cloud geometry. This is often done by segmenting a point cloud and labelling groups of points, assigning building element types into those groups, and then creating BIM elements from each of those labeled point groups. Even though much research has already been done in this area, there is still some lack of uniformity of methods used, and many of them represent a method of segmentation that merely reconstructs labelled surfaces, instead of volumetric solids, which are often used in

practice for BIM models. Even less research produces results that are already compliant with commercial applications or openBIM model standards such as IFC, the most popular vendor neutral schema for building data. A few researchers however obtained outstanding results, such as the work of Bassier and Vergauwen (2020), where point clouds are segmented and output IfcWallStandardCase entities, handling many types of wall arrangements and connections, with a level of accuracy better than manual reconstruction.

Many heuristics, previously known standards of how building elements are supposed to work and be arranged, are used in the process of recognizing walls, floors, ceilings, openings, and other elements in point cloud, and creating their representation in a building model such as an IFC file. For wall recognition a layout is often based of rooms, using the ceiling, for instance, and wall planes are expected to be are the boundaries of the room. Heuristics are used to eliminate false candidates, such as small planes or planes that do not start from the ground or do not go up until the ceiling. Ceilings and floors are often segmented and recognized as peaks in values for the z-coordinates of a building, in a z-coordinate histogram, and some heuristics such as knowledge of the minimum height of a ceiling can be used to eliminate false candidates. Openings are often tracked by the movement of a sensor that collects point clouds, as it needs to cross doors. They can also be collected by ray-tracing algorithms, that track if a “hole” in a point cloud plane, that could be a door or window, is really a place where the laser beams just crossed, or if another object covered the vision of the sensor and the area is just occluded. Another method is using thermal-camera data coupled with point cloud data to find openings, that likely leak out more heat than walls.

Some other works relevant in the update of models were further discussed. One of them is Hamledari et al. (2018), that updated IFC models using IfcOpenShell with a few manual interventions, for some elements like door types and electricity fixtures. Rausch and Haas (2021) proposed the idea of Dyna-BIMs, BIM models that are adaptable and can have their parameters adjusted to fit data from a point cloud using a genetic algorithm and simulated annealing. In this way a BIM model could be produced based on building data, and another interesting suggestion proposed by them was having access to changes among several versions of a project, keeping track of a building’s history in a Digital Twin-like framework. Collins et al. (2022) used Graph Convolutional Networks to compare data from an IFC model to data from a point cloud, using cosine similarity. In their work, elements could be matched based on their relation to other elements around them, geometry, and element type, allowing matching elements without depending on thresholds and proximity. This can be very handy for the update of elements for which a match using thresholds is difficult.

The literature review closes with a quick review of Facility Management principles, the importance of reliability of data and how important it is to have good communication from the start between designers, contractors and building managers. One important thing in having accurate access to trustworthy data is having a Common Data Environment, where all tools used interact and duplication of data is avoided. The use of point clouds in inventorying areas available for Space Allocation, and quantifying materials and building elements for renovations and building disassembly is further elaborated.

3 Methodology

3.1 Introduction

This Chapter will discuss the methods that underpin the developed procedure, that updates IFC files based on segmented point cloud geometry. Section 3.2 describes materials and methods used to collect point cloud data, and file formats that are used in the proof-of-concept. After that, the overlay of point cloud and IFC file geometry is discussed, and how this could be done manually and automatically. After that, the standard used for defining the extents of a wall in the IFC file that will be updated, and the standard used to define the extents of a point cloud wall in the segmentation process are described. The segmentation of ceilings and columns is also discussed. After that, a discussion is made of how the procedure developed can work in different data storage paradigms.

Section 3.3 discusses the principle used in comparing IFC and point cloud walls, where walls are compared one by one across point cloud and IFC model, by having its starting and end coordinate matched within a given threshold of tolerance. IFC walls that find no match are deleted, and point cloud walls that find no match create a new IfcWallStandardCase instance. The Section goes further to describe how the geometry created based on the segmented point cloud wall can be improved to match the pre-existing layout of the other IFC walls around each new wall. Section 3.4 discusses the principles used for the update of the height of ceilings, based on matching an IFC ceiling with the point cloud segmented ceiling, and setting the height of the IFC ceiling as the same height of the point cloud ceiling. Thereafter, Section 3.5 describes how the algorithms for the update of columns are conceived, as well as how to approach problems common in building elements that may or may not be embedded in other elements, such as columns embedded in a wall. Section 3.6 explains the basic characteristics of the user interface developed in the proof-of-concept that help visualizing and understanding the update process, as well as structuring the steps taken in updating IFC files. Section 3.7 describes Room Mode, that is a special case of check and update that can deal with the case where only part of the building is scanned. Because every IFC wall that does not find a match to a point cloud wall would normally be deleted, a local check with the scope of the scanned area needs to be performed when an area smaller than the entire IFC project is scanned. The building elements chosen to have their update demonstrated are walls, columns and ceilings, with a focus on walls. Walls and ceilings, bounding a room, are some of the most basic and researched components of a building, in Scan-to-BIM research. Columns are also a very interesting building element in what pertains their update, as they may or may not be embedded in walls and have their geometry accessible to LiDAR scanners, so methodologies to handle this complexity and uncertainty are necessary, on the way to automating IFC generation and update based on sensing technologies.

An overview of the processes involved in the procedure is shown in Figure 15, with the steps covered in this research highlighted. The present methodology assumes a model where the point cloud is registered and aligned in the same reference system of coordinates that the IFC model is, and also assumes a previous segmentation of the point cloud into elements such as columns, ceilings, walls, doors, etc. Those previous steps can be automated, allowing for a fully automated framework, but when developing the tests for this research, point cloud segmentation and registration of the point cloud into the IFC reference system were done manually. That is due to the fact that, despite research in the area being promising, those other steps were already approached by other researchers, and

however good their methods might be, they may not be fully accurate. Therefore, to ensure correct ground truth input to update the IFC models, those steps are performed manually. As a reference to automated registration of a point cloud into a building in an IFC file, the reader is directed to the work of Sheik, Veelaert and Deruyter (2022). Methods to semantically segment point clouds were discussed in the Scan-to-BIM Section (2.2.2).

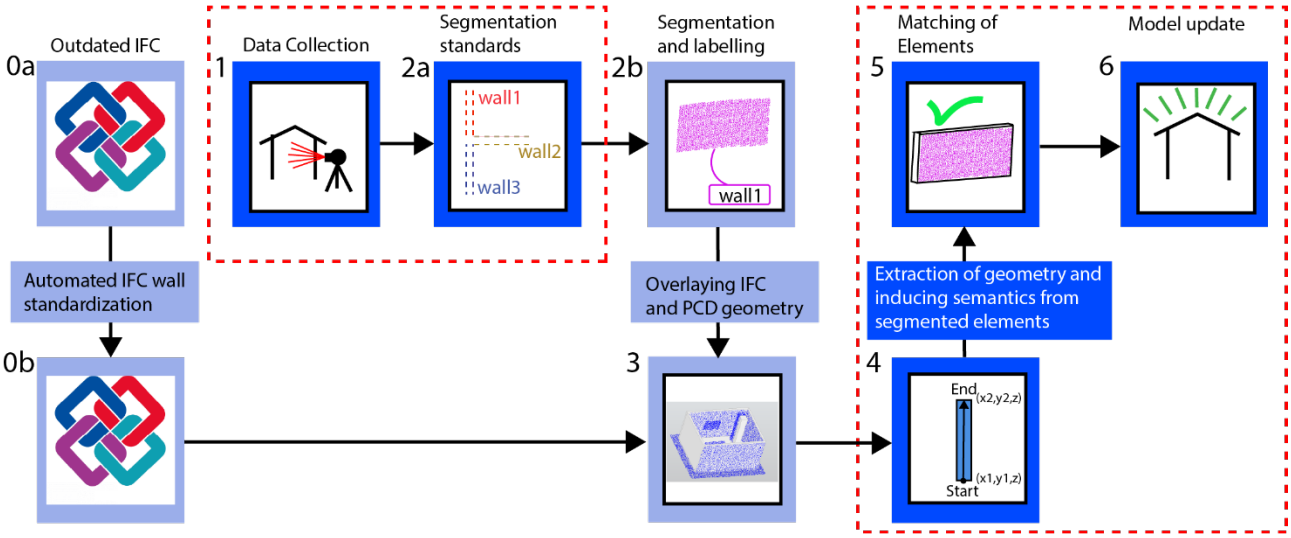

Figure 15: Overview of the IFC updating process. The steps highlighted are the focus of the thesis, specially steps 4 to 6, that have their automation researched. The topics in a lighter shade of blue were performed manually, and their automation is covered by current research from other authors (except 0b)

The **research questions** that drive the development of the procedure ask for a procedure that updates IFC files based on geometry extracted from elements segmented from point clouds, automatically, and allows the scan of only a part of a building to be used for a local check. The focus of the research is the update of IfcWallStandardCase elements, and IFC walls can be deleted from the model or added into it to obtain the as-is layout. For newly created walls, a previously existing IFC wall is used as a template to add semantics to the new wall, and after the basic geometry of the new wall is created based on the point cloud geometry, a geometry refinement algorithm is used to harmonize the configuration of the wall to its surroundings. It is also demonstrated how the update of ceiling heights can be done based on point cloud data, and the update of columns can be done (creation, deletion or threshold-based update).

The procedure developed is focused on the later stage of the model update process. It deals, for instance, with data requirements, as how the IFC file should be set up for optimal update, how the point clouds should be segmented – e.g. what is considered a wall in the process of segmentation and in IFC, both at the schema level and at the modelling level, so a congruent comparison can be established. Following this, the geometry extracted from a point cloud will permit obtaining the necessary information to create, delete and adapt IFC walls, but *if* the IFC model is also modelled properly, according to requirements, it can be guaranteed that a future update will be performed more effectively.

Even though part of the focus of this research is making point clouds collected using mobile devices (phones or tablets) useful in the update of IFC files, the methodology should be equally applicable to point clouds collected using TLS systems, and possibly even work better. In fact, some experiments

to validate the process developed shall be done using data collected from terrestrial laser scanning. That is because currently mobile devices have a limited range of the area they can cover with high quality and low drift of points. To perform initial tests of a tool, a small environment with basic variables is ideal, so a mobile device is ideal here. To validate the procedure developed on a more complex context, however, it is interesting to test how it performs on a larger model and larger scan. Figure 16 shows a guide into the tests created to validate the relevant updating operations based on point cloud geometry. The datasets used are further discussed in Section 4.1.2. Due to the limited accuracy of point clouds originated from mobile devices, the main focus of the procedure is updating an outdated layout of building elements, and not updating millimeter differences in length or small displacements of building elements. To further aid and enhance the application of the procedure developed the comparison of as-designed and as-is will be done in such a way that a complete scan of the building can be used, but a partial scan of an area of interest can also be used to check and update only that area. This is especially interesting for the application of mobile devices in LiDAR scanning, as they have better performance in smaller areas. Furthermore, not having to scan an entire building if it is not necessary, if a renovation performed is only local, is much more realistic and saves time and money. This process of local updates is further discussed in the Room Mode Section (3.7) and in the Results Section.

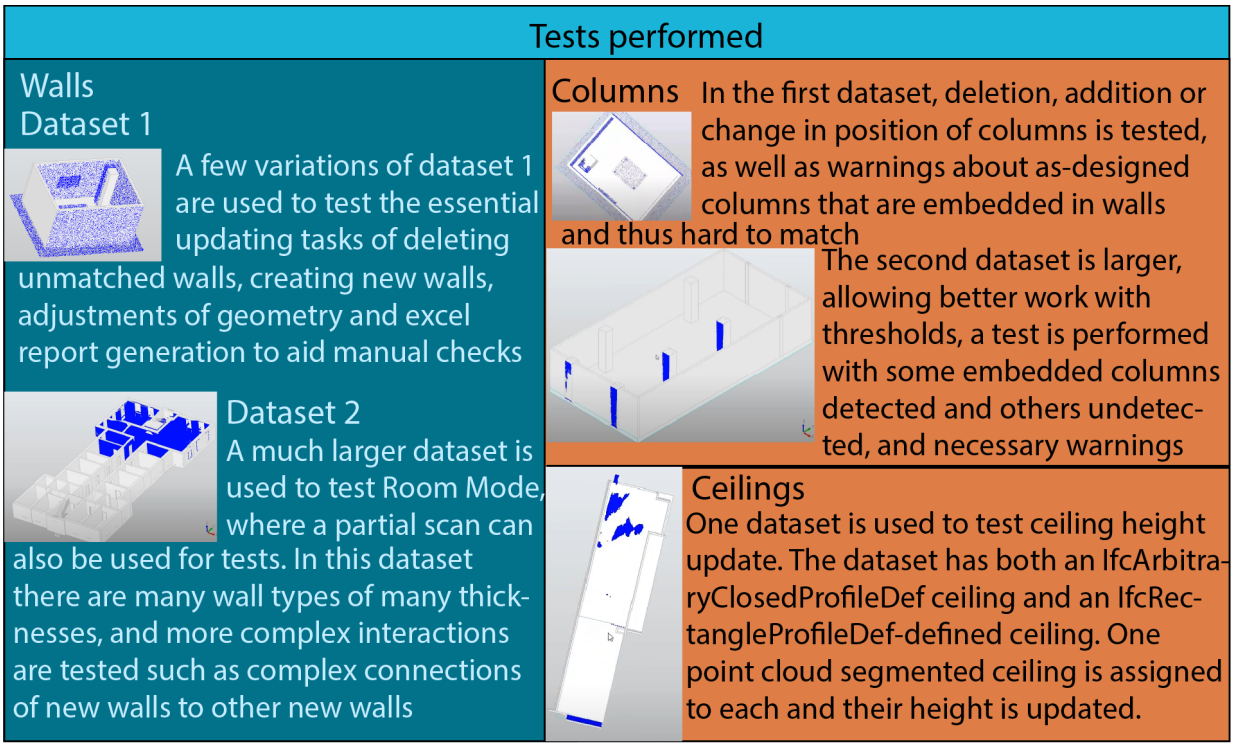

Figure 16: Guide to tests performed in the research. The tests were chosen as tests that can validate the main operations involved in update: matching, deletion, creation and translation of elements.

The decision was taken to keep each segmented building element in a different point cloud file, and have the label (e.g. ‘wall 3’, or ‘column 5’) as either the name of the file, or keep the files in a folder so that it is known that items in that folder are walls or columns or something else. This follows a similar standard to the popular S3DIS dataset, with the difference that elements such as walls are volumetric (both sides of the wall), which is more suited to the creation of IFC walls.

Once the elements are labelled, their points can be analysed to find their geometric properties, and depending on which element they represent a specific set of mathematical operations will be

performed to find the properties necessary to compare against or create an existing element in IFC. For instance, a point cloud wall can have its start and end point extracted, length, height, on which floor it is located, and knowing its general position it can be inferred which connections exist to other elements in the IFC file such as walls connected to it. Depending on the orientation of the wall relative to the x and y axes, different operations are performed to find its geometric properties. The choice of IFC as a standard for BIM model format in the updates performed in this research was made due to its popularity on the market, and open standards, allowing collaboration and preservation of data in a vendor-neutral paradigm. The code developed and tested for the procedure will be based on a Manhattan world assumption, that is, walls aligned to the x or y axes and perpendicular or parallel to each other. Many of the principles discussed and developed should however be applicable into the update of diagonal walls as well, and an overview of how this can be done will be discussed in Section 3.3.1. Walls are discussed more extensively, as they are the most reoccurring theme in the literature, one of the most important constituents of a building, and it is relatively complex to have their layout defined. For instance, a column can be there or not there, in a different position (but be the same column), or have a different profile. A wall can have multiple connections to other walls, helps defining a space on a project, can have different heights, openings, contain elements such as windows and doors, which adds much complexity when dealing with walls in IFC. Nevertheless, the update and check of other elements, i.e. columns and ceilings will also be discussed, developed and tested.

The procedure's proof of concept will be developed based on python code, which is considered one of the most accessible programming languages. However, making use of the tool requires some programming knowledge, even with vast annotations in the source code, which prompted the development of a user interface to make the procedure available for a wider range of stakeholders. For advanced or specialized users, the code remains available to change parameters, thresholds or adapt a few things, but for other users or routine applications a user interface can help, aid visualization and save time. The interface is developed in python using the pythonOCC viewer and the PyQt5 and matplotlib libraries. Due to the lack of good IFC visualizers in python, the interface uses OpenCascade (pythonOCC) to visualize a .STEP version of the ifc file. In the background, the operations are performed directly in the IFC file using IfcOpenShell, and IfcConvert updates the new statuses of the IFC file into STEP to aid visualization in the interface. The interface also allows the visualization of the complete point cloud or segmented elements in point cloud format, overlaid over the building geometry, which gives visual cues of the as-is geometry and possible discrepancies. In Room Mode, the hull that encompasses the scanned area can also be visualized, to be sure that the right area is being checked and to know if different parameters should be used. The interface is described in further detail at Section 3.6. Figure 17 shows the use case diagram of the backend procedure and the interface.

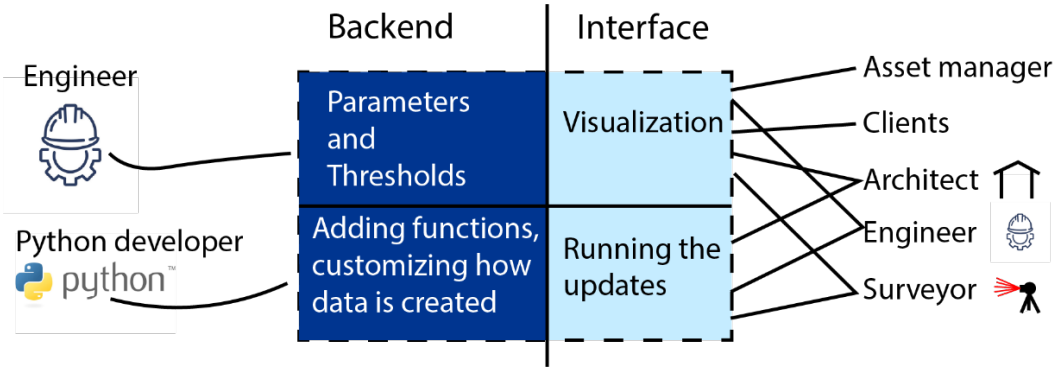

Figure 17: Use case diagram of the procedure by different parties

Both Scan-vs-BIM and Scan-to-BIM paradigms are used in the development of this procedure. Scan-vs-BIM is performed by comparing positions of point cloud and IFC elements, their existence within a space, checking which elements were changed (placement, size of column or wall, ceiling heights, etc.) and making automated reports about deviations of elements. After the comparison, comes a Scan-to-BIM stage where a new IFC file is generated based on scan information, with the updated status of the layout of indoor building elements, and maintained semantics (as long as possible) from the original model. Therefore, the way of matching the data will often differ per element type, but the general structure of this process is conceived in the following steps (steps 4 to 6 from Figure 17):

1. Extraction of geometry information of the building elements scanned and segmented, and some semantics by heuristics, such as inferring a wall type from wall thickness and building's region;
2. Using the extracted information to try to match those point cloud elements to the as-designed IFC elements;
3. Delete IFC building elements that were not matched;
4. Create a new IFC element for each point cloud element that did not find a match
5. For ceilings and columns, if elements are matched – within a threshold, but have a deviation in position, update the position of the element in IFC to match that of the point cloud

3.2 Data capturing and adjustment

3.2.1 Equipment and method used for point cloud collection

The procedure needs two types of data to be started: an IFC file of a building, considered as the as-designed version, and a point cloud collected at the building representing its current state. The IFC file should preferably follow some standards in its design that will be discussed later in Section 3.2.3. The point cloud used for the envisioned procedure is an unstructured point cloud in an ASCII text file, where every individual line represents a point, and in each line the following information is contained about the point, in order: coordinate x, coordinate y, coordinate z, and optionally Red value (ranging from 0 to 255), Green value (ranging from 0 to 255) and Blue value (ranging from 0 to 255). Always the first three values are the x, y and z coordinate values in *meters*. The tool works in the same way for files with or without RGB information. The values are separated by spaces. The files used are in the format .xyz or .txt, where there is no header at the beginning of the file and the only information contained therein are the points in the format mentioned. However, many other formats follow the mentioned pattern, but the point information is preceded by some type of header in the text file, with information on the count of points present, file authorship, application and equipment used to produce the point cloud, etc. Those file formats, such as *.pts* and *.pcd (those with ASCII characters)* can also be used by implementing very minor adaptations to the current code of this methodology, namely to skip the header. Ideally the point cloud should only contain x,y,z coordinates or x,y,z coordinates plus RGB information, and a simple script was developed to remove extra information if it is present, found in Appendix IX.

The device used to capture those point clouds will be an Apple iPhone 14 Pro, that is capable of producing colorized point clouds, on a mobile scanning framework, based on its movement sensors, colour cameras, and solid-state LiDAR sensor. Reasons for taking this decision are: first, the decision for LiDAR is taken due to the superior performance of laser scanning in indoor environments, compared to photogrammetry. LiDAR based point clouds are also already scaled in meters, and often

dispense the need for targets and re-scaling. Mobile point collection was chosen instead of a static laser scanner due to the lower rate of occlusions. A mobile smartphone-based point cloud collection was chosen due to its accessibility to a wider audience at relatively lower prices, and to explore the potential of this continuously evolving market in its possible use to the AEC industry. The phone model was chosen due to the advanced APIs and hardware available by Apple, compared to those of Android/Google at the time of the research.

The specifications of the camera of the device used for point cloud collection are a 12MP 13mm ultrawide sensor, with an aperture of f/2.2, and a field of view of 120˚. The ultrawide camera is chosen among the other camera sensors because its wide angle allows to collect more points per frame – the colours of the camera sensor are fused to the depth maps or points from the LiDAR sensor, and both have the same field of view. The camera projects colour information into each point from the LiDAR sensor, coded in RGB. The LiDAR sensor present on Apple devices works projecting a pattern of points onto surfaces in front of it. As it is a solid state sensor placed at the back of the device, it does not rotate from its point of reference to scan the entire environment, rather the user has to point the device to all areas that the user wants to capture, as the user walks. Geometry up to 5m away from the sensor can be captured, or, if using RoomPlan API based apps, a room height of up to 3,6 meters, as explained in Section 2.2.3. Figure 18 shows the pattern of dots projected by the LiDAR sensor.

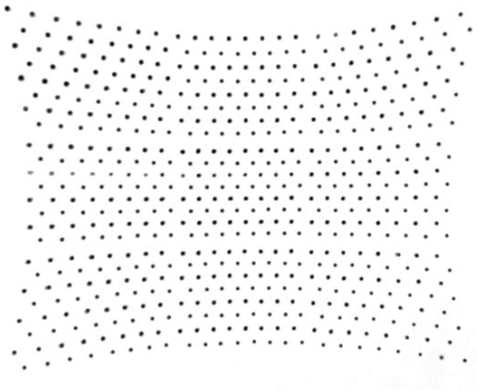

Figure 18: pattern of points with 8x8 point segments divided in 9 sectors, identified by Losè et al. 2022

There are two main APIs made available by Apple that can help the collection of point clouds for building interiors: ARKit and RoomPlan. ARKit, in the context of apps that collect point clouds of buildings, is mainly used to collect raw point clouds, while RoomPlan collects a 3D model of a building's interior with the help of user input and fusion of sensors and machine learning models to map a building as the user RoomPlan also allows the user to extract point clouds, and the interesting thing about those point clouds is that they retain the corrections and filtering done by the machine learning models used to reconstruct the interior. The planes (made by points) that constitute walls or floors are more plane, orthogonal to each other, continuous, and suffer less from drift compared to point clouds generated without RoomPlan. Therefore, the result has less occlusions, better precision of measurements, and much less problems with drift. Point clouds generated without the RoomPlan API, for instance, can "lose track" of how to best overlap all the points collected at each frame, and start having different plane segments at different heights trying to represent the floor, for a larger room, whereas in RoomPlan the floor plane is mostly continuous.

Two types of point clouds can be extracted using the RoomPlan API: the first one is a point cloud based on the generated 3D model of the building, that resembles more a synthetic point cloud and does not have a ceiling, but the walls and floors are perfectly regular with no occlusions. This can be useful for algorithms that struggle with occlusions, as even walls with much furniture covering part of them will still be fully represented with no occlusions. It is also possible to hide furniture in the app, to select more easily the points of a wall that has furniture right beside it, for instance. Furniture items can also be present in this point cloud, but they are based on recognized furniture items, and not necessarily the real furniture of the building. For example, the RoomPlan API of 2022 struggled to identify all chairs in an environment full of chairs, so if it had 10 chairs but only 3 were identified in the 3D model, this type of point cloud would only have points of those 3 chairs, and no points at all from other chairs. These points are also filtered and based on a library of "chair families", with the approximate configurations of the recognised chair, but not representing the exact chair.

Columns are other elements where the previously mentioned limitation can be found, if a column was not identified it would not show up in this point cloud – when the API identifies it, it identifies it as a set of 4 narrow walls. This first type of point cloud is shown in Figure 19. The 2023 version of the API identifies furniture with better precision, recognizing most of the items present in large quantities, and keeping better track of elements present in an environment. It also recognizes more types of elements (e.g. more types of doors and walls, curved walls, inclined ceilings, etc). The other type of point cloud is named as the "raw LiDAR" point cloud of RoomPlan, and it has colours assigned to each point, unlike the first type, and has some occlusions – only showing points that were actually collected, that had the sensor pointed towards – but retains straight planes, more orthogonal and with less drift than that of a point cloud collected without using RoomPlan. The advantage of this mode is that ceilings are present, as well as colour information, and even though there are some occlusions, all elements are present, such as all columns and all furniture. Figure 20 shows an example of this type of point cloud.

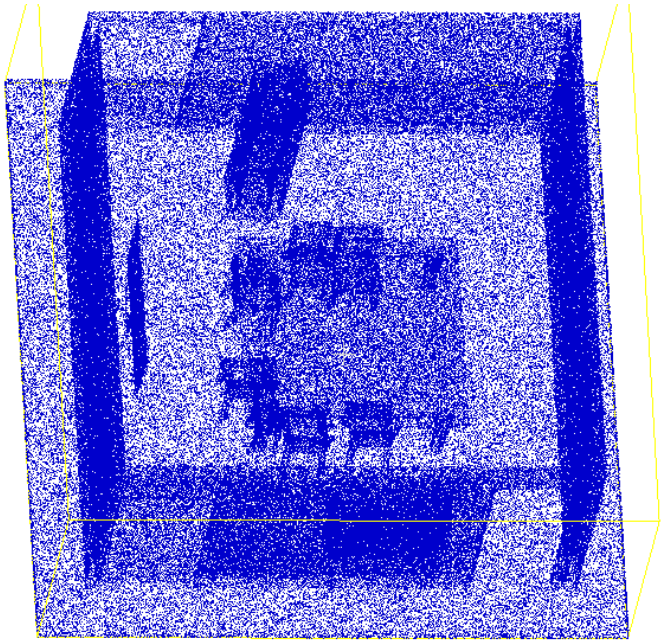

Figure 19: First type of point cloud produced by RoomPlan, based on the 3D model. App used for capture: Polycam. Sofware used for visualization: CloudCompare.

Figure 20: RoomPlan's "Raw LiDAR" point cloud. App used for capture: Polycam. Sofware used for visualization: CloudCompare. The ceiling is – ironically, as it is an advantage of this format – hidden in this picture, to allow interior visualization.

The procedure developed envisions using both types of point clouds to update the model – the first type has more consistency and no occlusions for walls, doors and windows, but the second type has ceilings and geometries of building elements such as columns that are often not detected in the more "synthetic" point cloud (see Figure 31 in Section 3.2.4). While the first type of point cloud can be obtained with a quick scan and not all areas need to be covered by the sensor, as occlusions are filled automatically to compose perfect planes, the second method requires pointing the sensor at all relevant areas during data collection. Therefore the method of point cloud collection consists of planning a route to walk with the sensor that covers all the geometry of the studied area, from all sides. That is, if the user is at a given point, they need to point the camera to the walls around them, to the floor, and to the ceiling, slowly, as they move, trying to cover areas of the floor that are covered by tables by pointing the sensor from many angles, and changing the height of the sensor for some areas. Special attention to capture the points around the areas where a column connects to the ceiling or to the ground, and all of its faces is also important. While the user wants to point the camera at all areas of the interior environment, pointing the sensor towards glass facades can confuse the recognition of walls for the module that builds a 3D model. Avoiding pointing the sensor directly and orthogonally to a window is therefore recommended, and the user should focus on pointing the sensor to the ceiling, and on the floor right next to the glass façade. If the sensor is pointed at the glass façade, it is recommended to point it as close to parallel to the glass pane as possible, to allow capturing the geometry of the glass, and possible frame of the façade, but no geometry outside. While the user walks, it is possible to visualize the current progress of the 3D model reconstruction, and see how the elements were detected, to check if more angles should be scanned to correct a wrong detection of geometry. Figure 21 shows wall detection on the 3D model even behind heavy occlusions and in a complicated sloped ceiling setting.

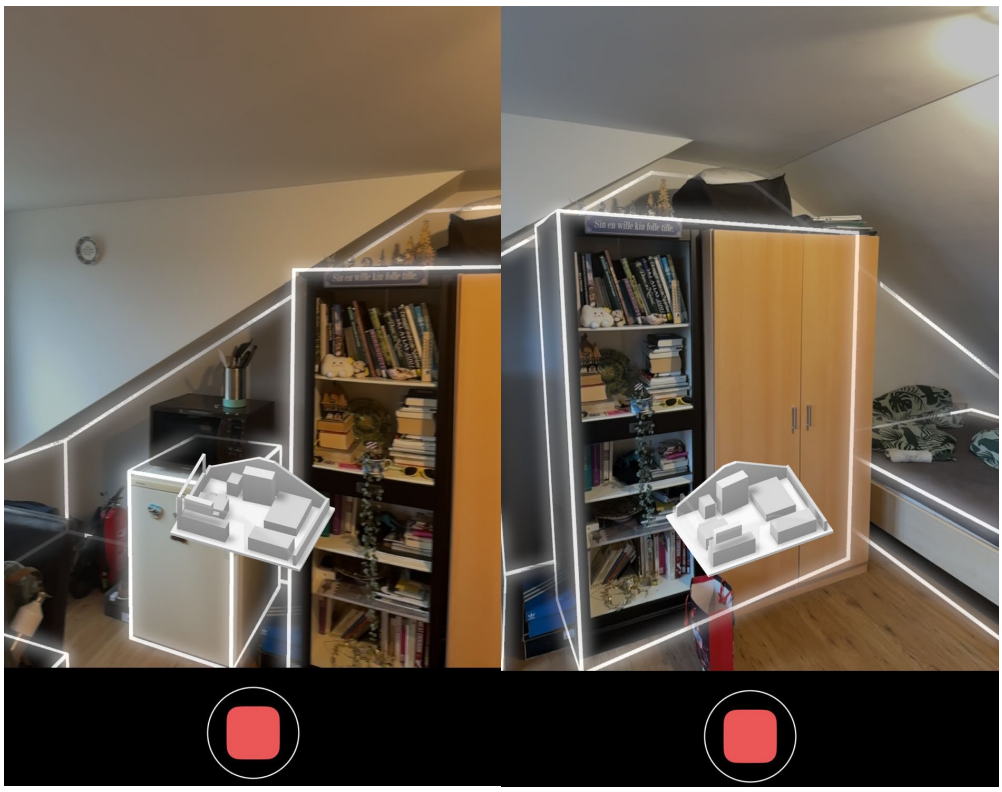

Figure 21: Real time mapping of the building in a SLAM-like paradigm, with full recognition and reconstruction of the walls behind elevated amounts of occlusion, in a complex arrangement of ceilings and walls. Application used: Polycam, using the RoomPlan API of 2023.

3.2.2 Data Formats outputted from point cloud data collection

The point cloud collection is done in the app Polycam, that uses the RoomPlan API in its "Room" collection mode. The point clouds are collected using the principles discussed above, and then they are exported using the application in the file format .xyz. To extract the point cloud of the first type discussed above, a point cloud is exported with the 3D model visible, preferably with furniture toggled off at the menu "PROCESS". At this same menu, it is possible to choose the option "PROCESS LiDAR", which generates the second type of point cloud aforementioned. At the menu "PROCESS", "LiDAR" can be toggled on and off to choose whether the first or second type of point cloud will be exported, after the processing of the "raw" LiDAR point cloud is done. As mentioned earlier, a point cloud of another origin, such as made by TLS, can also be used in the envisioned process. If a point cloud is not in .xyz or a similar format, CloudCompare can be used to convert the file format. The other file used to trigger the procedure is an IFC-SPF (STEP) file of the relevant point cloud, paired, possibly, with its respective STEP file. While STEP is a generic file structure that store 3D objects and their properties, IFC-STEP is an implementation of the IFC schema to structure STEP files by use of definitions and classes, making the use of STEP files for building projects standardized and more efficient. The processing of all those files mentioned above and how they come together is discussed in the following Section.

3.2.3 IFC file acquisition and coordinate matching of the point cloud

The IFC file used in the envisioned procedure should contain the as-designed BIM model pertaining the area scanned. For reasons such as workability and memory efficiency, if the available building model is much larger than the available point cloud of the area to be studied, it is recommended that the model is trimmed down and edited to fit the working area. With the Room Mode function

introduced later this is not necessary, but temporarily hiding or removing other floors can help in aligning the point cloud collected to its scanned area in the model, if the alignment is done manually. As a small study case to test and introduce the methodology, a room in the building Atlas of the Eindhoven University of Technology will be used to show the steps in preparing IFC data. Initially, essential steps in the update of an IFC file, based on point cloud geometry, will be tested in the small scale of this smaller project, such as wall deletion, wall creation, wall matching, update of column position and column quantities. After this initial demonstration, more complex tests will be done on larger datasets. The methodology is envisioned to be compatible with a multi-room and multi-storey framework, even though there are limits in the area that can be captured by RoomPlan API this can also be used for scanning multiple rooms, and other point cloud collection methods can also be used as a starting point to this methodology, whether mobile or static. But the methodology starts with a single room example to work its way up from that example. Other tests are also performed on data collected at the Erasmus MC hospital in Rotterdam (The Netherlands), and a larger dataset available, from the European public DURAARK project, to be found in their Github repository, named Haus30.

The room chosen is room 8.304, at the 8th floor of TU/e's Atlas building. An entire IFC model of floors 8 and 9 is made available at courses ministered in the Built Environment department of the Eindhoven University of Technology, and this model shall be used as the as-designed model to which point clouds will be compared in the initial case study. An image of the IFC model of the two floors is shown in Figure 22, with room 8.304 indicated, and its column selected in Revit.

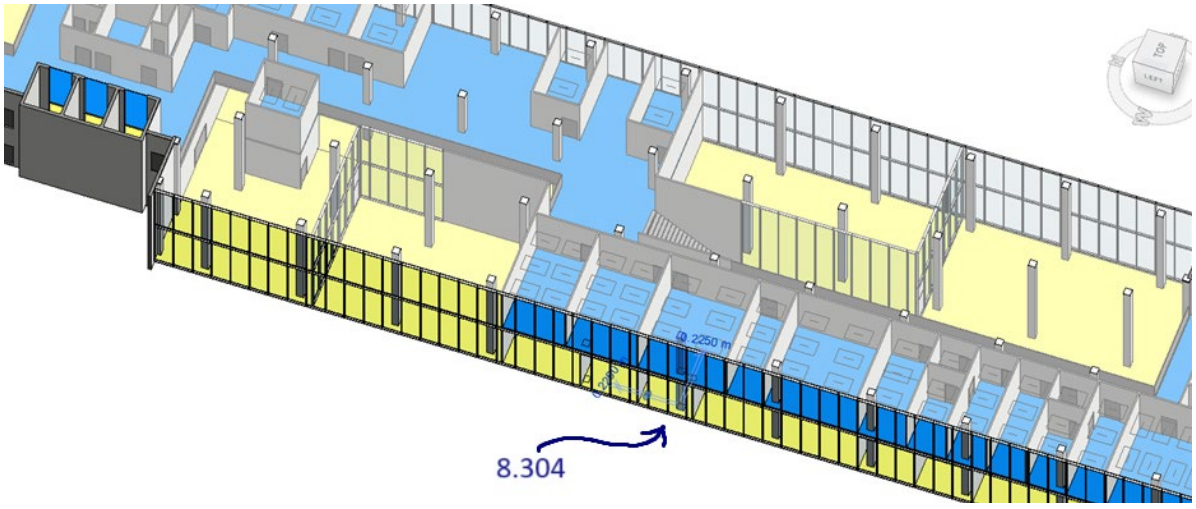

Figure 22: IFC model of floors 8 and 9 in the Atlas building, with the room used for the initial case study indicated in the picture

As the study case involves a part of the building that is way smaller than the complete IFC model, two choices are possible: change the model to remove parts that are not important to the relevant analysis, or make a new smaller IFC model that replicates the studied area. For this initial experiment, the second choice was adopted. The smaller model was developed in Revit and retained the same dimensions as the original IFC file. Because changes in door and window elements are not explored in the initial stage of the research, they were omitted, and the model is constituted of walls, a floor slab, and a column. Figure 23 shows the floor plan of the original IFC model, and the floor plan of the created reduced IFC model, especially for room 8.304.

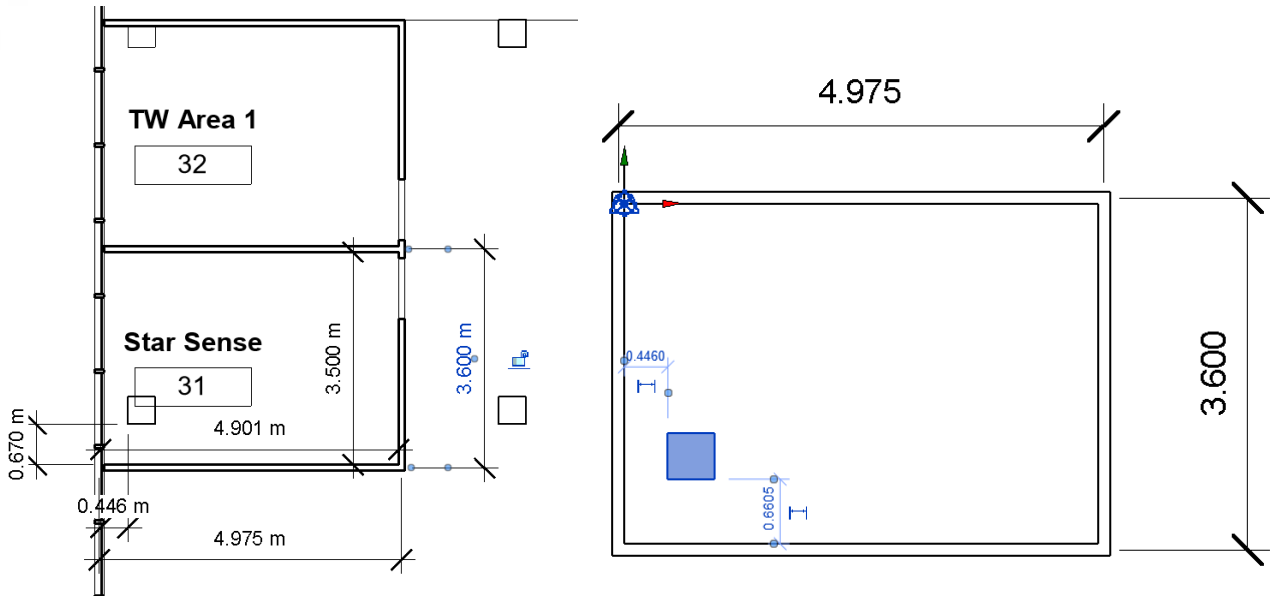

Figure 23: IFC model generated for Room 8.304 (right) and original IFC model (left).

Once the IFC is ready with a convenient Internal Origin, the point cloud of the building can be processed to be in the same coordinate system, so that the IFC building model and point cloud building are in the same place and all equivalent elements overlap. Setting the point cloud to the same coordinate system of the IFC model can be done in CloudCompare. CloudCompare does not work with IFC models, however, so a simple conversion of IFC into STEP format, which is accepted by CloudCompare, can be done using the software FreeCAD (that is indeed free). To do so it is sufficient to open the IFC file in FreeCAD, double click the visualization of the model, and go to the tab "file", and export it as "STEP with colours". Alternatively, the python package IfcConvert, that can be installed together with IfcOpenShell, can convert an IFC file into STEP using only a few lines of code. This idea is implemented and discussed in Section 3.6.

After that, both STEP file and point cloud can be opened in CloudCompare. First the point cloud can be opened, and be trimmed to remove some noise and points collected outside of the study area. This can be done by selecting the point cloud used on the menu to the left, and then using the Cross Section tool to trim the point cloud. Those options are shown in Figure 24. In the "Cross Section" function, use "Export selection as new entities", under the "Slices" option to trim the model.

Figure 24: By activating a point cloud in the menu shown in the left, some tools become available, such as the Cross Section tool, that can remove unnecessary points found around the working area

The conversion into STEP might change measurement units for the STEP file, and upon opening the point cloud and the STEP file, the STEP file might be in millimetres or a different unit. It might thus be necessary to downscale the STEP file to one millionth of its previous dimension. When point cloud and STEP file are at the same scale, it will probably still happen that they are not aligned to each other, as Figure 25 shows.

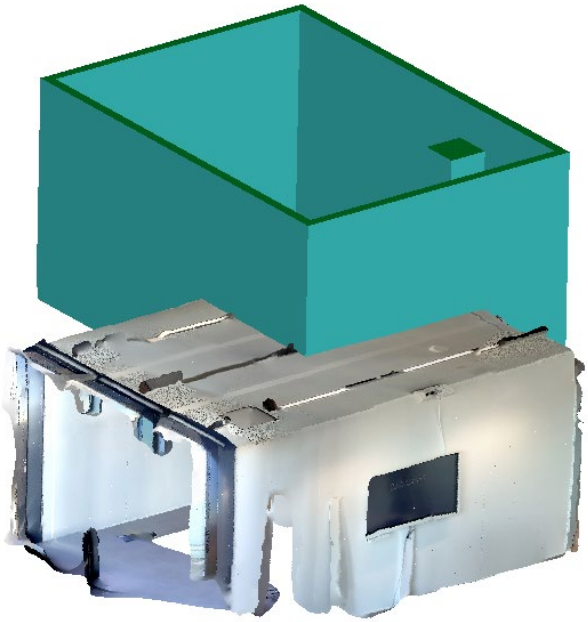

Figure 25: Difference in alignment between BIM model and point cloud

To align manually both models to each other, the point cloud will be selected as shown above, and translated and rotated using the "Translate/Rotate" tool. The reason the point cloud is moved instead of the BIM model, is that the IFC file has references to coordinates, which should not be changed. Moving the point cloud to the same alignment of the STEP file, inside the same working space, will make sure that points are in the same coordinate system. When that is done, having the point cloud selected, the option "Save" should be used to save the point cloud as a new file with the new coordinate system. Figure 26 shows point cloud and STEP file aligned to each other. The dark protruding building elements on the left side are part of the façade's frame.

Figure 26: Product of the alignment of point cloud and STEP file. Notice that it is already possible to spot a difference in as-designed and as-built position of the column (rectangular column at the left side of the image), which can be verified in the real building

Wall-extent standard adopted for IFC wall modelling

An important step in the comparison of segmented point cloud walls and IFC walls is that all the walls that should match also have the same extents, that is, start at around the same point and end at around the same point, in the as-designed IFC and at the point cloud. This is step 0b from the procedure mentioned in Figure 15. If that is not the case, the Scan-to-BIM functionality of the IFC updating tool will still make sure that the layout of the walls is updated to its as-is condition, but many as-designed IFC walls that were previously present at the model, and whose geometry is also found at the point cloud, will be deleted and replaced by other IFC walls that match the extents defined by segmentation. An example of how the layout of a building region can be represented by different extents of walls is shown in Figure 27. Even though in the real world the layout of the walls is the same, the extents of the wall entities and even the number of walls can be different. If the segmentation follows the pattern shown in the right side, and assuming the left side represents the as-designed IFC, many walls will be deleted from the left side, and replaced by other IFC walls, even though in practice the layout is the same, because the start and ending points of the IFC walls and the point cloud walls don't match. Assuming only a partial scan is performed, of only the small room on the top left corner of the space pictured in Figure 27, if the walls are defined in IFC as the pattern of the left side is, the wall that is called "wall1" on the right side cannot even be matched to any IFC wall, because its extents are only half as big as the "wall1" on the left figure. Having a segmentation process that defines walls based on shorter and continuous planes is therefore a better approach to match walls, specially when only a section of a building is scanned and has to be updated. And it is also optimal if walls are also defined in a similar way in IFC, even if this process of redefining the extents of the walls in IFC has to be automated by a script (that is outside of the scope of this research).

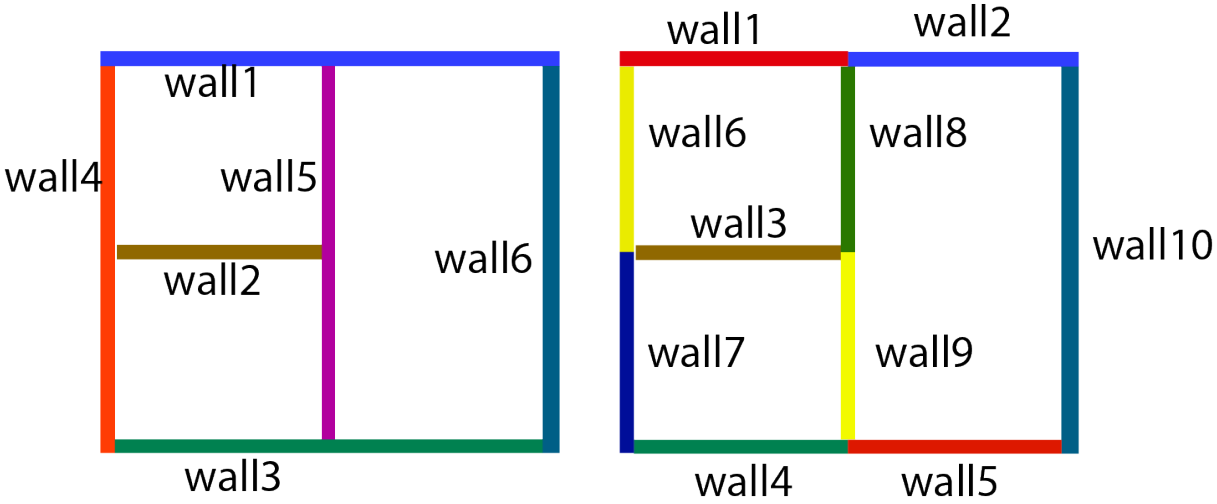

Figure 27: Example of how the same layout of walls can be represented in different ways.

While on a real building all walls (often) are connected and form a sort of a single mass, in construction software this is not so simple. In a simple small example such as that of Room 8.304, where only four walls are present with simple connections, this might not be evident, but larger buildings with e.g. corridors and rooms on both sides can have quite arbitrary definitions of walls that are often chosen by the modeler at random. In Figure 28, for instance, at one side of the corridor, one of the walls covers almost the entire extent of the corridor, and on the other side, the walls are defined in a different way, even though the corridor is symmetrical. In this way, as the comparison to check the geometry is done wall by wall, it would be impossible for a segmenting tool to know which parameters the building modeler chose to define the extents of any given contiguous set of walls. A

few methodologies mentioned in the Literature adopt an approach of generating the entire layout of all walls first, based on the point clouds, and then creating walls based on this complete blueprint. Because the approach of the current work is checking wall by wall, a methodology that reconstructs a merged layout of all walls and then splits them could hardly work without tremendous complexity. Besides, because the goal is performing local changes and maintaining semantics, recreating the entire layout and all walls would be counterproductive, and would furthermore maintain the struggle in comparing the updated with the outdated stages, as a total update approach does not guarantee that the definition of wall extents would be similar. For scans that cover only part of a building this is even more problematic, as a wall that is modelled continuously across the building might have its origin point outside of the scanned area, but part of its extent in the scanned area, but not be properly recognized because of its starting point outside the scanned area that needs to be checked.

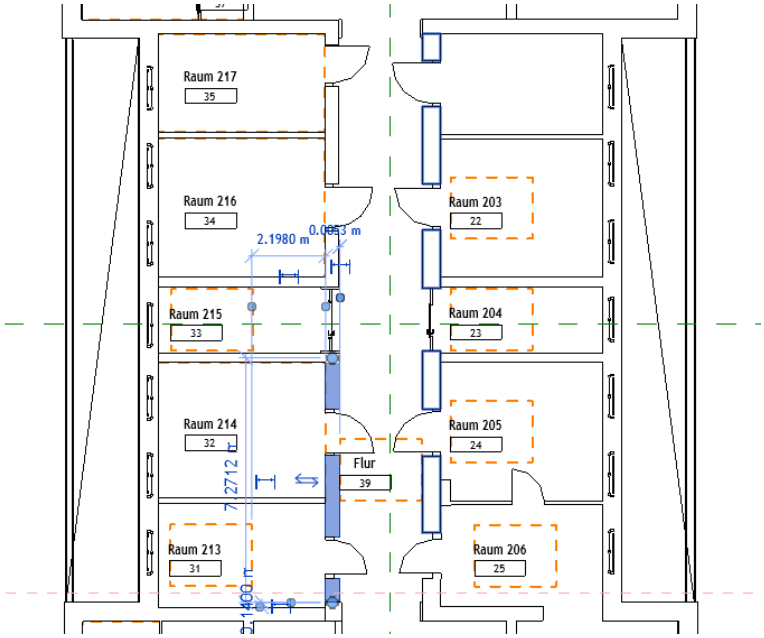

Figure 28: Problematic of the inconsistency of wall extent definitions in larger building models: one wall on the left, selected, occupies half the corridor, and another wall on the right, highlighted, occupies almost the entire corridor.

Therefore, the definition of wall units, and by extension their extent, where a wall should begin and end, was chosen as the shortest segment possible. The wall ends as soon as a connection to another wall, either orthogonally (or diagonally), or at the continuation of the wall in a different room. Figure 29 exemplifies an application of this type of wall design definition. Because in the case of contiguous walls it cannot be known where the modeler will stop drawing a specific wall, the wall is defined instead as the shortest assumption. This standard for BIM model production is useful in expanding the procedure developed in the current work into a multi-room and multi-floor application. It also helps better tracking of which walls were already present in the building and did not change. In this way, this can be a standard for modelling of buildings whose management benefits from updates, or, in the case of already designed IFC models, the walls can be automatically split using a script or plugin, which is outside the scope of this research. Therefore, the choice was made to fully delete IFC walls that do not fully conform with the point cloud status, and create new wall entities where necessary. This is further explained in Section 3.3.

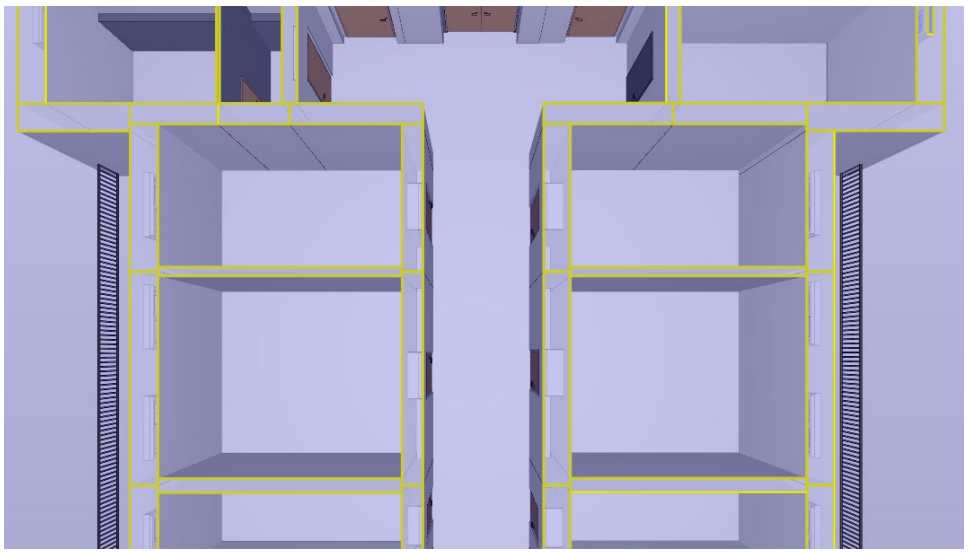

Figure 29: IFC model with the walls subdivided (in yellow) to fit the wall definition adopted

3.2.4 Segmentation and classification of point cloud into building elements

In a similar fashion to that of how Roman et al. (2023) departs from the preliminary input of a point cloud segmented in different ASCII text files, each one being a point cloud labelled as a relevant building element to be modelled, this research also starts with the assumption of segmented building elements (e.g. walls, floors, ceilings, columns, etc). Each one has a label that is the title of the text file, in a similar fashion to the S3DIS (Stanford large scale 3D Indoor Spaces dataset), a famous dataset used by many authors in the field, introduced by Armeni et al. (2016). One distinct difference from the current thesis' methodology from that of many methods explored in the literature, including that of Armeni et al. (2016) and their S3DIS data set, is how walls are classified and segmented. Many authors adopt a room-based approach and create separate files for each room, having e.g. one point cloud file for each room, and more point cloud files for each element in the room. The room is considered by them as a sort of box composed by the visible faces of the walls that bound the room, the visible face of the floor, and the visible face of the ceiling. Each one of those elements (e.g. each wall) has their own point cloud file, of the plane that represents them, and other elements such as furniture, columns or clutter also have their respective point clouds, but then in 3D, usually. This seems to make sense on a point cloud paradigm, where the available information is the visible information, and not the encased information. This is also related to the strong computer vision background of much of the current research – researchers are often focused in classifying and locating elements in space, not necessarily reconstructing their entire hidden geometry. In a process that seeks to reconstruct BIM models, however, this is not the best approach, and the trend and goal found in the current literature is to reconstruct volumetric elements, specially walls, as their representation in BIM standards such as IFC is also volumetric.

Nevertheless, a minority of authors has been able to create frameworks that reconstruct volumetric and IFC-based walls. Very interesting procedures such as that of Ochmann et al. (2019), Bassier and Vergauwen (2020) and Valero et al. (2021) are good examples of models that can reconstruct volumetric elements in IFC based on point clouds. Usually those frameworks that create volumetric elements encompass the process of point cloud segmentation and IFC generation under the same

procedure. They deal with the creation of a layout of all walls at once using semantic segmentation based on ML techniques, and then generate IFC walls based on the blueprint of the wall layout discovered. For the envisioned procedure worked on this thesis, however, not all walls are created deleted, or edited at once. For the most part, the tool is envisioned to update changes done within the internal walls of a building (even though exterior walls can also be changed), and changes in other elements such as column position, dimension or ceiling height. Many of the elements present in the model are thus kept intact, and this is also important, as their semantics can be used to enrich new elements that are changed or added. For example, an interior wall found to be at a different position than that of the as-designed project may receive the same standard wall type of other interior walls of similar thickness, unless the user decides to manually choose another type. The implication of this objective is that the walls should be identified and segmented in a volumetric fashion, but also take into account the flexibility of maintaining most walls and being able to modify others. To make that possible, it was decided that it works better to ***define a volumetric wall as the segment of a wall that has no discontinuities at both sides of the wall – the wall does not end nor does an orthogonal wall start at either side of the wall. In this way, walls are kept shorter, which gives more flexibility to work with them.*** This works in the same fashion of the adopted definition of wall design in IFC just discussed at the Section above. It helps avoiding the ambiguities found when going from a surface based geometry into solid based geometry, that were also found and outlined by Nagel et al. (2009) when converting CityGML files into IFC. Most rooms will have the dimension and location of its bounding walls coincident with the room boundaries themselves, or as a sum of (probably up to two or three) wall segments, at an individual side of a real wall – the latter option applicable if there is an orthogonal wall on the other side of a wall face. The way to achieve this at the segmentation stage would be to merge planes with opposite normal vectors within a threshold comparable to the thickness of a wall, and that also fulfil other characteristics of a wall such as appropriate height, but break the creation of a wall element when a discontinuity is found or when another orthogonal plane is found. Figure 30 shows an example of this concept of wall segmentation.

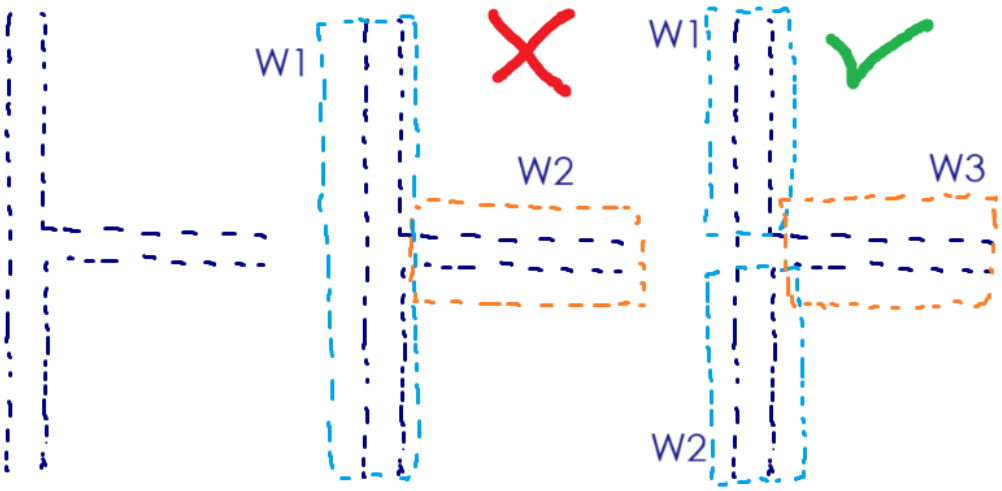

Figure 30: Method adopted for wall segmentation – requires continuity of points (and no connections of walls) on both sides

With these criteria for segmentation defined, the semantic segmentation can be performed manually using CloudCompare. It could also be done automatically, based on the criteria established, but

development of ML methods to detect planes and semantically segment building elements is beyond the scope of this thesis. Due to their reliability not being yet 100%, a manual procedure yields furthermore the completely reliable input data that is necessary to kick-off the update procedure. To manually segment building elements in CloudCompare, the same "Cross Section" tool discussed in Section 3.2.3 can be used to segment the vast majority of building elements, that is, all those that can be fit in a bounding box, as long as this bounding box does not include other building elements. If that is the case, the unnecessary points can be removed. Other tools are also available in CloudCompare to select points for more complex geometries. Using the "Cross Section" tool, the user should limit the extents of the bounding box until only the desired element is visible, and then click "Export selection as new entities". After that, the segmented element is saved as a new point cloud at the desired directory, with the semantic label of which type of building element it represents as its filename. In the methodology used for this work, both point cloud types discussed in Section 3.2.2 can be used to verify the as-designed building and update it. The first type of point cloud is more reliable for walls, as it has complete planes with occlusions filled in. The second type of point cloud is used to segment ceilings, as those are absent in the first type, and is also more reliable for column geometry – and other possible building elements such as fixtures or beams, whose update is outside of the scope. Figure 31 shows the difference in column recognition between both methods.

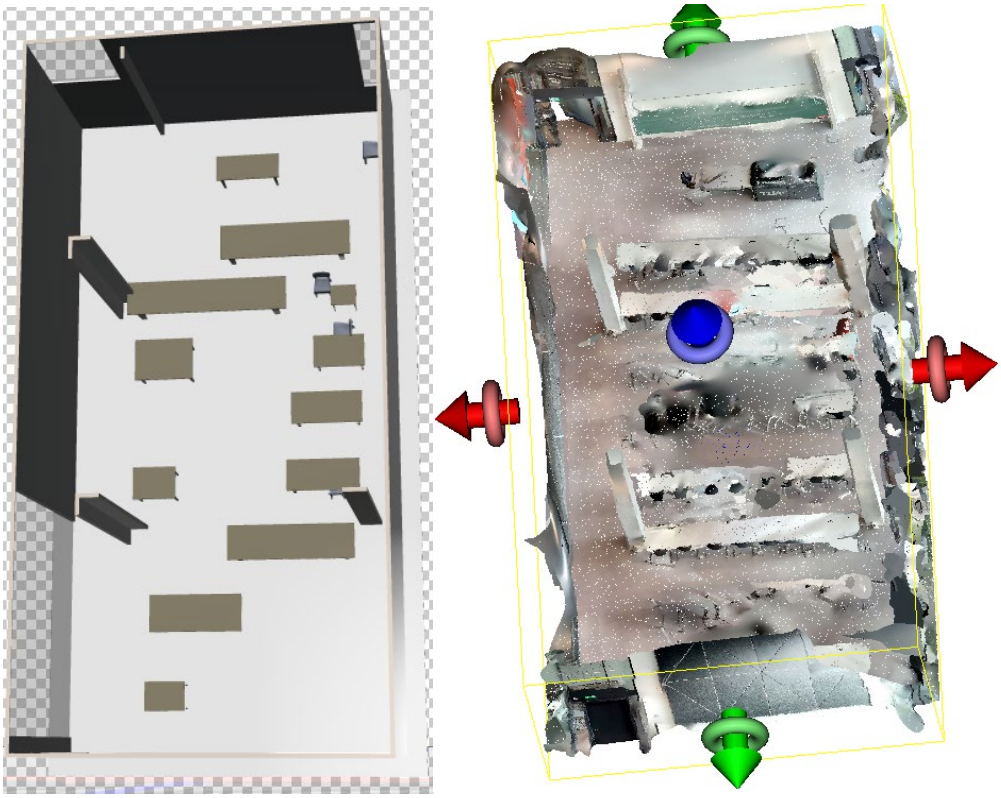

Figure 31: To the left, the generated 3D model, on which the first type of point cloud described in 3.2.2 is based. There, columns are incomplete and identified as wall segments. To the right, the column geometry is more complete (the ceiling at the right side was hidden to aid visualization)

Below, Figure 32 shows the process of segmenting a wall from the first type of point cloud, and segmenting a ceiling from the second type of point cloud, in CloudCompare. It is worth noting that the envisioned method can also be applied to point clouds from TLS or mobile sensors, in which case only one type of point cloud (more similar to the second type, a raw LiDAR point cloud) is used. While the process of volumetric wall generation requires some considerations discussed above, the

update of ceilings envisioned in this framework can work well with information extracted from the surface labelled as ceiling. As most deviations that can happen in the geometry of a ceiling will involve deviations in its height (or area), the procedure adopted to update ceiling geometry will be to extract the height of the ceiling at the point cloud, and update the height of the ceiling element in an IFC file. Another possibility of changes that a ceiling can incur are changes in its area in the case of a renovation of the room in which it is encased. Those changes are likely to be two-dimensional and follow the outline of the room's boundaries. A similar approach to that shown in Figure 32 would be used to segment a column, a door or a window.

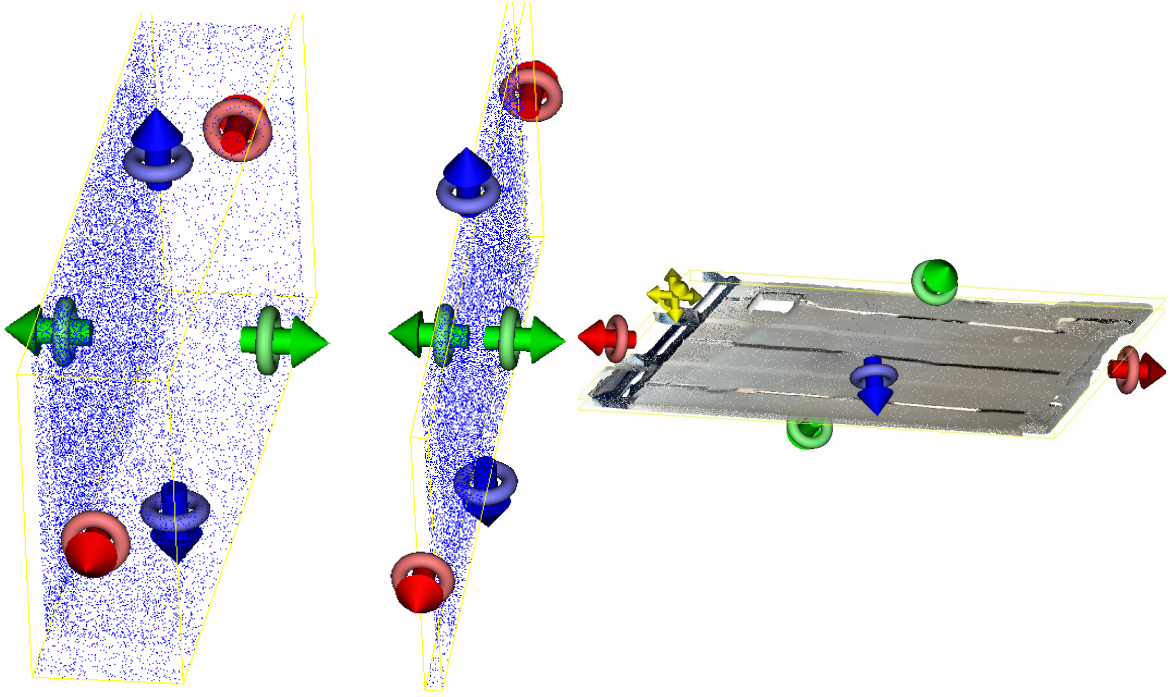

Figure 32: To the left and centre, the process of moving the extents of the section box to select only the two faces of a wall. To the right: face of a ceiling selected to be exported as a separate point cloud

3.2.5 Data storage and management

Even though the proposed procedure is done on a file-based paradigm, a more ideal solution would be bringing the model update into a linked data and web-based approach. In this way, the data can be split into small nodes of data, that are individually updated, such as replacing one property of a building element, instead of replacing the entire file of the building model because of a small change. Another advantage of a web-based implementation of the proposed procedure is working directly with IFC viewers, which is much faster than converting files from IFC-STEP into STEP to visualize them in python. "Room mode", a concept that is further explained in Section 3.7, applies to some extent the idea of limiting the scope of elements that are processed or edited. This mode consists of performing a local check instead of a global check across the entire model. For instance, for the base idea of the developed procedure, the collected and segmented point cloud is taken as the ground truth, as the geometry of the real building, and every building element in the entire IFC model is compared to this ground truth geometry. If an IFC element does not match to an element found at the point cloud, it will be deleted. The problem is that if a scan is incomplete all elements that are not scanned will in theory be deleted. Furthermore scanning tools such as smartphones can currently only scan a limited area. For larger buildings it is also the case that changes will often only occur at a section of the building, making the complete scan of the building burdensome and unnecessary. Room mode

solves this by obtaining the extents of the scanned data (a bounding volume around the point cloud, will a small tolerance buffer), and only checks IFC elements within that bounding volume, updating the local changes and leaving the rest of the IFC model unaltered. In the proposed implementation, data can be freely stored in folders in the form of files, in IFC-STEP format for the BIM model, and .xyz or .txt for point clouds. Reports are produced in .xlsx.

3.3 Match and update of Walls

One of the most important and most explored building elements in the present literature is the wall. Together with floors and ceilings, they make the basic boundaries of building spaces, and most of the other geometry and elements relate to them and build on top of (and around) their topology. This complexity in connections and relationships in the IFC data structure creates a lot of variables that need to be accounted for, when changing, creating, or deleting a wall element. This Section on walls will therefore explain the process envisioned to match as-designed and as-built walls, and update the IFC file with the new wall configuration. Figure 33 shows the general scheme of the update of walls according to the procedure's steps described in Figure 15. Section 3.3.1 shall discuss steps 4 and 5, Extracting geometry data from segmented elements and Matching them, and Section 3.3.2 discusses Step 6, the three steps of model update for walls. Section 3.5 however includes more information on Room Mode. The check and update process is based on IfcRectangleProfileDef-defined walls.

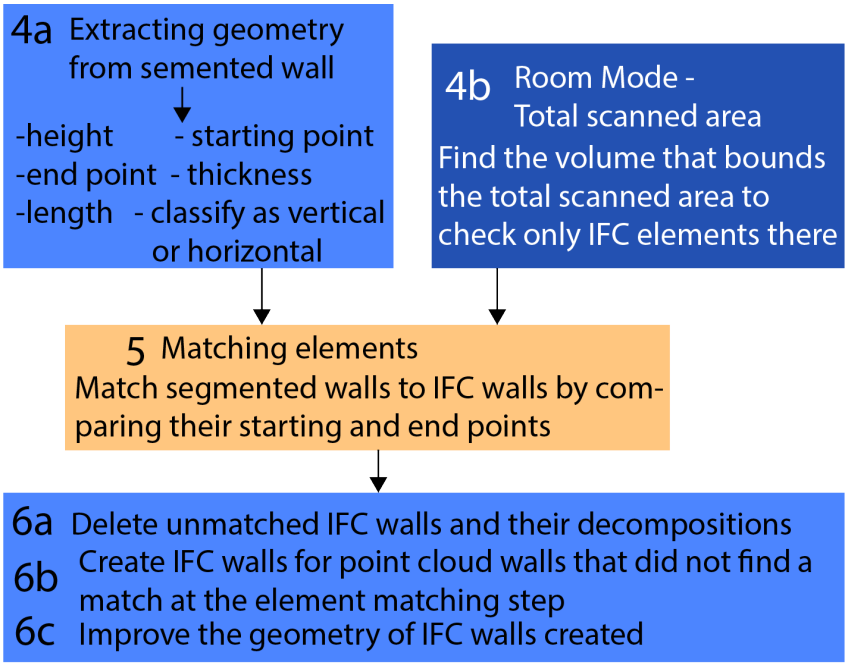

Figure 33: Overview of the steps taken in wall update following the procedure described in Figure 15

3.3.1 Recognition and matching of walls

The first step to aid the update of a building project into its as-is state is to recognize the similarities and dissimilarities when comparing as-designed and as-is state. For some wall types, such as walls defined by an IfcArbitraryClosedProfileDef and diagonal walls in general (which are most often defined by the profile type just mentioned), the tool may not be able to automate all updates without some features being added. All walls created by the algorithms developed have a profile of type IfcRectangleProfileDef . Principles that became evident during the implementation of the procedure in a Manhattan-World scenario, and that would aid the implementation of diagonal wall types into

the procedure developed will be however discussed in Section 3.3.1.2 and Section 3.3.1.3. It is also important that the process is overseen and sanctioned by a professional even in fully automated processes. At steps 4 and 5 of the process mentioned above, segmented walls in point cloud format, and walls of an IFC file are examined to match the wall layout of the as-designed and as-is building. A list is produced as output, containing which point cloud walls found a match with which IFC wall, which point cloud walls did not find a match, and for those latter walls, how they can be added to the IFC file. Furthermore, IFC walls are also examined, and the ones that do not find a match with the point cloud walls are walls that do not exist in the real building [anymore], so their global ID is outputted and they are deleted. The output of this process is generated both as an excel file with information for the modelers, and the same data is internally used by the script to update the IFC file automatically in step 6. One of the reasons to adopt a framework where walls are deleted if they do not match, instead of merely updating the definitions of a wall is the standard for extents of a wall described in Section 3.2.3 and 3.2.4, that makes it easier to change the layout of walls by having smaller elements constituting the layout. The most important reason however, is to avoid ambiguities that could be generated in some cases where the layout of walls is changed. Consider the situation in Figure 34, where a wall receives in opening, in a renovation executed, and is then divided into two separated walls. If the choice would be made to edit the previously existing wall, there would be ambiguity when choosing which one of the two distinct walls that now exist is the old wall entity that becomes edited, and which one will be represented by a new wall entity created. It could be either one, and in complex wall arrangements this could generate problems in interpreting the situation. Besides, the principle of preserving the individual wall entities that undergo changes, and track their changes, is corrupted, as wall 1 and wall 2, at the lower side of the figure, the remaining segments of the original wall1, cannot be both represented as the same wall entity anymore.

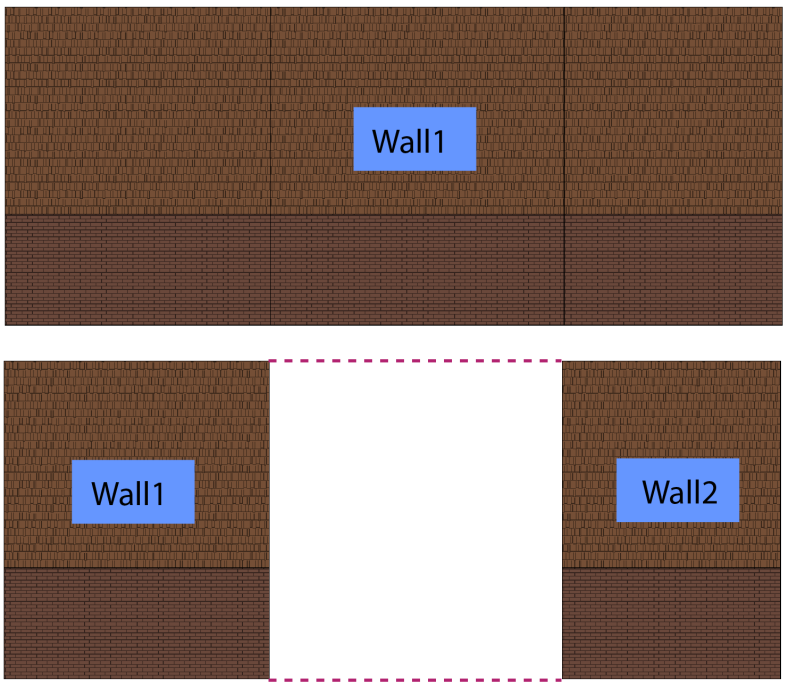

Figure 34: Example of a case where changes in a wall create ambiguities into the individual history of that wall. Therefore, non-matched walls are deleted, and new walls tagged as modifications are created to represent the changes in layout. The history of the layout is tracked for the collective of walls instead of focusing on individual walls. Each wall is tracked to an OwnerHistory, that is only updated if the wall is also updated.

3.3.1.1 Standard for wall segmentation

When generating the segmented walls in point cloud format, a choice is done to have walls defined as volumetric wall segments that have continuous planes on both sides, as defined in Section 3.2.3. and 3.2.4. The partial exception of the wall definition would be external walls, that rarely have their exterior geometry included in interior scans. In this case a check has to be done on whether the single plane, that constitutes the interior side of an exterior wall, finds no relevant point cloud geometry (besides noise) on the opposite side to its normal plane. Then it can also be considered a wall at the segmentation step. The wall segmentation step should take place either before of after the coordinate alignment step mentioned in Section 3.2.3, so that the building elements are all segmented in the expected alignment of the IFC model, or segmented first and then collectively moved to the correct alignment.

3.3.1.2 Wall types in a point cloud (vertical and horizontal, diagonal)

Most buildings generated in BIM authoring tools, as well as most point clouds generated by the RoomPlan API used in this research, have the main walls of the building aligned with the x and y axes. Viewing a building from its top view, as a floor plan, the standard is to see walls that have their longitudinal axis in the horizontal direction (growing x values), and walls that have their longitudinal axis in the vertical direction (growing y values). They will be called here horizontal and vertical walls, respectively. The remaining walls, not aligned with the main axes, or that are angled for different reasons, shall be called diagonal walls. The python implementation developed to test the procedure of this work will only check and update horizontal and vertical IfcWallStandardCase entities, with their body shape representation defined by a IfcRectangleProfileDef, to introduce the principles of wall update by using the most common wall type.

The distinction of horizontal, vertical and diagonal walls is done to help locate point cloud walls and IFC walls in space, compare their locations, and compare their base and end point coordinates. At a first look, the coordinates present in a point cloud file representing a wall may seem chaotic, but with those three heuristic assumptions – horizontal, vertical and diagonal walls – the starting point and ending point of each wall can be obtained. It is important to classify a wall in each one of those three, because the way of finding their starting and end points will be different for each one of those wall typologies. Horizontal walls, for instance, will have the y values of their coordinates within a very narrow range, compared to the x values. That is because two planes are present (one for each side of the wall), and, with the exception of minor noise in the data, most y-values will stay withing a range of e.g. 25cm, or something around that. In that way, a list can be made with all y-coordinate values present at that wall, and check the minimum y value, and compare its distance to the maximum y-coordinate, and if this distance is within the 25cm mentioned range, or a similar range to the expected wall thickness of the project, and the difference between the lowest value of x and the highest value of x is much larger than that of the y values, then this wall can be classified as horizontal. Some models can have walls, especially external walls, much thicker than the average though, as will be seen in the Results Section (4). The threshold should be liable to changes depending on the dimensions of the thickest walls of a project.

Given that the y-coordinates stay constant, within a certain range, for horizontal walls, the largest variation in coordinates will happen along the x-axis. The starting point on the x-axis of the wall can therefore be determined as the lowest x value, and the ending point on the x-axis can be determined

as the highest x value. Both the starting and end coordinates of the wall on the y-axis will be the average value between y-min and y-max, that is, the center of the wall. In IfcWallStandardCase entities the wall is also determined as starting at a point aligned to the centerline of the wall, and the polyline that represents the wall is the centerline of the wall, which allows an easier comparison. A scheme of a horizontal wall with its centerline, and x and y values that allow the calculation of the start and end points of the wall is shown in Figure 35. Likewise, in the procedure to find starting and ending coordinates for horizontal walls, vertical walls have a more or less constant x value instead, and the wall changes in value from the lowest value of y to the maximum value of y. The x coordinate for starting and ending points of a vertical wall will therefore be the average x value, representing the centerline, and the y coordinate of the starting point is the minimum y value, and the y coordinate of the end point will be the maximum value of y present at the point cloud of that wall. A vertical wall should also have a higher difference from the maximum y value to the minimum y value than the difference between the maximum x value and the minimum x value, that is, it changes much more in the vertical direction.

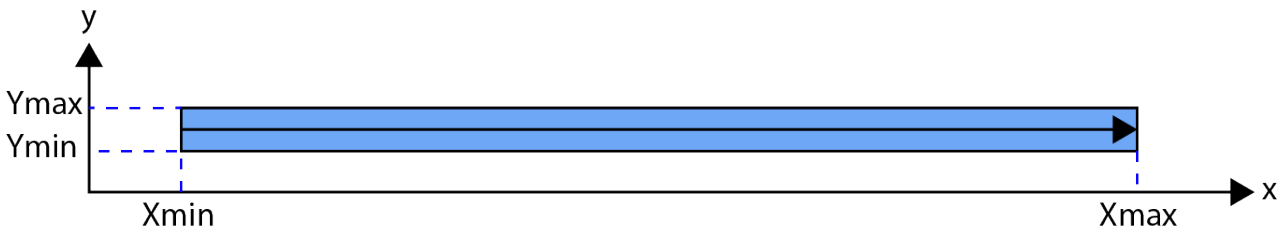

Figure 35: A horizontal wall (seen from above) with the relevant x and y values that indicate the starting and ending points

Walls that don not fit either criteria, that is, have neither x or y values within a small range, are classified as diagonal walls. In the developed tests to apply the procedure proposed, diagonal walls are not used, but relevant information about diagonal walls could also be extracted from their geometry to apply them into this methodology. An alternative way of classifying a wall in horizontal, vertical or diagonal would be making a linear approximation of the x and y variables, and then estimating from the vector that gives the direction to the wall, which x and y component the Reference Direction of the wall will be. If a diagonal wall has an inclination of 45 degrees from the horizontal x axis, for instance, its RefDirection will be (0.5, 0.5, 0.0), that is, it is growing as much in the x axis as in the y axis in the positive direction along the length of the wall. For a linear approximation that indicates that the wall is aligned to the x axis with a tolerance of plus or minus 5 degrees, it can be classified as horizontal, and if it indicates that it is aligned to the y axis with a similar tolerance, it can be classified as vertical. Other walls could then be considered as diagonal.

The method used to classify walls in horizontal or vertical here, however, is simply based on the axis (x or y) that they grow the most, and keeping variation in the other axis within a threshold, equivalent to the range of the expected thicknesses of walls. Other information should also be obtained about the geometry of the wall. The height of the wall is estimated as the difference between the maximum z value and the lowest z value in the segmented wall, and the thickness will be the difference of y_{max} and y_{min} for horizontal walls, and x_{max} and x_{min} for vertical walls. The length of the wall is the opposite, as the thickness, the difference between x_{max} and x_{min} shall be the length of a horizontal wall, and the difference between y_{max} and y_{min} is the length of a vertical wall.

3.3.1.3 Locating walls starting and ending positions in an IFC file

At the previous Section, the methodology to find the starting point and the end point of a wall in point cloud format was outlined. The starting point is considered the lowest value in x and y (one of them being perhaps constant depending on the wall being vertical or horizontal), and the end point is considered the highest value in x and y (one of them possibly being constant as well). It is then assumed that vertical walls start at a low y value and end at a high y value, relatively, and horizontal walls start at a low x value and end at a high x value, relatively, growing from down to up and left to right, respectively. In IFC files, however, this convention is often not followed.

IFC elements have a local placement that is always relative to the subdivision in space in which they are located. An IFC wall has thus its location given relative to the IfcBuildingStorey it is located in, and the building storey has in turn its location represented relative to that of the IfcBuilding, and the IFC building has its location relative to the IfcSite. The coordinate system of the IfcSite is not relative to any other entity, so it can be considered as a Global Coordinate System (GCS) (Tsimpiskakis, 2024). In practice, what we see, is that the shape of a wall is described in its IfcShapeRepresentation entities, in the local coordinates that represent the geometry of the wall. If in the global schema of the building the wall has an orientation that can be classified has horizontal, vertical or diagonal however, this wall might, in many cases, receive a reference direction (RefDirection) entity. This RefDirection entity gives the vector, or reference axis, that describes the positioning of the longitudinal axis of the wall relative to the coordinate system of its context, that is, of its building storey. The length and starting and end point of a wall are described in its shape representation always in the x-coordinates, regardless of them being horizontal or vertical walls. If they are vertical their “vertical” position relative to the storey will be given using a reference direction. This reference direction also helps representing the wall as vertical in a 3D visualization. This reference direction also describes the direction (positive or negative) of the wall relative to its starting point. For example, a vertical wall that starts at a higher point in the y-axis, relative to the IfcBuildingStorey, than the y-coordinate where it ends, will have a RefDirection of (0.0, -1.0, 0.0), as shown in Figure 36. The reference axis is -1 in the y direction because the wall “grows”, that is, has its direction, in the opposite direction of the y-axis, the coordinates in y become smaller towards the end of the wall, looking at the coordinate system of the storey.

The Local Coordinate System (LCS) of the IfcBuildingStorey can be considered as the same of the IfcSite – the GCS – for most buildings, with the exception of the z-axis, as the z-coordinate where a wall, or other IFC elements start is relative to the level of the IfcBuildingStorey. Therefore, the starting point of a wall will often have 0.0 as its z-coordinate in IFC, starting at the same level of the floor. Point cloud walls however have their starting point z-coordinate with GCS values, usually different than 0.0 for multi-storey buildings, so this difference in local and global coordinates has to be taken into consideration in the algorithms that compare and match walls. To find the end point of an IFC wall, and compare it to the end point of a segmented point cloud wall, it is also necessary to take into account the possible RefDirection of the wall. Each IfcWallStandardCase entity has its starting and end points described in local coordinates as a polyline. The starting point of the polyline will be (0.0, 0.0) and the end point will be (x, 0.0), where x represents the length of the wall. Based on this, the length of the wall can be obtained, and the coordinates of the end point in the Global Coordinate System, that the point cloud uses, can also be calculated. They will be calculated using the RefDirection, so the length value can be added or subtracted to the x or y axis of the starting point, depending of which RefDirection is used, in order to obtain the end point in GCS. After that, the

starting and end points of IFC and point cloud walls can be compared under the same coordinate system. When the Local Coordinate System of the wall has the same orientation as that of its context, which is a horizontal wall, that has its longitudinal axis along the x-axis, and has its end point with a higher x-coordinate than its starting point (grows along the x-axis), relative to the building storey, it will have no RefDirection under its RelativePlacement property.

Diagonal walls have their RefDirection containing coordinates in both x and y axes, with a value different and lower than 1, e.g. (0.5, 0.5, 0) for a wall inclined 45 degrees compared to the IfcBuildingStorey's x axis. Figure 36 clarifies how IFC represents the start, end point and direction of an IfcWallStandardCase entity.

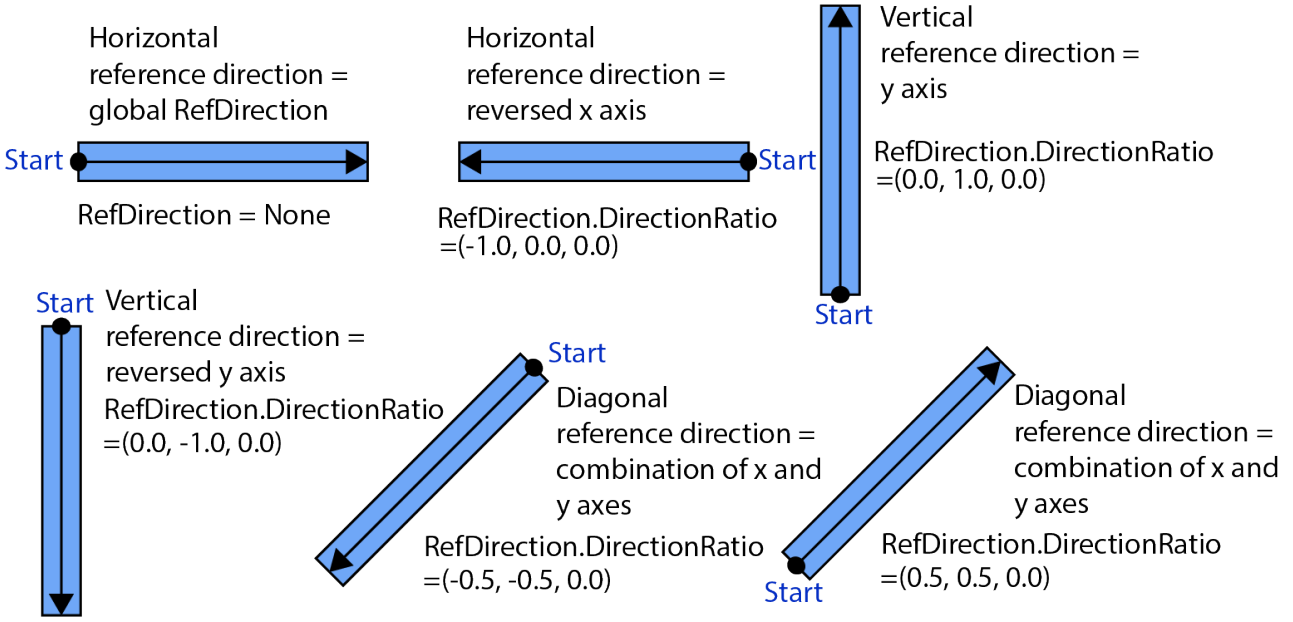

Figure 36: Based on the DirectionRatio tuple of the Reference Direction of a wall, the orientation of a wall in IFC can be obtained, and follows the scheme shown above. If the general directions of the wall are the same as the storey, no RefDirection attribute needs to be assigned to it. In each wall shown, the starting point of that typology is shown, located at the centerline of the wall. The arrow shows the direction and the end point of the wall, as well as the length of the wall and the polyline that represents it.

Obtaining the starting point of a wall in IFC, in global coordinates, to compare it to that of the point cloud wall is quite straightforward. The local placement of the IFC wall in the IfcBuildingStorey entity has a 3-dimensional coordinate, where the x- and y-coordinates are the same as the global coordinates. The z-coordinate is relative to the building storey, so finding the level of the building storey also allows knowing directly the z-coordinate where the wall starts in global coordinates. But finding the end coordinate is more complex. At the description of a wall's geometry, one of its IfcShapeRepresentation entities, the end point is described as the end of a polyline that starts in (0.0, 0.0). This end point represents thus the length of the wall, and this length can be extracted, and based on the RefDirection orientation of the wall, it can be added to the starting point to also find the end point of the wall.

To test which walls are present in the model two approaches can be taken. One of them is to compare IfcWallStandardCase elements to point cloud walls, in a big conditional that checks whether the start coordinates of the point cloud match to that of the IFC Wall, and the end coordinates match, or alternatively the start coordinate of the point cloud match with the end coordinates of the IFC wall

and the end coordinate of the point cloud match the start coordinate of the IFC wall, as the IFC wall may have its start and end coordinates "reversed" by a reversed reference direction axis. The other possibility is to avoid this complex conditional by first sorting out the orientation of each IfcWallStandardCase entity, as in Figure 36, and then label in a data dictionary its lowest value point as start point and highest value point as end point. In terms of coding and logic both solutions work about the same, but the second option may increase readability. The first option is adopted however, because knowing what IFC considers as the starting point of a wall is useful in other operations, as the starting point in IFC also determines the placement of the wall in its context.

If the points where a wall starts and ends at the point cloud file match the points where it starts and ends in the IFC file with the possibility of reversed axis considered, and within a certain threshold, the walls can be considered to be matched. A threshold is given due to possible imprecisions in point cloud measurement, differences in the definition of where a wall starts or ends between IFC file and segmentation method, or acceptable deviations from point cloud data and as-designed data. Walls that are matched can be classified as verified, and walls that did not find a match fall under two scenarios: the first scenario is an IfcWallStandardCase instance that did not find a match in the as-is state – it needs to be deleted as it does not exist in the current version of the building. The second scenario is a point cloud wall that found no IFC wall match. For that case a new wall needs to be added into the IFC file, either manually or automatically. An automated report produced by the script developed is shown in Table 1, with walls that need to be added and the extents that determine where this new wall(s) start, and the end coordinates. In the normal procedure, all IfcWallStandardCase elements are compared to each point cloud wall to see if they match, and a list is produced of matched point cloud walls and matched IFC walls. In Room Mode, discussed in more detail in Section 3.7, only the as-designed IfcWallStandardCase instances within the scanned area, with a small tolerance, are compared to the as-is point cloud walls. This makes sure that as-designed walls that fall outside the scanned area are not deleted, because they could not find a match in the scan or be properly checked.

IFC Wall Name	IFC Wall GUID	Status			
Basic Wall:Interior - Blockwork 100:299288	0TTniTe3HA5PhZy2pD3QrA	Matched with wall3			
Basic Wall:Interior - Blockwork 100:299380	0TTniTe3HA5PhZy2pD3Qqc	Matched with wall2			
Point Cloud Wall Name	Matched IFC Wall	Coordinates to build new wall, if needed			
wall1	No match found	Start: [-0.089985, -3.529509, -0.155652], End: [4.932769, -3.529509, -0.155652]			
wall2	0TTniTe3HA5PhZy2pD3Qqc				
wall3	0TTniTe3HA5PhZy2pD3QrA				
wall4	No match found	Start: [-0.079281, 0.109614, -0.155652], End: [4.932744, 0.109614, -0.155652]			

Table 1 : In this automated check's report, two out of four point cloud walls did not find a match, and two did. All the IfcWallStandardCase entities found in the as-designed model were checked. The result is that two new IFC walls should be added to the model, and their starting and end points are given. Those points are used as starting point of the automated update procedure, but can also be used by a modeler for manual interventions.

3.3.2 Update of walls

3.3.2.1 Deleting unnecessary walls

The first step taken in updating the general layout of walls is deleting the IFC walls that were not matched, and their decompositions such as doors, windows and voiding elements assigned to the wall. This is done because new walls will be added into the IFC model, and for these new walls, relations will be established to elements around it, and to avoid creating relationship entities to walls that will be excluded anyways, the deletion of outdated walls is performed first.

From the previous step where walls are checked to see if they are present in the real building, a list of IFC matched walls is produced by the script. The next step will then be to remove all IFC walls that are not in this list, or all IFC walls from a specific region of the model, in the case of room mode, further described in Section 3.7. In the specific case of Room Mode, an explicit list of walls to be deleted is made, composed of IFC walls that are within the scanned volume that were not matched to point cloud walls. This list of walls to delete is then directly fed to the functionality that deletes walls. If the principles of Room Mode are not used, all IFC walls in the model that are not in the list of IFC matched walls are deleted. In a way Room Mode performs the same function as this other normal mode, but in more cases and with more flexibility, being slightly more computationally expensive.

In order to not have an inconsistent model with decompositions missing their host, all decomposition elements such as windows and doors contained in those to-be-deleted walls also need to be deleted. They are in fact deleted first, before the deletion of the walls, as they are assigned to walls and retrieved in the script as a decomposition of the walls. The deletion is done of both walls and building elements contained in them, and voiding elements where those sub-elements are found, attributes such as the connections that wall had to other walls, geometry description of the wall, among others are also deleted. Entities that describe for instance all the items contained in a floor, will also make a mention to that previously existing wall, and they need to be edited, but not deleted, to remove just the items that are not valid in the project anymore. Those operations can be performed in python using the new IfcOpenShell API, and are quite straightforward using the proper methods.

3.3.2.2 New IFC wall generation and preservation of semantics

After the not-matched IFC walls are deleted from the original IFC file, the non-matched point cloud walls need to be built in IFC – that is, every wall from the real building that is not yet, or is in a largely different state, present in the old IFC file. This is a quite complex process, where new entities will be created in IFC with all the necessary properties and semantics, and this process takes data from the point cloud walls, from the previously existing IFC model itself, and the list of IFC walls matched and point cloud walls matched. The list of walls to be created is done by checking which point cloud walls are not in the list of matched point cloud walls. Then, for each point cloud wall, based on their geometry, a new IFC wall will be created. The general structure of the steps taken in wall update is the following:

1.Creation of a new OwnerHistory for the new walls that will be added into IFC;

For each point cloud wall not matched:

2. Check the region around the wall-to-be-created to look for a wall of similar characteristics, such as thickness, to use this wall as a semantic template to create the new wall, if no good match is found close by, search the entire model for the best fit;

3. Create a global ID for the wall and classify it as horizontal or vertical;

4. Create an ObjectPlacement with all needed entities – IfcLocalPlacement, IfcAxis2Placement3D, IfcCartesianPoint, and if it is a vertical wall, IfcDirection for Axis' DirectionRatios and RefDirection;

5. Create Shape Representations, one for the axis of the wall, and another for the body of the wall, with all needed IFC entities, and populate them with geometric information from the point cloud;

6. Create other entities and definitions necessary for the wall, using as much semantics as possible from the template pre-existing wall (such as assigning materials, visualization properties, etc);

7. Assign new wall into the right floor and fix its z-coordinate, that until here was a global coordinate, making now reference to the floor the wall is located. Adjust wall height if applicable to match its surroundings (e.g. if there is only a difference of less than 40cm.);

8. Improve the geometry (starting point and length) of the wall to have smoother connections to other previously existing walls around it. This involves also updating shape representations;

9. Improve the geometry of the wall to have smoother connections to new walls just generated by the updating tool, that are around this new wall. This is done after all new walls were already created and tentatively improved;

10. Add connections of type IfcRelConnectsPathElements to establish connections to all walls that connect to the start or end of a wall, make sure that connections between two walls are created only once.

One of the first steps will be the creation of an owner history entity for the walls that will be created, specifying the moment they were created, to distinguish walls that were previously present at the model, and walls that were updated, and mark their status as "modified" with the date of creation of those specific new walls. Attributes such as IfcPerson and IfcOrganisation can also be personalized depending on the need. After that a copy of one of the walls present in the model can be made, to use them as a semantic template, and replace the properties that are not the same, mostly geometry. This copied wall should be a wall from the same floor, and a check is placed to know which wall should be copied if there are several wall types present in the project. A check is done to find a template IFC wall around the area that the new wall will be created by looking for an IFC wall of similar thickness. If no such wall is found, the entire floor is searched for the best match.

To this wall, a global identifier is then assigned, of a similar type to other walls. Then, depending on the wall type (horizontal, vertical and diagonal, discussed at the previous Section) a different set of geometrical operations is performed to obtain and assign the properties of the IFC wall to be created. Just like when IFC and point clouds were matched to each other, horizontal and vertical walls have different reference direction axes. A reference direction axis will have to be created for a wall if they are vertical (0.0, 1.0, 0.0), as seen in Figure 36, and if it is horizontal RefDirection is set to None.

When copying an example IFC wall to use it as a template, a simple copy of the wall entity is made using IfcOpenShell's API. The entities that represent the geometry of the wall are created one by one, and where possible the semantics of the template wall are used. Also to avoid unnecessary creation of entities, many entities make reference to the main directions defined at the top of the IFC file, such

as (0.0, 0.0, 0.0) or (1.0, 0.0, 0.0), so a reference is made to those directions via the template wall, instead of creating new identical direction entities.

The implementation of the creation of the ObjectPlacement for a vertical wall is shown in Listing 1. The snippet also includes the operation of adding the placement of the wall in its context, based on the base point of the point cloud wall (represented as one of "wall_properties"). Later in the procedure, however, this base point will be adapted. The third coordinate of the wall's base point location, its vertical coordinate, will change from a global coordinate perspective into a local coordinate, relative to the storey the wall is located. This will happen right after the correct floor is assigned to the wall, then its z-coordinate base value will be the same as other walls in its floor (likely 0.0 or close to this). The x and y coordinates of the base point may also be slightly changed later, to smoothen out the geometry of wall connections, compensating for measurement imprecisions.

```
new_placement = model.create_entity('IfcLocalPlacement')
new_placement.RelativePlacement = model.create_entity('IfcAxis2Placement3D')
new_placement.RelativePlacement.Axis = model.create_entity('IfcDirection')
new_placement.RelativePlacement.Axis.DirectionRatios = (0., 0., 1.)
new_placement.RelativePlacement.RefDirection = model.create_entity('IfcDirection')
new_placement.RelativePlacement.RefDirection.DirectionRatios = (0., 1., 0.)
new_placement.RelativePlacement.Location = model.create_entity('IfcCartesianPoint')
new_placement.RelativePlacement.Location.Coordinates = wall_properties['base
point']

new_wall.ObjectPlacement = new_placement
```

Listing 1: Creation of the ObjectPlacement of a wall in IFC based on point cloud data, for vertical walls.

In Listing 1, for the Axis and RefDirection entities, the DirectionRatios were inserted as a tuple. The same happened to Coordinates, but if a single coordinate has to be changed, it cannot be changed as just an index e.g. Coordinates[2] = 3.8, a new tuple has to be assigned as the complete parameter, with the value to be changed and the other values that stay the same present in the assigned tuple e.g. Coordinates = (Coordinates[0], Coordinates[1], 3.8), where the first two values are just the pre-existing values, and the third value is the value we want to change in the location. When we create an IFC entity that is populated by more than one IFC entity, we also cannot assign them through indexes, but should initialize the tuple of the parent entity, and then assign the children entities as tuple additions to the current children of the entity, as in Listing 2, that shows the shape representations of a IfcWallStandardCase, that usually takes more than one shape representation.

The two IfcShapeRepresentation entities represent the basic geometry of the wall and therefore are the IFC entities that receive the most information directly from the point cloud data. The first representation is that of the axis of the wall. This information can be extracted from the point cloud using the method discussed in Section 3.4.1, depending on the wall type. A polyline is made to represent the wall's Axis. The first point is (0.0, 0.0), and the second one is (x, 0.0), where x is the wall length. The second representation of the wall's shape is that of its body. As IFC represents shapes as solids whenever possible, the wall is defined as the extrusion of a rectangle. The center of this rectangle is located along the center line of the wall, at the middle of the wall's length. The dimensions of the rectangle are the length of the wall (its XDim) and its thickness (its YDim). What IFC calls "Depth" of the wall is its height, that can be obtained at the point cloud as the difference between the maximum a minimum z-coordinates.

```
# Wall Representation
new_representation = model.create_entity('IfcProductDefinitionShape')
new_shape_representation1 = model.create_entity('IfcShapeRepresentation')
new_shape_representation2 = model.create_entity('IfcShapeRepresentation')
# Initialize the Representations attribute as an empty tuple
new_representation.Representations = ()
# Add the new items to the tuple
new_representation.Representations = new_representation.Representations +
(new_shape_representation1,)
new_representation.Representations = new_representation.Representations +
(new_shape_representation2,)
#The representations are filled with more entities + data about the wall's geometry
...
new_wall.Representation = new_representation
```

Listing 2: Initialization and assignment of children entities as tuples into an IFC entity

Some entities that are usually connected to a IfcWallStandardCase, such as the IsDefinedBy properties of a wall, contain information such as shading property sets for the visualization of the wall, wall type, and several generic properties of a wall that are not really unique. They can thus be copied from an existing wall, the template wall, but the process of assigning the new wall into those properties presents some complications when using IfcOpenShell and python. IfcWallStandardCase instances usually have 5 IfcRelDefinesByProperties and one IfcRelDefinesByType items connected to it by a IsDefinedBy relation. Pre-existing semantics of the model can be used to enrich the semantics of the new wall by assigning the new wall as one of the RelatedObjects of those six definitions. When looping over those items in python, using IfcOpenShell, the index of some items is reshuffled in an order that impedes adding the new wall as a RelatedObject to all items, and adds it two or even three times to some of them, as seen in Figure 37.

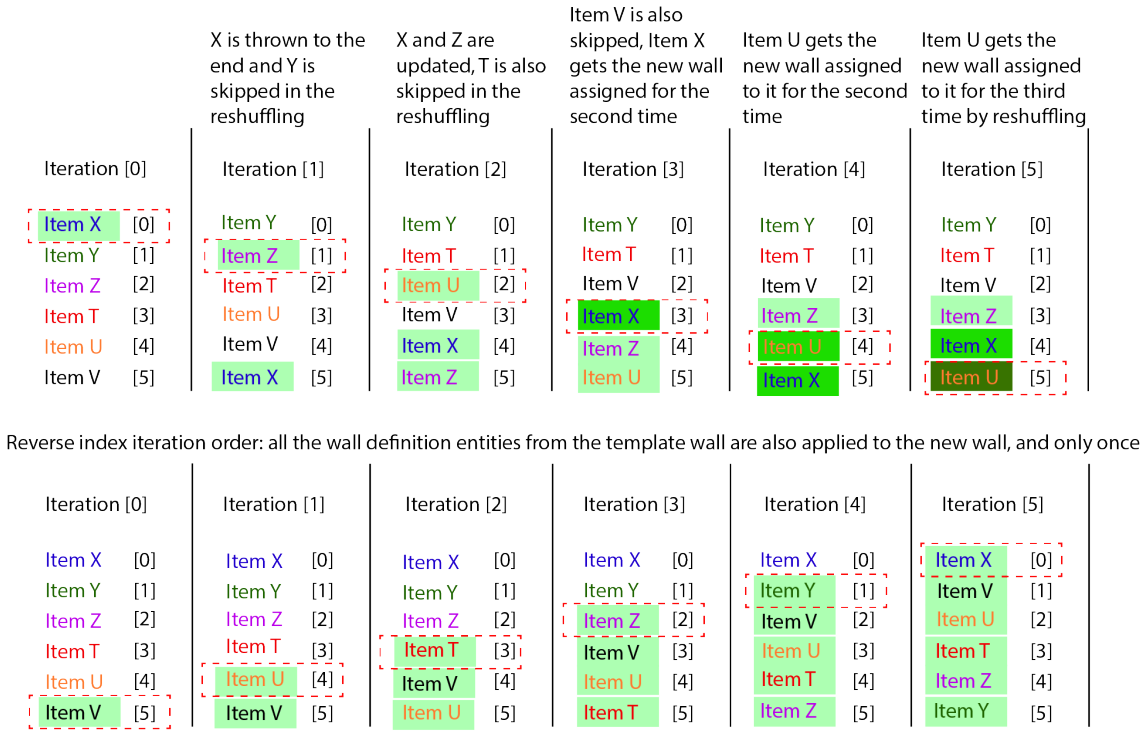

Figure 37: How indexes are reshuffled after each iteration for IsDefineyBy items. This problem can be solved by looping the indexes in the reversed order, as shown in the lower side of the scheme.

This problem is solved by using the reverse order of looping indexes, that is, going as [5], [4], [3], [2], [1], [0]. This is necessary because after each iteration the item iterated receives the last index after being iterated, reducing the index of the items that were at an index that was not yet iterated, as can be seen in Figure 37. This causes the loop to skip some items, but by reversing the looping order only the items that already were updated get their index shuffled, and every iteration brings a new item that was not updated yet. to still be iterated have their order reshuffled, because the previous item iterated receives the last index

After those basic geometric properties and attributes are assigned to the new wall, so that it has the basic information it needs, the final details that make it consistent with the IFC model are assigned, and some advanced refinements are done to improve the consistency of its geometry. First a check is done among existing IFC walls in the model (remembering to not include any wall previously generated by this process), and based on the z-coordinate of the starting point of the point cloud-based new wall (its level in the global space), a check is performed among those existing walls in the model to find one that has the same global Z coordinate value, within a threshold. This has to take into account that the new point cloud-based IFC wall, until that point, will have its wall.ObjectPlacement.RelativePlacement.Location.Coordinates[2] attribute (the z coordinate of the base point) as a global coordinate value that came directly from the point cloud, which can be e.g. -3.2, 3.5, 10.2, or any value related to the floor the wall is in. The existing IFC wall, however, will have the z-coordinate of its base point around 0.0, as its height is only relative to the floor it is contained in, instead of being referenced as a global coordinate. The new wall is not yet contained in a floor. So the z-coordinate value of the base point of the new wall has to be compared to the level of the floor the previously existing “template” wall is contained in. Only walls represented by a IfcRectangleProfileDef are used as reference to the new wall. If the starting points of the new wall and the existing “template” IFC wall are comparable in the z coordinate, it means they are on the same floor and the new wall is assigned to that floor. The new wall also gets the same z-value from the existing “template” IFC wall assigned to the z-coordinate of its own starting point. The precision of the point cloud can cause variations in the extrusion height of the walls generated, so a test is performed to see if the total height of the new point-cloud-based wall is comparable to that of walls around it, e.g. if it is within 30 or 40cm the new wall receives the same extrusion height from the existing walls. This is done to level the walls so that they end at the same height if they should, but if the new wall is a pony wall it still has its short height preserved.

After all new walls are created, assigned into the right floor and have the z coordinates of the location of their starting points adjusted, have their wall height adjusted, and having previously had their relevant basic geometric and semantic information assigned to them, it is time to create connections between the walls. This is left for the moment where all walls already exist because we want to check all the possibilities of connections a wall has, which could not be possible if some new walls were still to be created. The entities created are of type IfcRelConnectsPathElements, and each entity connecting two walls has their own global unique ID, which can be generated in IfcOpenShell in python using ifcopenshell.guid.compress(uuid.uuid1().hex). IFC has three types of possible wall connections, ATSTART, ATEND and ATPATH. Each one of those will be expressed for related and relating element, depending on the starting or ending point, or path, of a wall. ATPATH connections will not be created in the procedure developed, as the definition of wall adopted for segmentation and IFC files is of “short” walls that connect only at their start or end point, as discussed in Section 3.2. To create those connections four tests need to be done: if another IFC wall connects to the new wall

being looped, this other wall may have its start (1) or end (2) close enough *to the start of the new wall*, or its start (3) or end (4) close enough *to the end of the new wall*. Here it is important to know what IFC considers as the start and end, which can be counterintuitive due to the sometimes reversed-axis reference directions, but to assign a connection properly, as happening at the start or at the end, and according to the IFC schema, it is important to follow it as the IFC file had defined start and end. The new walls created by this procedure always have the starting points on the "lower" x and y coordinates, and the ending point at the "higher" x and y coordinates, as previously explained, however. To do this comparison, there is a function that finds the lowest distance between the end of a given IFC wall and the start of this given IFC wall, to the start of the new wall. If either of those distances is within a small threshold of the start of the new wall, then a check is done to see which one is closest, if the start or the end of the other IFC wall are closer to the start of the new IFC wall. Similarly, if it is to the end of the new IFC wall that the other IFC wall being tested is very close, by a small threshold, then a test is done to see which part of the other wall is connected to the end of the new wall. For many projects a threshold of 25 cm is good to find a connection, but if very thick walls are present in the model a bigger threshold can be needed.

The steps described above produce a sound IFC wall (or set of new walls) that represents the information contained at the point cloud and enriches this geometric information with semantic information from the model. Use is also made of heuristics to determine whether walls are close enough to each other to form a connection or close enough in height to apply the same height for all of them. Some imprecisions of scanning or movement sensors (especially mobile sensors contained in smartphones), and inherent deviations between the real building and the digital model, as well as small imprecisions in the alignment of point cloud and as-designed model, can make that the new walls generated in this process have slight misalignments to the as-designed IFC geometry around them. Depending on how walls connect, BIM authoring tools might define the exact start or end of a wall in a seemingly arbitrary way, as shown in Figure 38. This also might generate a difference between what is the start of a wall in a point cloud and what is the start of a wall in an IFC file.

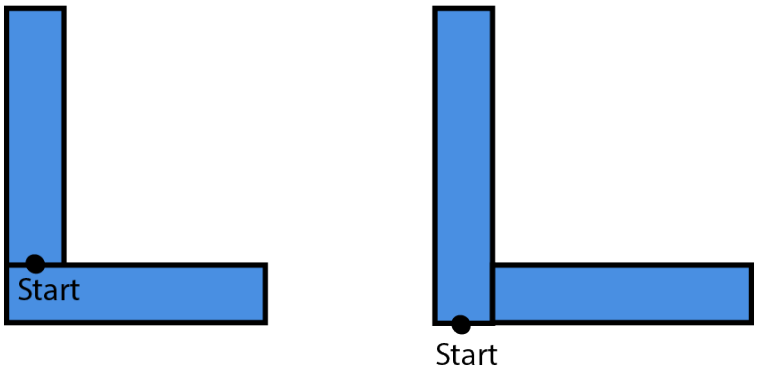

Figure 38: For the same configuration of a corner connection in the real world, IFC or solid geometry authoring tools might have arbitrary positions of the starting point of a wall, such as the starting point of the vertical wall above, that can be right for both alternatives shown

All of those factors can generate wall corners that have visually misaligned geometries and end in an dented corner. And example of a new wall created by an early stage of the procedure developed, that displays this problem, is shown in Figure 39.

Figure 39: wall with dented corners due to measurement imprecisions and conflicts with determining where a wall starts or ends in a point cloud versus IFC paradigm

A redefinition of a wall's starting point on the x and y coordinates and the length of those new walls can be used to correct this geometry. Because changing the starting coordinate of a wall also changes the end coordinate if the length is not changed, first all new walls are aligned to walls around them in their starting points, and only after that the length is changed to try to find an alignment at the corners around the end point as well. Evidently, the end coordinate is not changed (as it is not even explicitly defined in IFC, relative to the storey), and because moving the end coordinate would also move the start coordinate that had just been adjusted, so the length is changed instead. An algorithm is developed to perform those adjustments, and has four cases that handle adjustments of the starting point, and after those four cases are tested on all new walls, there are other four cases to adjust the length of the wall and deal with smoothening the corners at their ends as well.

The algorithm is structured as a function that first classifies the new wall to be improved in vertical or horizontal (which can be done using the reference direction of the wall). After that, the pre-existing walls of the model are iterated across this new wall, and a check is done to see if a given wall being tested at a time is at the same floor as the new wall, as they could be close in the x and y axis, looking connected, but being in different floors. After that, the existing wall being looped and checked against the new wall is also classified as vertical or horizontal.

Out of a total of 8 cases that check for IFC walls around the new wall, there are four cases that check for walls around the start of the new wall. Out of those four cases, there are two cases for when the new wall is horizontal and two cases for when the new wall is vertical.

Case 1 and 2 are applied to check for nearby walls at the start of the new wall, if the new wall is horizontal. An initial test is performed, to see if any of the walls close to the start of the new wall are also horizontal, if so the alignment of the new wall is made to that other horizontal wall, as a horizontal-horizontal alignment keeps the geometry more smooth. This means Case 1 has preference.

1. If the new wall is horizontal, and the wall being compared is also horizontal then the starting point of the wall will be set as the same point as either the start point or the end point of the "old" pre-existing IFC wall nearby, whichever one of the two is very close to the start of the new wall. In this way the walls become contiguous.

2. If the new wall is horizontal and the wall being compared is vertical, then the y coordinate of the starting point of the new wall is set to the same as the "old" wall. The x coordinate of the starting point of the new wall is shifted to the x coordinate of the starting point of the old wall minus half the thickness of the "old" wall, moving it in the negative x direction. This ensures that the face of the start of the new wall matches the left face (in floor plan view) of the vertical old wall, making the geometry of the corner smooth. This process is shown in Figure 40.

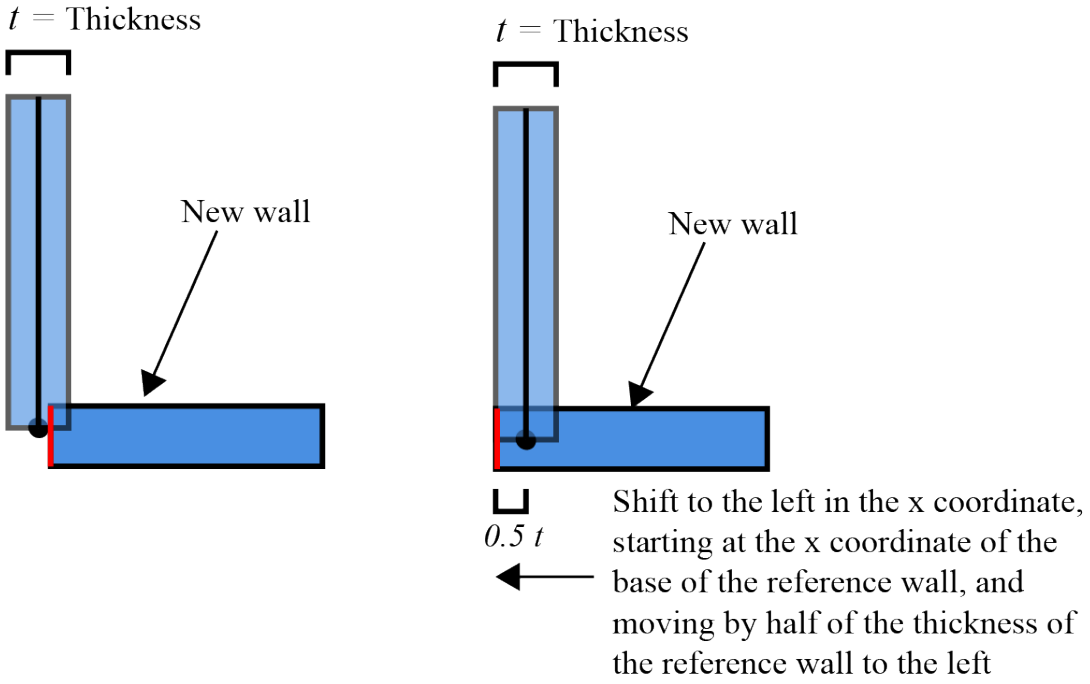

Figure 40: process of adjusting the position of the start of a horizontal wall to smoothen its connections with other walls

This is a simplified solution that creates some overlap inside the walls, a complete solution could be created taking into account that the "old vertical wall" here might be above or below the horizontal wall, and in both cases the point above or below can be either a starting point or an end point. This can be figured out analysing the reference direction of the old wall and the point (start or end) that is the closest to the new wall, which will orient the decision of increasing the y-start point coordinate of the new wall by half the thickness of the new wall plus the y-start point coordinate of the old wall, or reduce half of the thickness of the new wall from that value. This would work for this simple case of connections between two walls, if there are contiguous walls in the vertical or horizontal direction more complexity, tests, cases and exceptions need to be added to the algorithm. As a generalization, this simple method with overlapping works quite well and demonstrates that the alignment can be done, and could be further improved with more complex algorithms.

Cases 3 and 4 are applied to finding walls nearby the start of a new wall, when the new wall is a vertical wall. Similarly to horizontal walls, an interim check is done to look for walls nearby the start of the new wall that also have vertical orientation. If any is found, preference is given to align the geometry of the new wall into other vertical walls, so Case 3 has preference for vertical walls.

3. If the new wall is vertical and the "old IFC wall" around it is also vertical. Similar to the case where both walls are horizontal, if the starting point of a vertical wall has either the start or end point of another wall close to it, the starting point of the new wall will be set as either the start or end point of the old wall, whichever is closer to the start point of the new wall, to aid geometry continuity.

4. If the new wall is vertical and the old wall is horizontal. Here we take into account whether the horizontal walls around the vertical new wall are on the left side or right side of the new wall, and whether the point that is close to the new wall's starting point is also a starting point of the "old" IFC wall nearby or an end point. If the nearby IFC wall is at the right side of the new wall, the new wall is moved to the right, at a position equal to the start coordinate of the old wall plus half the thickness

of the new wall, moving in the x direction. If the "old" wall is at the left side of the new wall, the new wall is moved to the left, at a position equal to the start coordinate of the old wall minus half the thickness of the new wall, in the x direction. The rendition of this algorithm (cases 1 to 8) is available in Appendix VI, at the end of the wallUpdaTor file.

All new walls have their starting point adjusted, and then a new check if performed across all new walls at their end point, to update their length and smoothen the corners made by connections at the end, in cases 5 to 8. In cases 5 to 8 the new wall being updated is similarly classified as vertical or horizontal, and then all walls previously present at the model are also classified as vertical or horizontal, and each one is compared to this new wall being tested, to see if their start or end point are very close to the **end point** of the new wall. Because the starting point of the new wall was probably changed in the previous cases, and the length is not changed it (will be changed now), it can be possible that the wall end is now more distant from the walls it should be connected than it was before cases 1 to 4 being applied. The threshold used to check proximity of neighbouring walls to the end of the new wall should therefore be higher than the threshold used to search for walls at the start of the new wall.

Here again Cases 5 and 6 will deal with the end connections of new horizontal walls, and Cases 7 and 8 will deal with the end connections of new vertical walls.

5. This case deals with a new wall that is horizontal and has a horizontal wall connected to its end. First a check is done to see if the end or start of the neighbouring wall is closer to the end of the new wall, and then the length of the new wall is set to the distance between the starting point of the new wall, and the end or start point that touches the end of the new wall, making them contiguous.

6. This case deals with a new wall that is horizontal and has a vertical wall connected to its end. Because the starting point of the wall was already defined in the previous step, and we do not want to change it and mess up the connections at the start, we cannot shift the new horizontal wall in the y direction, only make it longer by changing its length and therefore changing its ending point in the x axis. Figure 41 exemplifies the change that has to be made: the new wall, as shown on the left side, cannot align with the left face of the neighbouring wall to avoid overlap as it would generate (or keep) a dent and irregularity in the connection. Without either redetermining the layout of all walls at the same time, or adjusting the starting and end positions of "old" walls from the as-design stage, which would be an iterative process that would drastically increase the complexity of the algorithm, the best solution is having some overlap, and moving the end face of the new wall to align it to the right face of the vertical wall that neighbours it, as shown at the right side in Figure 41. This is done by setting the length of the new wall as the difference between the start point of the new wall and the starting or end point of the old wall (whichever is closer) – which will align the end face of the new wall, shown in red at the figure, to the middle of the wall close by – and then add half the thickness of the neighbouring wall to this distance.

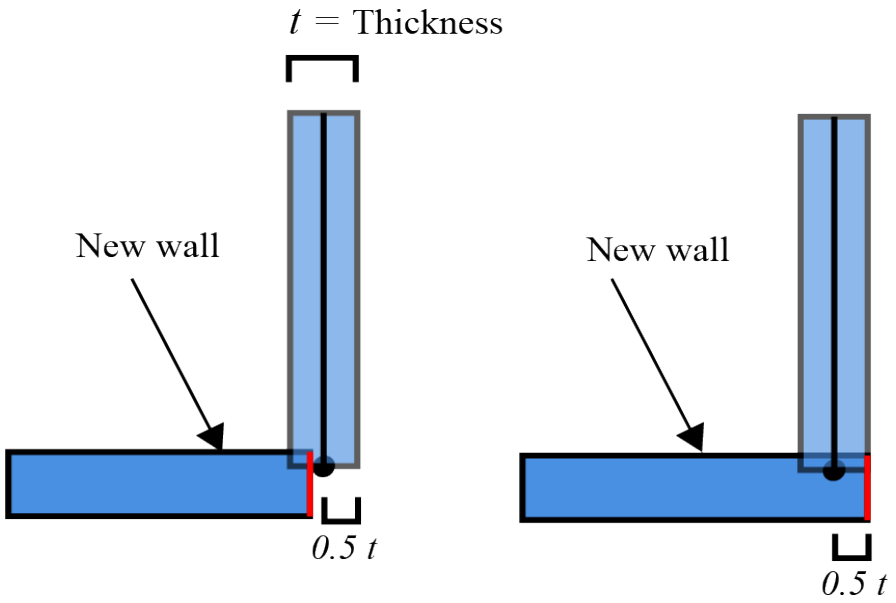

Figure 41: Altering the length of a wall to smoothen the corner of a connection

7. This case deals with the scenario where both the new wall and its neighboring wall are vertical. Then a check is performed to determine whether the end of the new wall is closer to the start or the end of the neighboring wall, and then the length of the new wall is set as the distance between the starting point of the new wall and this closest point of the new wall, making them contiguous.

8. The last case deals with a vertical new wall connected to a horizontal wall. In this case the distance between the y coordinate of the starting point of the new wall and the y coordinate of the starting point of the pre-existing wall that neighbors the new wall's end is calculated, and half the thickness of the neighboring wall is added to this distance, setting the new length of the new wall.

This process yields smooth corners at the connections of walls, which changes the geometry seen in Figure 39 into the improved corner geometry shown in Figure 42.

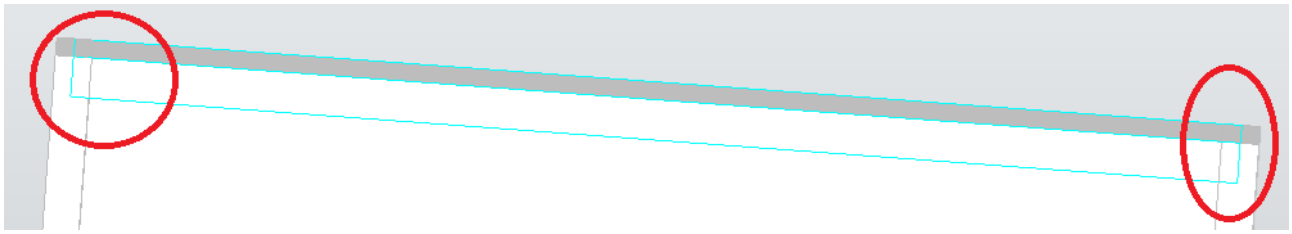

Figure 42: Updated corners with smooth geometry and no indentation, seen in the OCC viewer

After the geometric improvements to match the as-designed geometry, those cases are also applied to improve the geometry of new walls to match the possible connections that might exist to other new walls. One of the tests done in the implementation of the tool involves a complex wall arrangement of newly generated walls, all created by update by point clouds, and is described in Section 4.2.

3.4 Match and update of ceilings

Besides the extensive update of walls, a demonstration is also proposed of how ceiling elements and column elements can be updated in IFC. This section will discuss the methods used in the update of ceiling elements, more specifically the update of the height of ceiling elements, for IfcCovering entities defined by both an IfcRectangleProfileDef and by an IfcArbitraryClosedProfileDef.

The recognition of points that represent a ceiling in a point cloud, to segment them, is often done using z-coordinate histograms, as discussed in the literature in Section 2.3.2. The input used for ceiling update discussed here is a segmented ceiling, in .xyz point cloud format, according to the standard described in Section 3.2.2. Starting from those segmented ceilings, the 3 essential steps in model update defined in Section 3.1 (step 4 to step 6), extraction of geometry, matching of elements and update of IFC model can be broken down into the following steps for ceiling height update:

Open all segmented ceilings, and for every segmented ceiling:

4. Extraction of geometry

4a. Find the mean value of z to find the estimated point cloud ceiling height;

4b. Find p1, the center of gravity of the segmented ceiling, to locate its approximate center;

4c. Create 8 points, p2-p9, in a square around the center of gravity, within a certain buffer of e.g. 60 cm. They will be distributed as: p2 = cg + (0.6m in the y direction), p3 = cg - (0.6m in the y direction), p4 = cg+(0.6m in the x direction), p5 = cg - (0.6m in the x direction), p6 = cg + (0.6m in the x direction) and + (0.6m in the y direction), p7 = cg + (0.6m in the x direction) and – (0,6m in the y direction), p8 = cg – (0.6m in the x direction) and + (0.6m in the y direction), and p9 = cg – (0.6m in the x direction) and - (0.6m in the y direction);

5. Matching IFC as-designed geometry and segmented ceilings – for each point cloud segmented ceiling go over all IFC ceilings to find a match:

5a. Find the global height of each IFC ceiling to compare it to the point cloud ceiling;

5b. Find the bounds of the IFC ceiling in global coordinates, transforming the local coordinates found at the entity to test whether a match is found. This transformation is done based on the local coordinates of the IFC ceiling and the reference direction of the ceiling;

5c. Check if at least 6 of the 9 points mentioned in steps 4b and 4c, present at the point cloud ceiling, are within the projected area of one of the IFC ceilings. A buffer in the z-coordinate of the point cloud ceiling is done to check for ceiling entities above or below it;

6. Updating IfcCovering elements

6a. If step 5c's check is positive, the ceilings are considered matched, and the IfcCovering entity in IFC has its height adjusted to match the mean z-value of the point cloud found in 4a.

The matching of point cloud ceiling and as-designed IFC ceiling is thus done by checking whether most of the central points of a point cloud segmented ceiling fall within the area of some of one of the ceilings present at the IFC file. The reason to chose for more central points in order to match ceilings is that if points at the border of the ceiling were chosen, geometrical imprecisions or differences in how the ceiling is defined in IFC can make the match less accurate. Another reason to match a point cloud ceiling to an IFC ceiling, instead of an IFC ceiling to a point cloud ceiling, is that

ceilings can be large elements that span several areas of a building in case of e.g. a corridor. If only a limited area of the building is scanned, instead of the entire building, and part of a corridor is scanned, it is possible to know that the scanned part of a ceiling has a certain height, and it is part of a given IFC ceiling. The entire area, or a majority of the area of this scanned ceiling will fall within the area of one of the IFC ceilings. But the very large IfcCovering ceiling entity in IFC will not find a very good match with the segmented ceiling as it might match only e.g. 25% of its area. This approach helps thus with updating the height of a ceiling even without a complete scan of a building. The dataset chosen to test ceiling update using the principles underlined above is a section of the Ca building at the Erasmus Medical Centre in Rotterdam, The Netherlands. Part of one corridor of this building was scanned, in a section that has two different ceiling heights, and, in the IFC model, two ceilings defined by different profiles too, one defined as an IfcRectangleProfileDef and the other one as an IfcArbitraryClosedProfileDef profile. In this way, this small dataset can be used to test the update of different ceiling heights and different ceiling types. Figure 43 shows the scanned area, and Figure 44 shows the simplified IFC model used for testing, based on the larger IFC model of the entire building.

Figure 43: Section of the entrance of the building Ca from the Erasmus MC

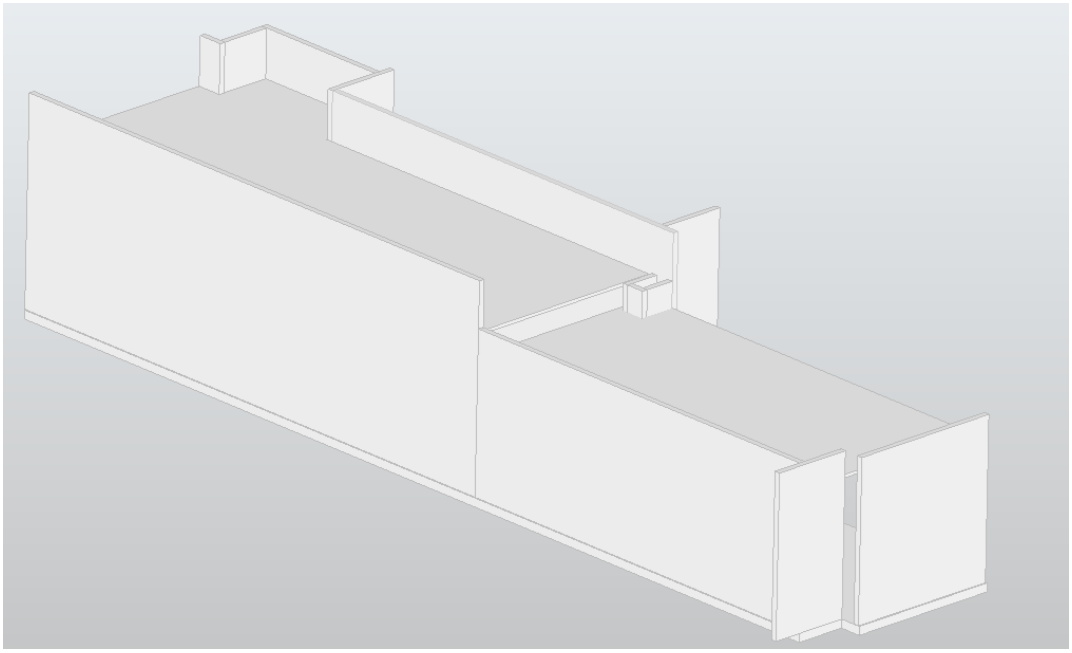

Figure 44: IFC model with one ceiling defined by a rectangle, and another one by a polygon

Step 4 is followed in the same way for ceilings defined both by an IfcRectangleProfileDef profile, and by ceilings defined by an IfcArbitraryClosedProfileDef profile. So is step 5a, where the global height of ceilings is found in order to compare their height under the same coordinate system to the point cloud ceiling's height. IFC expresses an IfcCovering's height relative to the IfcBuildingStorey it is located in, so to find its global height we need to add the height of the IfcCovering to the level of the IfcBuildingStorey. After the global z-coordinate of the IFC ceiling is found, step 5b looks at the definition of the IfcCovering's profile, to convert the bidimensional information from that plane and find the x and y coordinates of the ceiling's profile in global coordinates, in order to compare IFC ceiling and point cloud ceiling. This is done in two different ways, one for IfcRectangleProfileDef profiles and another for IfcArbitraryClosedProfileDef profiles.

Finding the boundaries of an IfcRectangleProfileDef-defined ceiling in global coordinates

In the ObjectPlacement properties of a ceiling, its relative placement to the building storey is given. This relative placement is the center of gravity (CG) of the rectangle, for ceilings with their profile defined by a rectangle. In the geometric representation properties of a ceiling, that detail the profile, the dimensions are given as local coordinates for the ceiling, instead of being relative to the building storey. The convention is to, locally, have the largest side of the ceiling aligned longitudinally to the local x-axis. The side of the rectangle that is locally aligned to the x-axis, determines the XDim of the ceiling's profile, the dimension on the x-axis of the ceiling. The dimension of the profile on the y-axis of the ceiling is named YDim, and usually is the shortest side of the rectangle. If the orientation the ceiling has in the definition of its profile is different than the orientation it has in the coordinate system of the IfcBuildingStorey, it will have a RefDirection of (0.0, 1.0, 0.0) (assuming a Manhattan-World assumption), and its surrounding points will be calculated differently than the case where the local orientation is the same as the orientation at the building storey. Figure 45 shows the differences in how the bounding points are calculated, if the axes are rotated the x-coordinates receive influences from YDim and the y-coordinates receive influence from XDim to find the four points that surround the rectangle.

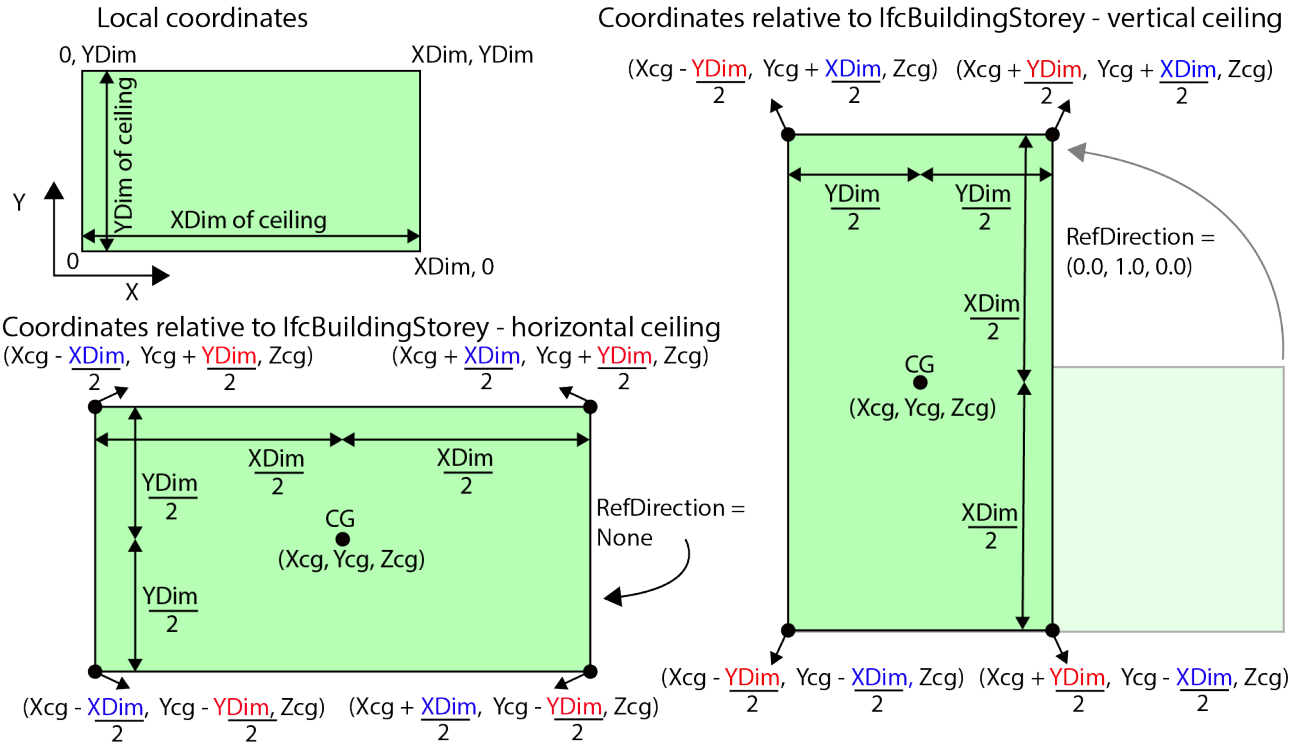

Figure 45: Calculation of the boundaries of rectangular ceilings, vertically or horizontally oriented

To find the values in x, y and z that limit the area, to later check if the values of the point cloud ceiling fall within those ranges, we can use the formulation shown in Figure 45 to find minimum and maximum values for x and y. In the case where RefDirection is (0.0, 1.0, 0.0), the minimum value of x shall be X_{cg} – (YDim/2) and the maximum value of x shall be X_{cg} + (YDim/2). The minimum value of y shall be Y_{cg} – (XDim/2) and the maximum value of y shall be Y_{cg} + (XDim/2). If RefDirection is None, making the orientation of the slab relative to its own geometry description the same as the orientation of the slab in the building storey, the minimum value of x shall be X_{cg} – (XDim/2) and the maximum value of x shall be X_{cg} + (XDim/2), and the minimum value of y shall be Y_{cg} – (YDim/2) and the maximum value of y shalle be Y_{cg} + (YDim/2). The minimum and maximum values of z are the same for both scenarios, and can be considered as a buffer value that creates a volume around the IFC ceiling, to test if the point cloud ceiling can be matched to it. A value of 0.5 above and below the value of Z_{cg} was adopted, but this parameter can be customized depending on the standard heights of ceilings used for the projects that need to be updated. As long as the z-buffer values are not too high that an IFC ceiling of the wrong building storey could be used for the check, they can be increased.

After the ranges for x, y and z coordinates are determined, the 9 points of the point cloud ceiling, described in step 4b and 4c, can be checked to see whether at least 6 of them fit within the projection of the IFC ceiling described above. If so the average z-value of the point cloud ceiling is used to update the IFC ceiling's height, remembering to reduce the IfcBuildingStorey's height from the global value of the point cloud ceiling's height. Then the IFC ceiling can have a new updated height, that is also relative to its building storey.

Finding the boundaries of an IfcArbitraryClosedProfileDef-defined ceiling in global coordinates

In ceilings defined by an IfcArbitraryClosedProfileDef, the placement, or center point, of the ceiling, relative to its storey, is also given in the ObjectPlacement property. The profile of the ceiling is however not defined by a rectangle, but by a closed curve composed of points. Those points compose a polygon, that extruded, forms the geometry of the ceiling. The representation of the geometry of the ceiling will usually still consider the ceiling's largest dimension as aligned to the x-axis in the local representation of geometry. The coordinates of the points that compose the curve that defines the layout of the ceiling are given referencing the distance of each point, in the x and y axes, to the center point of the ceiling, in the local coordinate system of the ceiling's geometrical representation. Relative to the building storey, however, those coordinates might have to be transformed, when the ceiling has a reference direction. In a Manhattan-World scenario, when the RefDirection is not None, it will likely be (0.0, 1.0, 0.0), meaning the coordinates are rotated in the direction of the y axis. When RefDirection is None, no complex coordinate transformations are needed, and obtaining the coordinates of each point that composes the bounding polygon of the ceiling can be done by adding the x coordinate of the polygon point to the x coordinate of the center point of the ceiling, and adding the y coordinate of the polygon point to the y coordinate of the center point of the ceiling. When RefDirection is (0.0, 1.0, 0.0), the conversion of coordinates of each point is influenced by the 90-degree counter-clockwise rotation: the local x-axis is rotated to the y-direction, and the local y-axis is rotated to the -x-direction (negative x) at the building storey. This is shown in Figure 46. In this way, x-coordinates of each point in the polygon are added to the Y-coordinate of the center of the polygon, to find the y-coordinate of that point in global coordinates, and y-coordinates of each point in the polygon are subtracted from the X-coordinate of the center of the polygon, to find the x-coordinate of that point in global coordinates. This process is visualized in Figure 46.

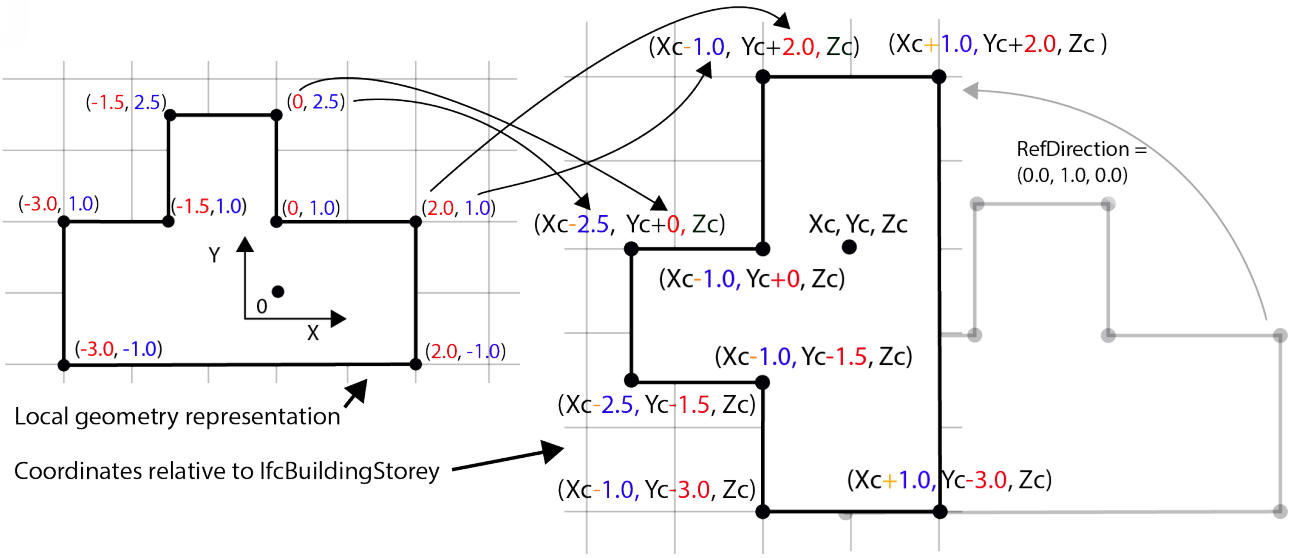

Figure 46: Coordinate transformation to find global coordinates of an IfcArbitraryClosedProfileDef-defined ceiling, with a RefDirection of (0.0, 1.0, 0.0)

After the polygon that defines the profile of the ceiling has its points found in global coordinates, a buffer in the z-axis can be created above and below this polygon, to create a volume where a test will be performed to see if the point cloud ceiling can be matched to this IFC ceiling. The shapely library in python can create a polygon with the points found, and test if the 9 points of the point cloud ceiling defined in steps 4b and 4c are within the projection of the IFC ceiling. If more than 6 points are matched, the ceilings are considered as matched, and the height of the IFC ceiling can be updated based on the height of the point cloud ceiling. Here again, the level of the IfcBuildingStorey is subtracted from the height of the point cloud ceiling, to allow the new IFC ceiling height to still make reference to the building storey it is located in.

3.5 Match and update of columns

Besides the update of walls and ceilings, the third element chosen to have its update in IFC, based on point cloud geometry, was the IfcColumn. Columns are common building elements, are an important part of the structure of a building (together with beams and slabs), and present one particular challenge that has to be addressed: how to deal with elements that may or may not be encased inside other building elements. A number of columns might be present in a building, and some, or all of them, are visible in the spaces of the building, while some of the columns, or even all of them, might be encased in walls or other such geometry. We are left with a difficult situation that complicates a complete unsupervised automated update: the scanned data may check and identify columns at the building, but no matter how well the laser scan is performed, some columns might not be detected by the scan. In some buildings, all columns will be detected, however. This generates uncertainty. A similar scenario happens with elements such as MEP elements – in some buildings they are visible, in some buildings they are encased, and a mixture of both is also possible. Those building elements require thus extra supervision of the involved professionals when their update is being performed, in the case of columns also because they are important for the structural integrity of a building. But that does not mean that the automated update of those elements in IFC cannot be performed, or that the updating process cannot be aided, and time be saved, by semi-automation. This section will discuss how this can be done. Following the steps described in Section 3.1, used for the update of IFC models by point cloud-based geometry, the process of IfcColumn entities update can be broken down in:

4. Extraction of geometry from point cloud

4a. Assuming a Manhattan-World assumption the minimum and maximum x-coordinates are found within a segmented column, and the minimum and maximum y-coordinates are found, and the median value of them should be the center point in x and y coordinates. The center-point z-coordinate is taken as the lowest z-coordinate in the segmented column. This works for symmetrical profiles. Column height is also calculated;

5. Matching IfcColumn entities and point cloud columns

5a. Every IfcColumn goes under a test to see whether it might be embedded in a wall or not. For this, IfcWalls are also parsed. A list is made of IfcColumn entities that are very close to walls, possibly being embedded in them, and a list of IfcColumn entities that are not close to walls;

5b. After that, parsing each point cloud column, a threshold of 1.1m is used to see if any IfcColumn can be matched to it. A distinction is made between point cloud columns that got matched to an embedded IfcColumn, and point cloud columns that got matched to IfcColumn entities far away from walls;

6. Update of columns

6a. The IFC columns that are not embedded in a wall and got matched to point cloud columns are updated to the right position. If an IfcColumn is not embedded in a wall and was not matched to any point cloud column, it is deleted. If a point cloud column is not embedded in a wall and was not matched to any IfcColumn, a new IfcColumn will be created in the model at its position. IFC columns that are matched to point cloud columns but embedded in walls are kept track of for reporting, but are not updated. A check is also done to see how many IFC columns are embedded in walls but did not match any point cloud columns, and those IFC columns are not deleted, as they could not be checked;

6b. A notification is generated, with a possible warning in case IFC columns are found embedded in walls and are not matched to any point cloud data. Another warning is made if the number of point cloud columns in the area that is distanced from walls is smaller than the number of IFC columns in that area – that would mean that out of the columns that have the highest chance to be detected, to few were found.

The geometry extraction step starts with point cloud files, each one representing a segmented column. The position of the column is extracted as described in step 4a. After that, a check is done to classify all IfcColumn entities in embedded in a wall or not embedded in a wall. It is taken into account that some columns have their location, relative to the building storey they are in, located in the ObjectPlacement property of the column, while other columns have their location at the MappedRepresentation property's position. The coordinates that locate the column might be located at the MappedRepresentation when there are several columns of the same type, that possibly were modelled as a copy of each other, and then the same column type entity is repeated, one new entity for each column, with a location for the column in its MappedRepresentation. One simple way to find at which one of the two locations the center point of the column is, is checking which one of the two properties has coordinates that are not (0.0, 0.0, 0.0). After locating each IfcColumn, the check is done to see if their center point is not embedded in any wall. With a Manhattan-World assumption, it is assumed that walls will have either a horizontal or vertical floorplan projection. If they are vertical, their x coordinates, relative to the building storey, stay the same, and the wall grows in the y direction. With a tolerance for partial embedding of e.g. 35cm, the ranges to classify the column as embedded in a vertical wall will be $x_{wall} \pm 0.35$, with x_{wall} being the x coordinate of the centerline of the wall, and $y_{min} = y_{wall\ start} - 0.35$, and $y_{max} = y_{wall\ end} + 0.35$. In this way, a ribbon is created around the centerline

of the wall to check for embedding or partial embedding of columns, if the x and y coordinates of their center fall within the specified ranges. If the wall is horizontal it is the y coordinate that will stay constant along the wall, making the range for embedding $y_{wall} \pm 0.35$, with y_{wall} being the y coordinate of the centerline of the wall, and $x_{min} = x_{wall\ start} - 0.35$, and $x_{max} = x_{wall\ end} + 0.35$. A list is then made for IFC columns that are embedded in walls, and another list is stored of IFC columns that are not embedded in walls. Figure 47 exemplifies how columns can be classified as embedded in walls, which is when their center point falls within the "area of influence of the wall" just defined. Notice that column 1 is embedded but can still be detected, while column 2 cannot be detected or matched.

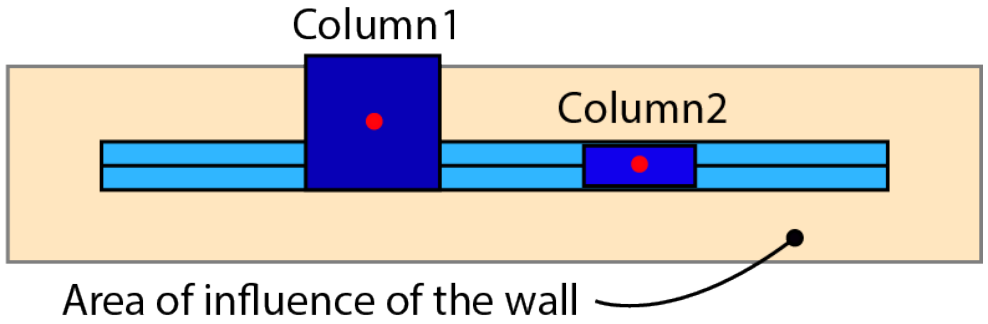

Figure 47: Different scenarios where a column might be embedded in a wall

When a list of IFC columns embedded in IFC walls is made, a check can be made to see which one of those IFC columns embedded in walls can be matched to point cloud segmented columns. In many circumstances, columns that are embedded in walls cannot be detected by scanned data, if they are fully embedded. However, in some cases depending on the column's profile, if part of the column protrudes out of the wall, or if the column is not really embedded but just very close to the wall to the point it is detected as embedded, the geometry scanned of the column might still be enough to detect it and segment those points as a column. Therefore, some embedded columns might be visible and matched while others, or all of them, might not be visible and matched. Because a partially embedded column might be matched, but not all of its geometry might be assessed, the choice is made to not update the position of matched embedded columns, as embedded columns always have a degree of uncertainty to them. The quantity of embedded matched and unmatched columns is however taken into account when comparing the total of columns found at the scanning process of that area of the building, and the total of IFC columns that were in the IFC file of the building. If less columns are found when surveying the building than the total of columns in the original design, the structural integrity of the building might be compromised, which asks for special human attention and intervention in the check and update process. To aid finding those inconsistencies, a special warning is made with the amount of IFC columns found that were embedded in columns and not matched to any point cloud geometry. To match IfcColumn entities and point cloud entities a threshold of a radius of 1.1 meters is used, based on the center of the point cloud column, to see if an IfcColumn is within that area. The threshold might be changed, but columns are usually not less than 1.1 meters apart, and if they are much more distant than that of each other, it is harder to ascertain that they are the same column. Embedded IFC columns that are not matched to point cloud data are *not* removed from the model, as they could not be properly detected and then checked.

After the embedded IFC columns are checked, a check is made with all the point cloud columns that were not matched to an embedded IFC column, to see if they match a non-embedded IFC column. Here again a threshold of a radius of 1.1m is used to see if the center point of an IFC column can be found around the center point of the point cloud column. If a match is found, the position of the IFC

column matched to the point cloud column is updated, based on the coordinates of the point cloud column. If the point cloud column finds no match, a suitable pre-existing IFC column is used as a template, that is, a copy of it is made and the location is set as the location of the point cloud column. After all point cloud columns are checked, the IFC columns that are not embedded in walls and were not matched to any point cloud column are deleted from the model. Algorithm 1 shows a simplified version of the algorithm that updates the IfcColumn entities' positions.

```
Algorithm 1: Column update
import ifcopenshell
Assign to model ifcopenshell.open('ifc file')
Assign to columns model.by_type('IfcColumn')
Define coordinateFinder(pointCloudColumn):
    Assign to pointCloudColumn open(pointCloudColumnFile)
    Create list for x coordinates; Create list for y coordinates
    For line in PointCloudColumn:
        Get first value and append to list of x coordinates, get second value and
        append to list of y coordinates
        End For loop
    Find average of x between xmax and xmin Find average of y between ymax and ymin
    Print coordinate of Column center point
    Return coordinate of Column center point as tuple with third value as z min
    End function definition
Define matchColumns(PointCloudColumnCenter):
    For column in columns:
        Find IfcColumn center point using IfcOpenShell
        If the center point of the point cloud column is within 1.1 meter of the center
        point of the IfcColumn:
            If the IfcColumn's center coordinate is not embedded in any IfcWall:
                Assign IfcColumn and point cloud column as matched to each
                other
                End If statement
            End If statement and function definition
Define updateColumn(PointCloudColumnCenter):
    For column in columns:
        Find IfcColumn center point using IfcOpenShell
        If center point of the Point Cloud column and the center point of the
        IfcColumn are within 5cm of each other:
            Skip do not update and store the two columns as matched
            End If statement
        Else if center point of the Point Cloud column is within a 1.1m radius of the
        center point of the IfcColumn's center point:
            Assign IfcColumn's center point as the center point of the point cloud
            using IfcOpenShell
            Print new center coordinate of IfcColumn to check
            End If statement
Execute defined functions for a dictionary of point cloud columns

Write new IFC file using IfcOpenShell
```

Algorithm 1: Script of the algorithm that can update the position of columns in an IFC file based on point clouds, based on the centroid of the column

Figure 48 shows the application of the script into the same room 8.304 from the Atlas building, of the previously exemplified case study described in Section 3.2.3, where the as-designed place of the column is much more distant from the wall than in the real building. When collecting data at the building, it is noticeable how much the column is closer to the wall compared to the available IFC model, it was even difficult to scan the two faces of the column that are closest to the wall and closest to the façade (at the left side in Figure 48). So this is a real case from a real building where there are deviations between as-is geometry and available IFC model, and was one of the reasons that inspired this research. Receiving as input only the IFC file, and the segmented column from the point cloud, the script automatically outputs an IFC file with the correct position of the column.

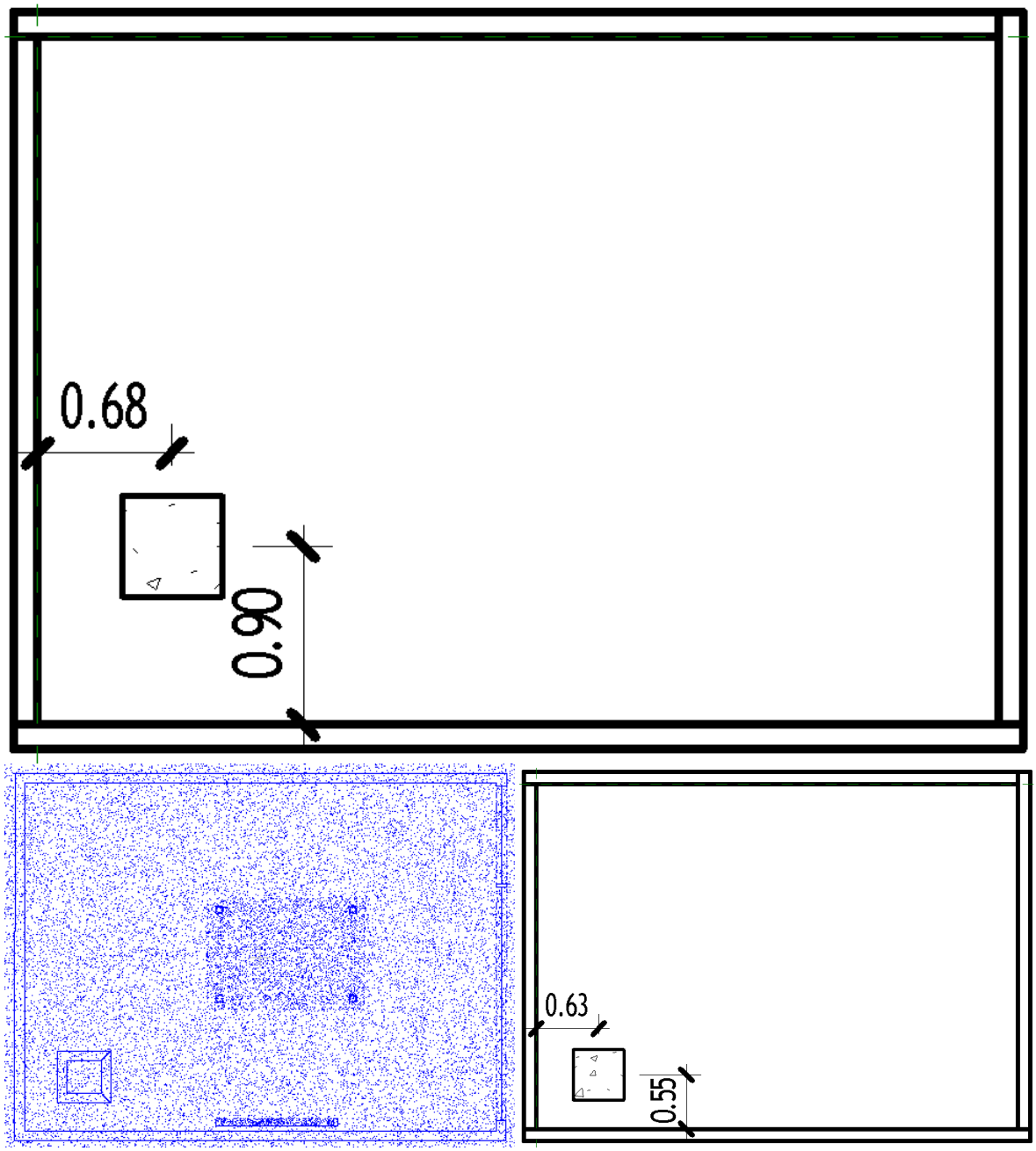

Figure 48: Above, the original IFC file with a column in the wrong position, seen from above as a floor plan. Below and to the left, the point cloud collected at the building showing the real column position, and to the right, the new IFC file, automatically generated based on the point cloud. The Column's name is Bart.

3.6 User interface

To make the method usable and accessible to a wider range of stakeholders, and aid visualization of the operations performed, a user interface is needed. Visualizing the process makes it more understandable, and gives cues to as-design and as-is differences when IFC model and point cloud are overlaid. Structuring the code around a user interface also helps the breakdown of the procedure into clear steps. Python was the programming language chosen to implement the procedure and create the python tool "Point Cloud to IFC Updater". Python is an accessible language to new users, widely used and a growing base of users, influence for instance by its libraries in data science. It is also one of the languages, besides C++, in which IfcOpenShell can be used, which is a toolkit that gives access to parse and edit data using the IFC schema. There are however, at the time of this research, no complete libraries that allow direct visualization of IFC files in python. IfcOpenShell can be installed with IfcConvert, however, which allows the easy conversion of IFC files into STEP format. This allows an implementation that uses the STEP version of an IFC file for visualization by use of the python OpenCascade implementation, and runs the IFC file directly in the background of the script.

Every big change brought into the IFC file that should be visualized is then converted into a STEP file which is visualized in the 3D viewer of the interface. The conversion of IFC into STEP is conceptually simple, and in small projects almost instantaneous, allowing a fluid interaction between the changes brought in the IFC file using IfcOpenShell on the background, and their visualization in STEP using IfcConvert and pythonOCC (OpenCascade), in small files. In files with complex geometry, especially files that have many surface representations of elements (instead of representations based on the basic solids), and that span a large area, the conversion can be considerably slower, as will be discussed in the Results Section (4).

Tool dependencies are python 3.10.14 and IfcOpenShell v0.7.0.231127. A python 3.10.14 environment was created in Anaconda where all necessary packages were installed, and Visual Studio Code was used as IDE. The interface makes use of several packages, many of which such as "os" and "math" are built in in python 3.10. "os" is used to edit and create filepaths and names for new files. "PyQt5" was used in version 5.15.10, and allows the OCC visualizer to have menus that give direct access to the folders in the computer to pick one or several files and load them into the procedure. PyQt5 is also used for pop-up messages and warnings, for instance. "OCC" was used in version 7.7.2, and is used for 3D visualization of STEP files, visualization of point clouds and general structure of the interface. IfcOpenShell is used to access the IFC schema and perform more easily changes in attributes, or create and delete entire IFC entities and even their ramifications. IfcConvert is used to convert IFC-STEP files into STEP format. "datetime" is used to create filenames with a date related to the time they were created, and to add date to a new OwnerHistory that will store the moment changes were performed in the project. "openpyxl" is used in version 3.1.2, and is used to create reports in excel .xlsx format, about matched and deviating elements. "Numpy" (version 1.26.4) and "pandas" (version 2.2.2) are used to retrieve information from the point cloud and use this data to match the as-is state represented by the point cloud, with the as-designed state present in the IFC file. "Numpy" is also used in Room Mode to downsample a point cloud and help make the processing of the concave hull more efficient. "alphashape" (version1.3.1), "mpl_toolkits" (version 0.4.0) and "shapely" (version 2.0.4) are used to create the alpha shape (a concave hull around the points) used in Room Mode. "matplotlib" (version 3.8.4) is used to visualize the alpha shape just mentioned, to

be sure that the shape being used was properly calculated and is similar to the outlines of the point cloud. “math” is built-in module, and is used to calculate distances in 3-dimensional space, when performing checks or updates in e.g. the wall update functionalities.

The interface is composed of a screen that is mostly occupied by the 3D visualization of the building project and point clouds, and menus on top of the screen that execute the commands outlined in the procedure. The interface follows the standard configuration of OpenCascade 3D visualization interfaces, with a display to visualize objects and menus on top. The first menu is used to open the IFC file to be updated. The second menu is a menu that allows visualization of point clouds, and one or many point clouds of a building or segmented elements can be opened at the same time. The other menus deal each one with one type of building element that can be updated, namely walls, columns and ceilings. The user should make sure that the steps necessary for an action were taken, that is, when an the submenu to update an element type is selected, both an IFC file and segmented point cloud elements need to be loaded previously. There is also a pop-up message when the interface is opened with basic instructions, telling the user this about the order in which submenus should be used, and which menu can be used for point cloud visualization.

The interface is shown in Figure 49, with a relatively large model loaded, and a point cloud of a section of the building being shown in overlay to the building geometry. The first menu on top “Open IFC and make STEP”, is therefore the first option that should be used, and by accessing it the user can pick an IFC file anywhere in their computer to be updated. The file is loaded as the model that will undergo several operations and updates using IfcOpenShell in the backend, and is shown as STEP geometry in the interface. The second menu, “Point Cloud”, allows the user to load point clouds (one or more), with the only purpose of visualization, that is, in other later menus, point clouds will also be loaded, but loaded into functions that use them to extract geometry and update the model. Here they are loaded just for visualization. A reason for that is that the user might want to visualize only a few elements, without over-cluttering the visualization with all point cloud information. Another reason is that the point cloud and the IFC file are at the same scale, but the STEP file retrieved from the IFC file has coordinates in a different scale, 1 million times larger than IFC. Therefore, point clouds used for visualization are scaled and should not be the same as those used for geometry extraction. The third menu is “Walls”, that presents the functionalities to update the walls of the building when a complete scan of all the entire floors available in the IFC file is provided. It has submenus to load the segmented wall files, which can all be loaded at once, and are automatically renamed if they follow an unstandardized name structure. There is a second submenu to check the as-designed geometry against the as-is point cloud geometry, that outputs an excel report about matched and unmatched walls, shown previously, and a third submenu that runs the update of the walls, deleting outdated walls, creating walls that should be at the model, and generating a new IFC file that is then shown at the interface as STEP. Room Mode Walls is the third menu, that executes a more refined version of the previous menu that updates walls, and it starts with the loading of the scan of the subsection of the building, computes a volume around it, and then, follows the same steps and submenus of the Walls menu. The check is performed only inside the volume, so e.g. in Figure 49, only IFC walls in the area in blue are compared to the point cloud walls, and only walls inside this volume could be deleted if a match is not found. There is also a menu for Columns and Ceilings, where their segmented elements can be loaded, matched to IFC elements, and the IFC file is then updated into the as-is state.

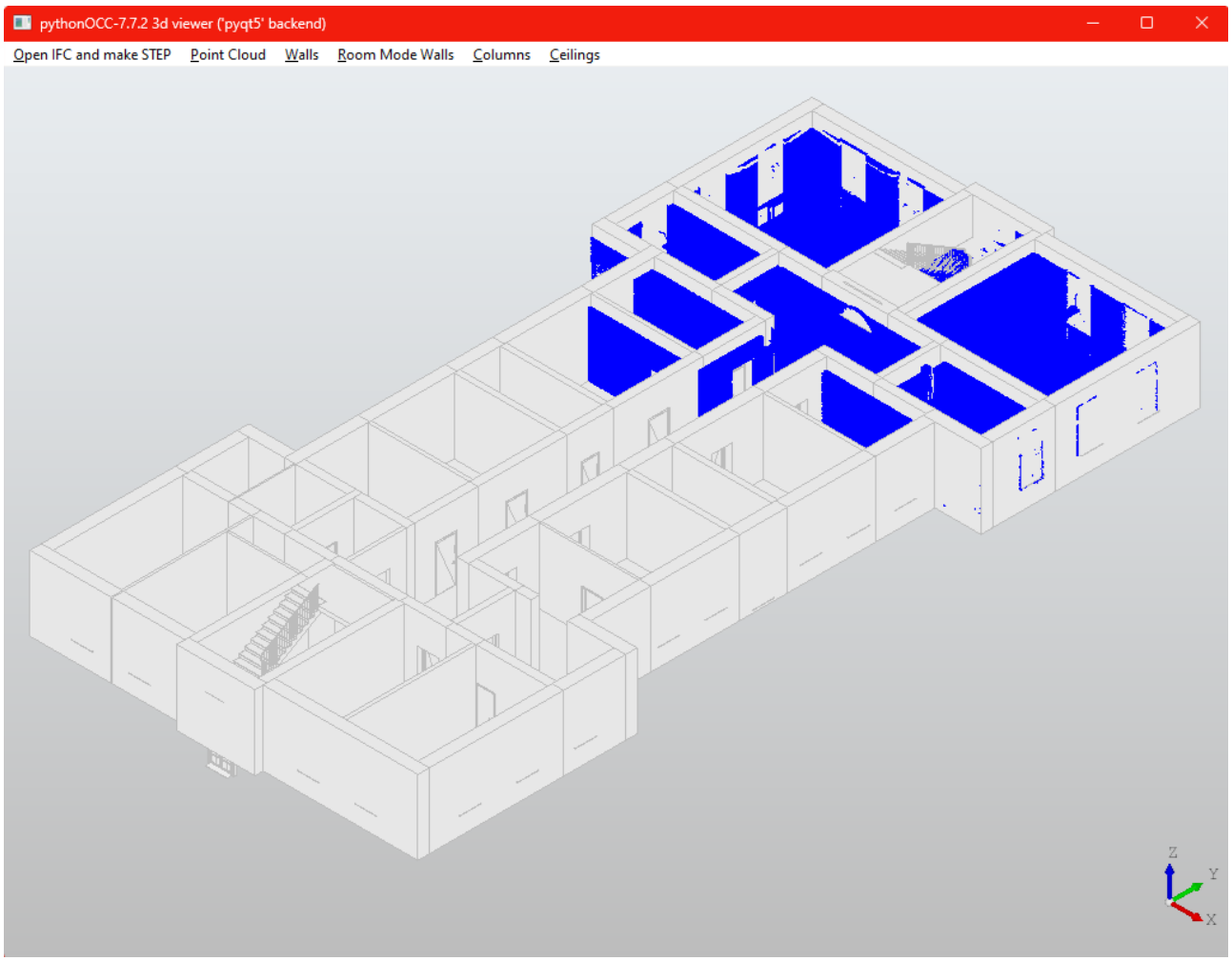

Figure 49: Interface developed with an IFC file being visualized as STEP, and a point cloud of a section of the building being shown in overlay to the building geometry

3.7 Room mode

The steps described above discuss how an IFC file can be updated by point cloud geometry when a complete scan is available. Whereas much of the current research is focused on creating complete new IFC files of buildings from scratch, which would evidently require a complete scan of the building, in a model update paradigm, this necessity of a complete scan is less evident. In fact, considering the costs and time involved in surveying a building, it can be concluded that only scanning a relevant section of the building, that e.g. underwent a renovation, would be much cheaper and faster, giving much more traction to the idea of updating building files by stakeholders such as building managers. Furthermore, if tools that are both cheaper, commonly found, and multifunctional, such as smartphones and tablets, can be used in this process of surveying, the process would become much more accessible, which could also aid the adoption of IFC file updates. Mobile scanning procedures done in smartphones or tablets, at the time of the research, often have a limited range of data collection, i.e. scanning an entire large building creates drift in point collection and imprecisions in the data, but they work well in smaller areas. This could also point to the use case of mobile smartphone/tablet scanners in updating models by scanning limited sections of a building.

The contrast between being able to perform a check in a relevant section of a model, instead of the entire model, lies in the problem that if point cloud data is not available for part of a building, but the

point cloud data is considered as the ground truth, against which all deviations are removed, then all the non-scanned areas would be deleted. If a local scan is performed, this is not a problem and only deviating elements that were properly checked are deleted, and data is also accurately reconstructed. This can be achieved by creating a concave volume around the area that was actually scanned, and performing checks of IFC as-designed elements only if they are contained in this area (within a small threshold of tolerance). This procedure becomes the menu of the interface called "Room Mode Walls", based on the principle of being able to check and update elements in just section of a building, instead of the entire building. Despite the name, any number of rooms or subsection of the building can also be used as input for Room Mode, even discontinuous scans, e.g. scans of different areas of the building that have no continuity. In this research, it was only implemented in the update of walls, but the same idea can be used for other building elements too.

A concave shape can be obtained in python using the "alphashape" library, for instance, aided by some other packages mentioned at Section 3.6, to perform operations that improve the calculation of the concave hull. For instance, the shape that encloses the point cloud should be based on the concave outline of the area scanned, under a floor plan perspective. That is, all the variations in height (z-coordinate) inside the point cloud are irrelevant for the estimation of the x and y coordinates where an element's starting and end points should or should not be, which is what is checked. The extra dimension of z-coordinates makes the computation of the concave hull extremely slower, so the point cloud can be "flattened" first, with the computation of the 2D outline of the concave area taking into account only x and y coordinates found in the point cloud. Then the area that represents the concave shape is just extruded to match the maximum value of z and the minimum value of z found at the point cloud, making the computation one entire dimension faster. The point cloud was also initially divided into voxels (volume pixels, division of the space into small cubes), and down sampled, to reduce the amount of points in each cube that does contain points, yielding the same outline necessary to compute the shape with less points, reducing necessary computation. Those two steps reduce the computation time by a factor of more than 30, in most cases, especially for very large files that had large point densities, and produce good results. A buffer is also applied to this concave shape using the package "shapely", to account for deviations in the project and imprecisions, but to still find elements that have their start and ending points within the area scanned. "matplotlib" is used to visualize the concave shape generated, and check if it matches the point cloud. Figure 50 shows an example of a point cloud of a section of the building that allows the check of walls to be performed only in this scanned area using Room Mode. Notice how the room in the lower corner at the right of the figure was not scanned, creating a "concavity" in the scan, and this room should therefore not be checked. Figure 51 shows the alphashape computed that matches with the scanned area and therefore will not check the walls of the room that was not scanned.

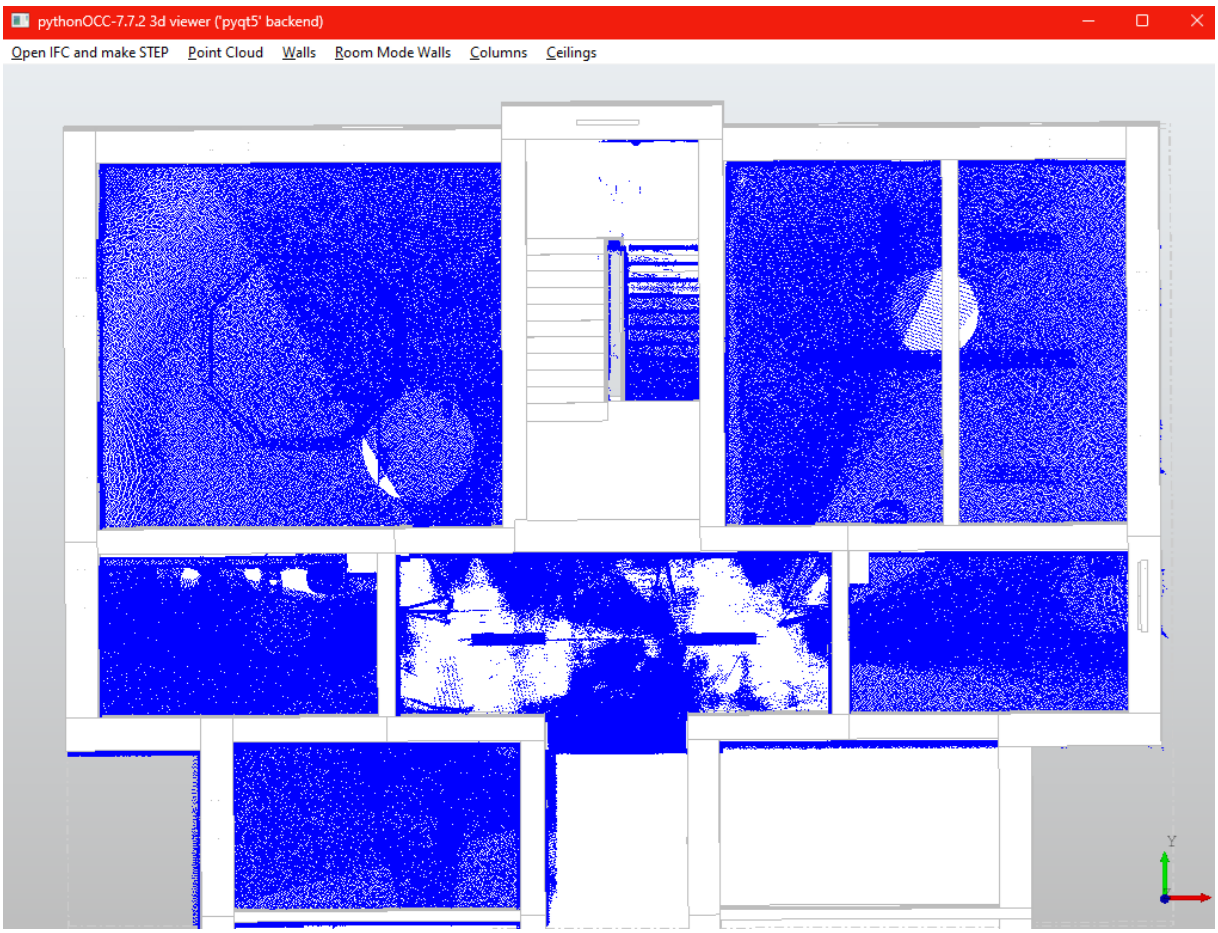

Figure 50: Scan of a section of the building that can be updated using Room Mode

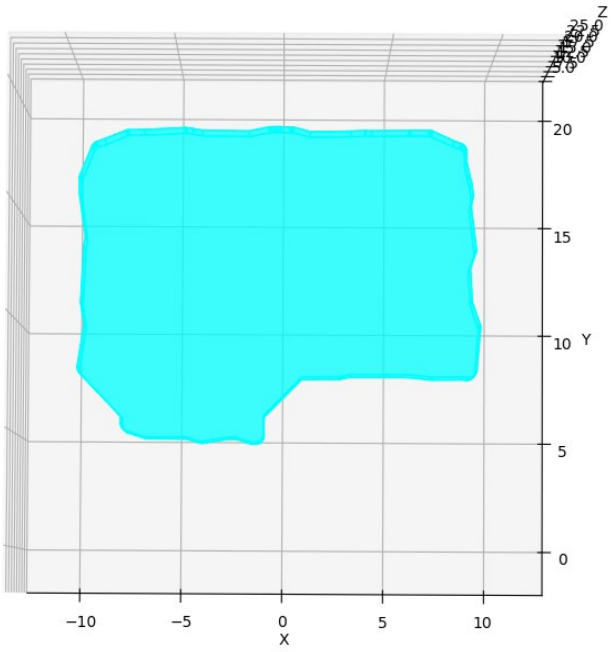

Figure 51: The concave hull generated for this scanned section, the check will skip the outside areas

One possible limitation of this Room Mode implementation, is that an automated matching of the coordinate systems of point cloud and IFC model (as described in Section 3.2.3) would become more complicated. If the scan covers the entire building it is much easier for a tool to find similarities and overlay both geometries. But if the area scanned is just a section of the building it is more computationally expensive to find a match, as it will be a match of a shape that matches a random

part of the other shape, instead of its entirety. Furthermore, some building typologies can contain many repetitions, such as rooms of similar or identical dimensions and similar openings, e.g. classrooms or hotel rooms. To solve this, some human intervention could be used, such as a simple interface implementation that allows the user to click on a room or area that was renovated, or the implementation of GPS data into the point cloud data (and also geolocation data in IFC) to match the location of the scanned area with a section of the BIM model.

3.8 Conclusions

This section discussed the principles used at the different stages of the update of an IFC file by point cloud geometry. The procedure starts with a pre-existing outdated IFC file, and then the following steps are performed: 1. Point cloud data is collected; 2a. Standards for segmentation are defined; 2b Point cloud data is segmented and labelled; 3. Point cloud data and the IFC file are set to the same coordinate system and have their geometries overlaid; 4. Geometric information is extracted from the segmented elements; 5. IFC and segmented point clouds elements are matched to each other, and 6. The IFC model is updated based on matched and unmatched elements. The focus of this research is the automation of steps 4 to 6, and the demonstration of how this process can be automated is done for IfcWallStandardCase, IfcCovering and IfcColumn elements. The walls of a model are updated by checking IfcWallStandardCase elements against scanned data, and deleting those outdated IFC walls that don't find a match, and creating new IfcWallStandardCase entities for point cloud walls from the real building that were not present at the IFC file. The new walls created are based on other walls existing on the model, use the same wall type and semantic characteristics, choosing the one that fits the best as a template. A "Room Mode" for walls is also introduced, where it is also possible to update walls in cases where only a section of a building is scanned, in that case the test is performed only in this limited volume. A methodology is developed to update the height of IfcCovering entities, both of type IfcRectangleProfileDef and of type IfcArbitraryClosedProfileDef, matching each point cloud segmented ceiling to one IFC ceiling, and then updating the height of that IFC ceiling based on the height of the point cloud ceiling. The update of IfcColumn entities is realized based on thresholds, and if two columns (one IfcColumn and one point cloud column) are matched to each other, within a maximum distance, the position of the IfcColumn is updated. If an IfcColumn is not matched to any point cloud column, and it is not encased in a wall, it is deleted. If it was encased in a wall, it is not deleted but a warning lets the user know how many as-designed columns could not be checked because they are embedded in walls. If a point cloud column does not find a match to an IfcColumn, a new IfcColumn is made for it. In the case of columns both creation, deletion and update of elements are dealt with, depending on the scenario, and reports based on undetectable elements are also made. An interface is created with a python implementation to demonstrate the principles outlined, test them, and serve as a proof-of-concept. The tool is named Point Cloud to IFC Updater.

4 Results and implementation

In this section, the results obtained testing the procedure developed are discussed. The section starts with Section 4.1 explaining the datasets used. The two files that start the procedure are the outdated IFC file that needs to be updated, and the point cloud scan of the building. To demonstrate and test the functionalities that are important for checking and updating each type of building element addressed in this research, different datasets were used for each building element. Section 4.2 deals with the update of walls. Tests need to be done to test the creation of vertical walls, the creation of horizontal walls, the deletion of walls, and the improvement of the geometry of walls. One room in the Atlas building of the Eindhoven University of Technology was scanned with a smartphone to perform tests of wall deletion, creation, and geometry improvements on a smaller scale. A small IFC file, based on a larger IFC file of two floors of the building, was created to represent the building accurately according to the larger IFC file. The other dataset used for wall update, both its point cloud and IFC file, was obtained from the DURAARK project's github repository. This is a multi-storey building scanned with a TLS. Section 4.3 discusses the update of ceilings, where only one dataset is used, that has two ceilings, with profiles of type IfcRectangleProfileDef and IfcArbitraryClosedProfileDef, and different ceiling heights. The point cloud was obtained scanning a section of the Erasmus Metical Centre. A small IFC of the scanned region was made based on the larger IFC files available. For the update of columns, whose results are dealt with in Section 4.4, two rooms of the Atlas building were used, the same one used for wall update, and another larger classroom. In the first room there was a real deviation in the position of a column, comparing IFC file and the as-is state. Some synthetic datasets were also produced to test features in column update such as how the procedure should deal with embedded columns.

4.1 Results of data collection

4.1.1 LiDAR data and best applications

To obtain LiDAR data tests were conducted with a six mobile apps, namely: Metascan, SiteScape, Polycam, Scaniverse, 3dScanner App and PIX4Dcatch. Among those, 3dScanner App and Scaniverse allow access to a good range of their functionality for free. 3dScanner App had good performance for scanning small items or limited areas of a building, but struggled more with buildings with tall ceilings. Scaniverse had better results compared to 3dScanner App in larger environments or environments with high ceilings, but still had some drift problems in rooms with a large ceiling. Metascan had limited results. SiteScape and PIX4Dcatch had the best results in terms of reduced drift, and correction of points collected. PIX4Dcatch has, besides its quality as in terms of uploading options, control of density of points to be used, also the possibility of using an RTK for professionals – a device that gets attached to the user's phone and increases the GPS signals by a large magnitude, improving point collection. SiteScape and PIX4Dcatch are expensive professional solutions, however, with costly memberships. All the applications described above, with the exception of Polycam, collect point clouds in a traditional LiDAR fashion, making no use of the RoomPlan API that can generate regularized point clouds with filled occlusions. Polycam has thus a good drift rate and allows access to those Room Plan functionalities, so data was collected with this application. The idea of testing Room Plan functionalities was leveraged to take advantage of point clouds with less occlusions that can be collected faster. Those corrections of occlusions are advantageous to the methodology developed – highly cluttered walls, that would normally have much missing data, now

can have a precise definition of their starting and ending points, that are then compared to those of the IFC file.

LiDAR was collected at the *Atlas* building of TU/e, in rooms 8.201 and 8.304. Data was also collected in the building *Ca* of the Erasmus MC Hospital, on Rotterdam, The Netherlands. Rooms with high ceilings present more challenges in collecting data using a smartphone, with more drift and less regularity of planes. But collecting data in small rooms is not always completely straightforward either, depending on lighting conditions, movement pattern when collecting data, and random errors, data can be collected in a room considerably fast or quite slowly. The RANSAC-styled approach of building layout reconstruction in the app allows the user to visualize the progress of the scan, which walls are already scanned or not. But sometimes the movement pattern might "confuse" the mapping of the environment, and an element that was already properly recognized, such as a wall or a door, leaves the mapped objects and needs to be scanned again, what is not always successful. Sometimes the user has to re-start their scanning process from scratch. Nevertheless, the process often works quite well and the new implementation of the RoomPlan API already works much better than the previous one, so it could be expected that it will keep improving.

As with many other LiDAR scanners, terrestrial or mobile, problems are often found trying to scan large windows or glass walls or façades, as the laser beams collect much data outside and a lot of noise in the data is generated around glass surfaces. When using a RoomPlan-based application such as Polycam, for instance, that tries to detect and reconstruct building elements and floorplans, this problem can be exacerbated, by the application trying to create an "infinite" wall continuation, following the end of the walls from the building, when pointed to a glass façade. The user is therefore advised to point their sensor as little as possible to a glass façade, and if done so point, it as parallel as possible, instead of orthogonally.

4.1.2 BIM data

The BIM data used was composed of IFC2X3 files encoded in STEP file format (but .ifc extension). The data used to kick-off the development of the procedure was a very small project, a simplified version of a section of the Atlas building, and once good results were obtained in this small scale, the procedure was tested, improved, and extended by implementing it in larger and realistic models.

Because the research aims to update an IFC model based on discrepancies found between point cloud and IFC file, most of the datasets used underwent intentional changes in the IFC files to simulate renovations or modifications in a building. Those intentional changes were tailored to generate operations required in IFC file check and update. Datasets were adapted to test the generation of new walls in different scenarios, including complex connections of new walls to other new walls, or deletion of walls and their decompositions, or matching of walls in a model with many wall types and thicknesses. For the update of ceilings, the IFC available used as as-designed was also an accurate as-built model, so the ceiling heights were changes to test and demonstrate their height update. In the case of column update, one dataset had a real discrepancy that could be updated, discussed in Section 3.5, and other synthetic changes were made to that IFC file (room 8.304) and the IFC file of room 8.201 to test deletion and creation of new columns, column position update, and reporting of embedded columns that were or were not matched. Those intentional changes were performed in Revit, exporting them as new IFC2X3-STEP files.

4.1.2.1 Small-scale controlled test

To test the updating features on a smaller scale, the first tests were done in small IFC files of rooms of the Building Atlas. One of them is Room 8.304. Apart from the original configuration of the room, some synthetic variations of this IFC model were created to further test the update of IFC columns and walls. Room 8.201, a larger classroom, was also used, only to test the update of columns. Figure 52 shows the iterations and datasets used to test the update of columns. Figure 53 shows the iterations of room 8.304 to test the update of walls.

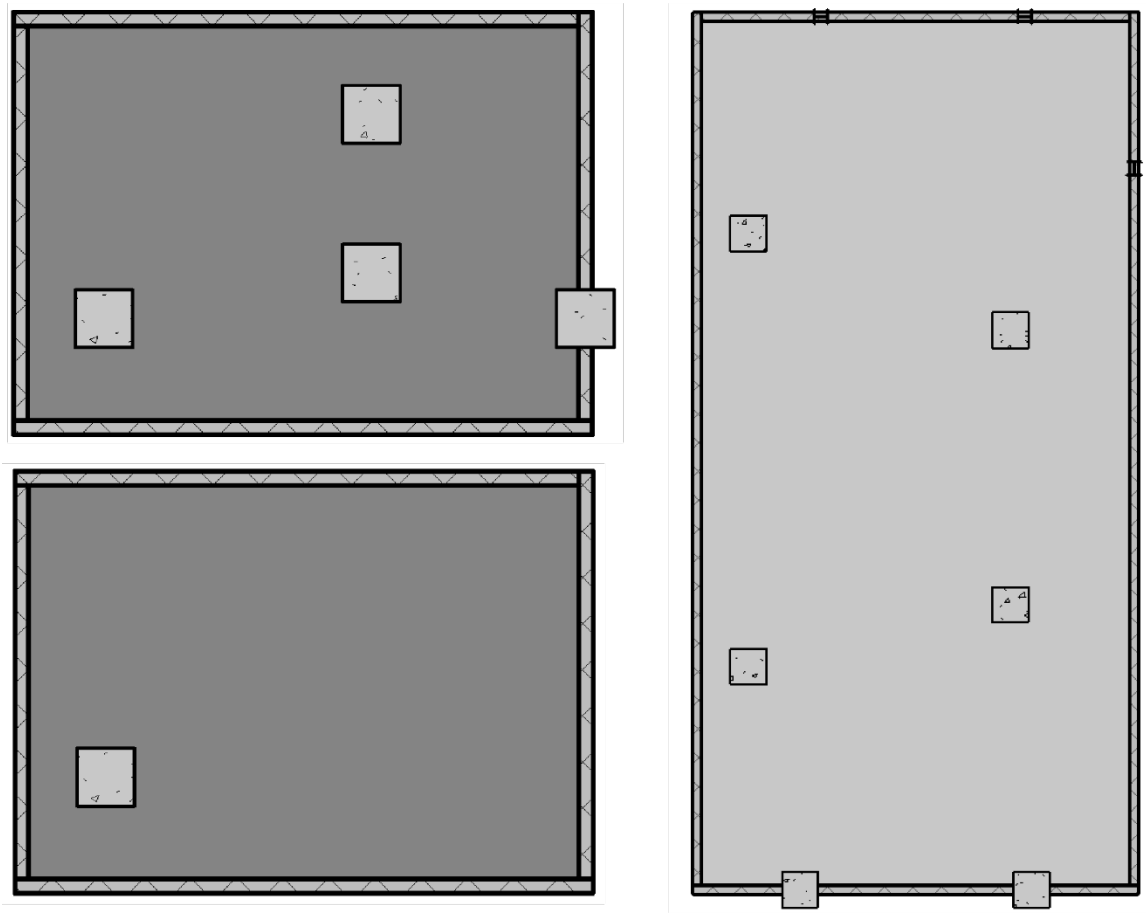

Figure 52: Above, to the left, an example with more colums, where one of them is partially embedded in a wall. Below, to the left, the original model of Room 8.304. To the right, Room 8.201 with partially embedded concrete columns below, and fully embedded steel columns above

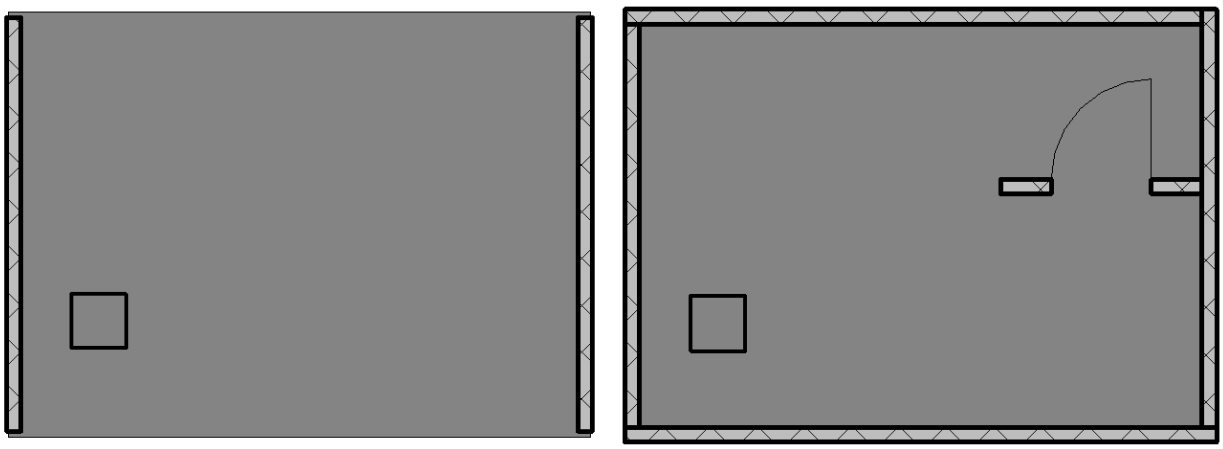

Figure 53: To the right, an iteration of Room 8.304 with two walls missing, prompting the creation of two new walls into the model. To the left, an iteration of the same room with an extra wall that has a door: here both the door (decomposition) and the extra wall should be removed

4.1.2.2 Bigger and more realistic test

After the basic principles of how the update should work were developed in the smaller files, a much larger dataset was used. While the previous datasets had one room and four walls – or five, depending on how "renovations" were simulated – this larger dataset has 84 walls as its ground truth, and can have new walls added or existing walls deleted to simulate changes, discrepancies and renovations. This dataset used is the "Haus30" dataset, available at the Github of the public European project DURAARK, and is composed of the IFC file of a multi-room multi-storey building, and the TLS laser-scan of the first floor. Even though the scan data is available for only one of the floors, the fact that the scanned floor is above ground floor, making it necessary for the comparison to access floor data in IFC, allows the tool to be tested in a multi-floor paradigm. This dataset was chosen to be used for the test of wall updates.

Where walls were not of type IfcWallStandardCase, changes were performed in the IFC file. This was done to walls that were modelled with arches, as not IfcWallStandardCase walls, they could not be edited and therefore updated as IfcWallStandardCase entities normally are. No basic change in layout was made, just a changes in wall types, remodelling the walls of special geometry as IfcWallStandardCase of rectangular profile entities. Walls were also split to fit the modelling standard for walls defined in Section 3.2.3, with no .ATPATH. connections. A ground truth file was defined for the IFC file and for the scan of the building, with every wall being attributed a number at the moment of its segmentation, and mapping this number to where the wall is in the IFC file on a floor plan. The segmented walls, segmented manually in CloudCompare to serve as ground-truth, are then saved in a folder, each wall with their right number. Later, several different tests can be performed by changing a copy of the IFC ground-truth file, and by picking a number of point cloud walls that the user wants to consider that would be found when scanning a section of a building. Figure 54 shows the number of each wall, attributed to the segmented walls, seen in floor plan view. As it can be seen, this dataset has several wall thicknesses, with external walls reaching 56 cm in thickness, presenting a good opportunity to test the choice of a template wall when automatically creating new walls.

By selecting only a section of the scanned point cloud, that can be a section with concavities, and not necessarily a box shaped section, as could happen in a real situation of a partial scan, many tests can be made for specific areas of the project. One way to produce point cloud sections with complex shapes, that simulate the scan of only a few rooms of the building, and apply this idea into a "Room Mode" test of matching walls, is by cropping box-shaped selections in CloudCompare, but then merging the desired boxes to obtain a non box-shaped selection. One example of such a simulation is shown in Figure 55, where a scan, and therefore a test, is performed in an area of the building, but there is a dent, as the room to the bottom right bound by walls 67, 16 and 76 should not be included. The walls in Figure 55 shown by the lightning markings, walls 7, 8 and 73, are removed from the IFC file, but are included in the scanned segmented walls list, to imply that they were found in the scanning session of those rooms, so they should be automatically generated in the IFC file. Then it would be present both in IFC and at the point cloud, making the wall matched and not deleted. All other walls present and numerated in the selected area to do the test, shown in red, should be copied to the folder where the test is conducted – as the tool renames wall1, wall2, wall44, wall55 to wall1, wall2, wall3, wall4, etc and we want to keep our ground truth folder intact. The results of this and other tests will be shown in the subsequent sections.

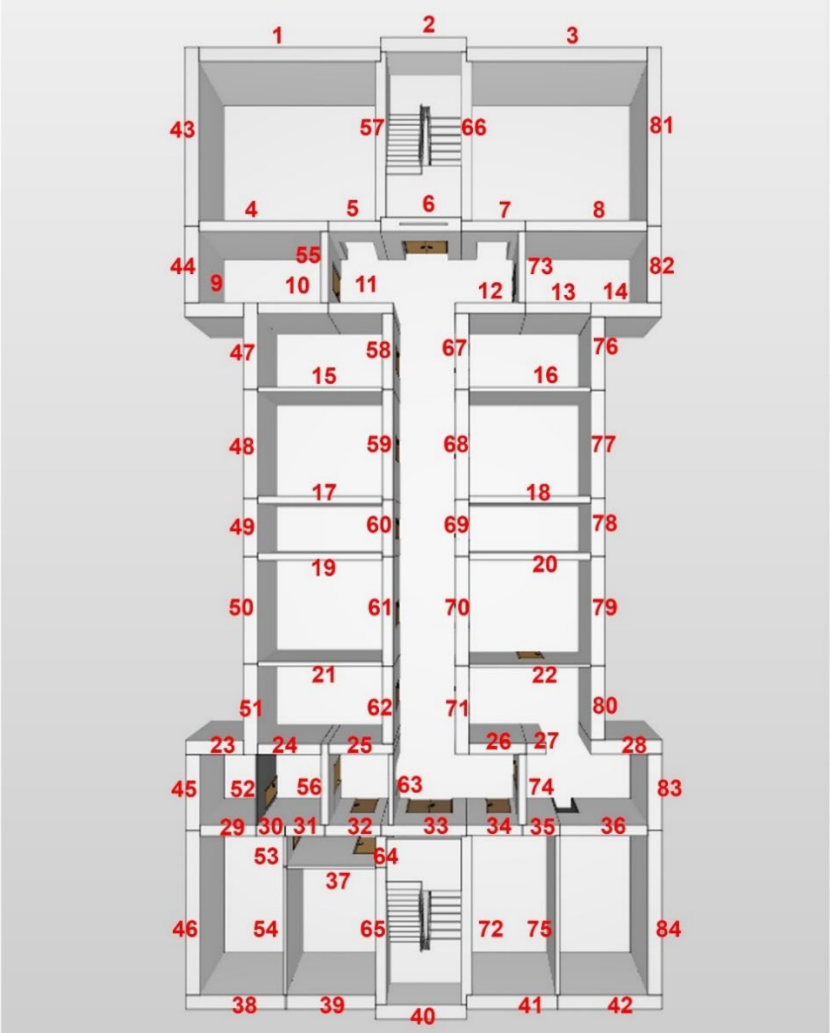

Figure 54: Haus30 project file, with every wall assigned to a number

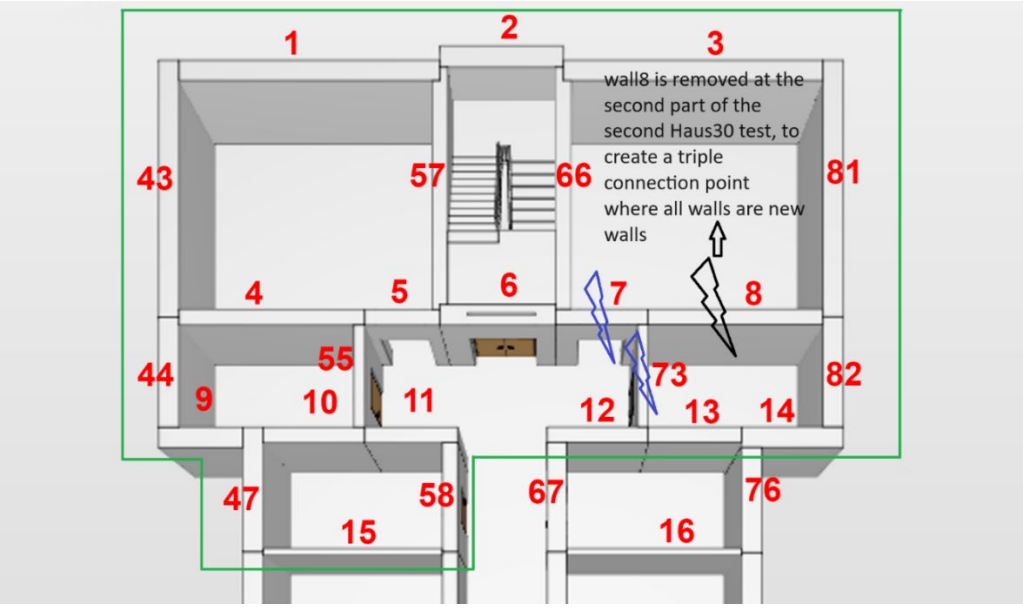

Figure 55: One of the two tests developed to test the Room Mode Walls' update functionality. This test is further discussed in Section 4.2.2

4.1.2.3 EMC dataset to test ceiling height update

IFC data of one of the Erasmus Medical Center's buildings, namely the Ca building, was provided by the EMC, and one particular area of interest of this building has an interesting configuration to execute tests based on checking discrepancies based on ceiling heights, as two different ceiling heights are present in consecutive areas. The area of relevance is shown in Figure 56.

Figure 56: To the top, relevant section of the IFC model with two different ceiling heights. At the bottom, the LiDAR scan of that area, made using Polycam

As the original file shown in Figure 56 has very complex geometry in its façade, opening the file in IFC visualizers is already challenging in terms of processing. Converting the IFC file into STEP, which is the methodology adopted by the python tool for visualization, becomes such a long process that is not practical. Therefore, an IFC model was generated with the same layout of walls, slabs and ceilings, and same dimensions of the original IFC file, but without overly complex geometric detailing. This IFC file is shown in Figure 57. A variation of this IFC file with the two ceilings in different heights is also made, so that their height can be adjusted based on the point cloud geometry. The test with this dataset is further discussed in Section 4.3.

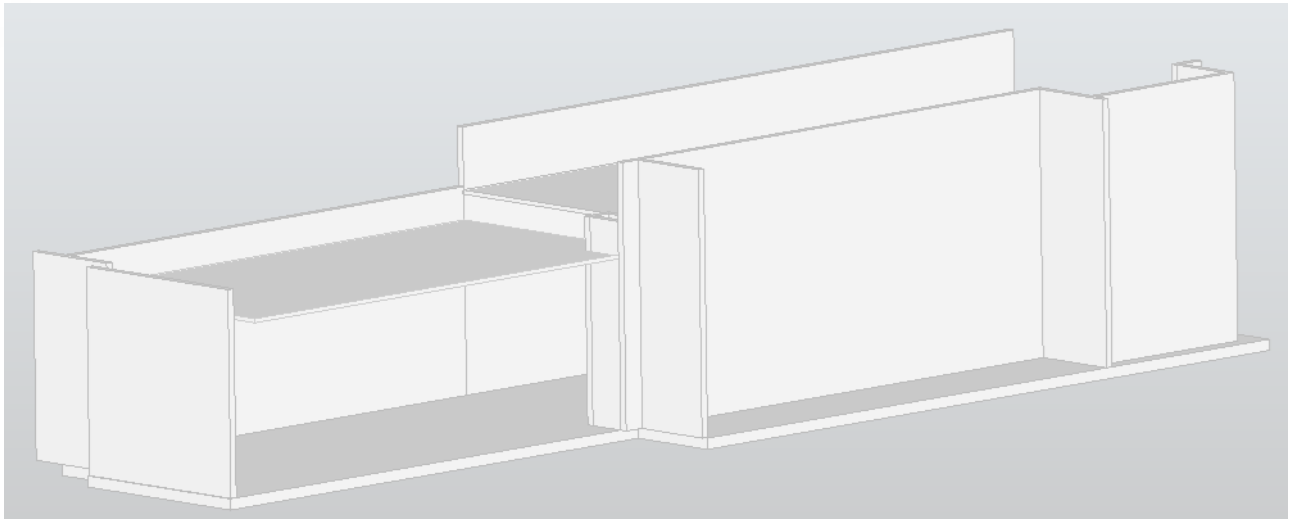

Figure 57: Simplified version of the IFC file of the area studied from the Erasmus Medical Center

4.2 Tests on wall update

4.2.1 Small-scale tests in the Atlas building

The initial test was performed on a small-scale scenario, by removing one of the walls in the IFC file of room 8.304, from the ground truth IFC file. Both the IFC file used as ground truth (the as-designed state of the building) and the point cloud ground truth (the de facto as-is state of the building) have very little deviation, and can be considered as correct, with the exception of the column in the as-designed IFC file. Here only the walls are being studied, first, so changes brought into the IFC file simulate a different as-design concept of the building. If one of the walls is removed from the as-designed file, but is still found at the point cloud, that is, the real building, then the procedure created will have to create a new wall automatically that is equivalent to the wall that was deleted. This scenario can be seen in Figure 58, where a simple IFC file is shown of a room with only three of its enclosing walls, when the point cloud indicates that the at the real building the room is fully enclosed by walls from all four sides. This could have happened in a scenario where a building was conceived as a open space with access to other rooms, but a change was made to keep this room separated by this vertical wall on the right side.

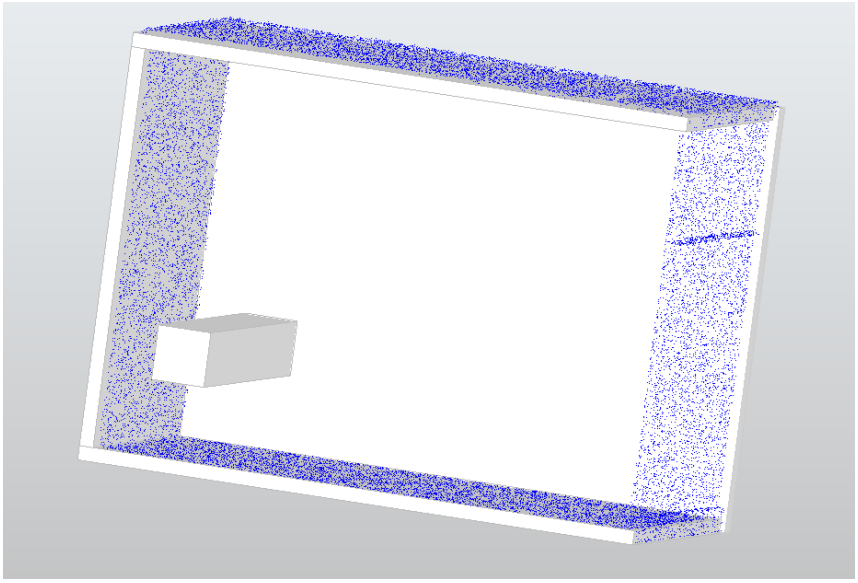

Figure 58: The as-designed model represented an open space, but the surveying of the building shows that the room is fully bounded by four walls

Such a simple model, that has scan data for all of its area, can be updated using both the Walls and the Room Mode Walls functionalities, with the simpler walls functionality being slightly faster. In an AMD Ryzen 7 5800H laptop with 16GB of RAM the conversion of IFC into STEP took less than 1 second (reported as zero). The wall creation is successful in both "Room Mode Walls" and normal "Walls" mode, as shown in Figure 59.

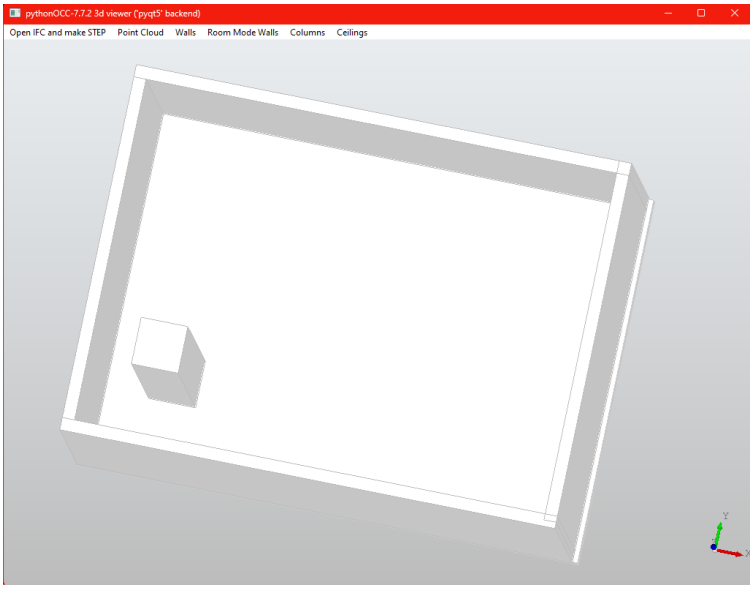

Figure 59: Successful automated creation of new wall to match the as-is reality of the building

The updated IFC file is outputted automatically with its generation date and time as part of its filename, and has all relations properly established, such as containment in a level or connected elements, as shown in Figure 60. Other properties such as identification properties, wall type that matches other walls in the model, chosen by thickness and walls commonly found in that region of the building, and a unique identifying ID, as well as quantities are also properly present as expected.

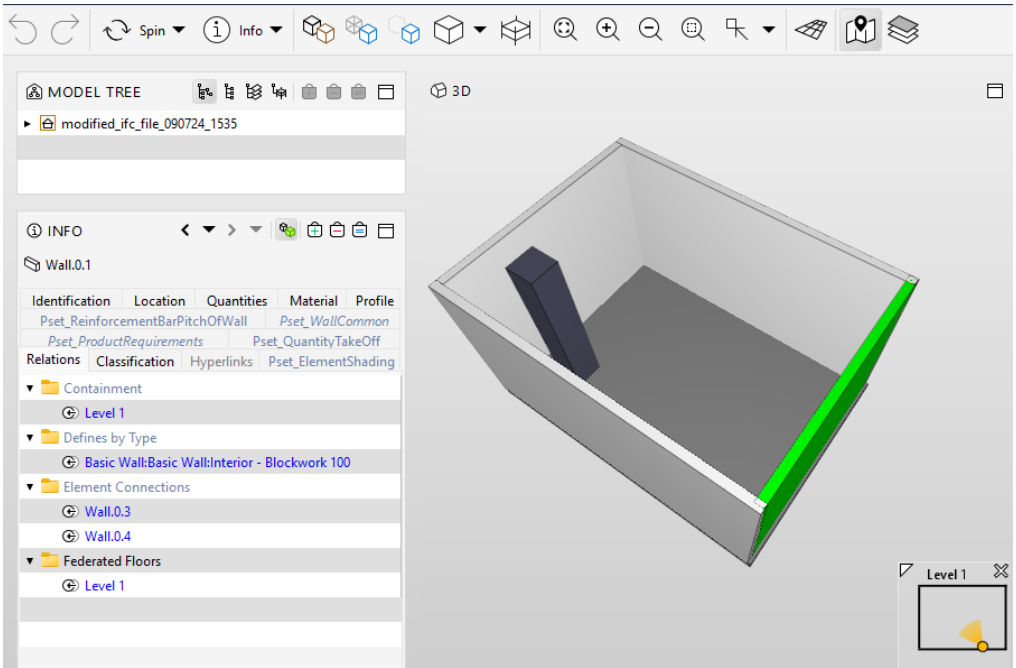

Figure 60: IFC data seen in Solibri, note at the top of the window the filename that refers to the timeframe of the building update, and the relations of the selected wall (the new wall is shown in green), that properly indicate connections and containment

After verifying that the model can update vertical walls that are missing in the as-designing IFC file, for simple projects, another test is made to see if the tool and algorithms also work properly on horizontal walls that are missing from an as-designed IFC file. Figure 61 shows a scenario where the room was designed as even more open-spaced, both the horizontal walls are not present, but at the real building, when surveying it with a LiDAR scanner it is found that the room is fully enclosed by walls. The right side of the figure shows the successful automated generation of the two missing horizontal walls. Notice the smoothened wall corners created as the methodology proposed in Section 3.3.2.2. Here too the generation of properties, connections and attributes worked as expected, as can be seen in Figure 62.

After demonstrating that the generation of new walls in as-designed IFC files works well in this small scale, a test also had to be performed to see how the deletion of walls that did not find a match with the point cloud worked. As this is a very small project, and the "as-is" state of the point cloud only represented the four bounding walls, there are two ways of testing this deletion of walls. One of them is creating an extra wall in the as-designed IFC file, inside of the room, that naturally is not present at the point cloud, and the other way is not loading one of the four segmented walls at the stage of selecting segmented walls. This second option would give the idea that only three of the four walls exist in the real building, and the other one, not loaded, is not at the real building and therefore should

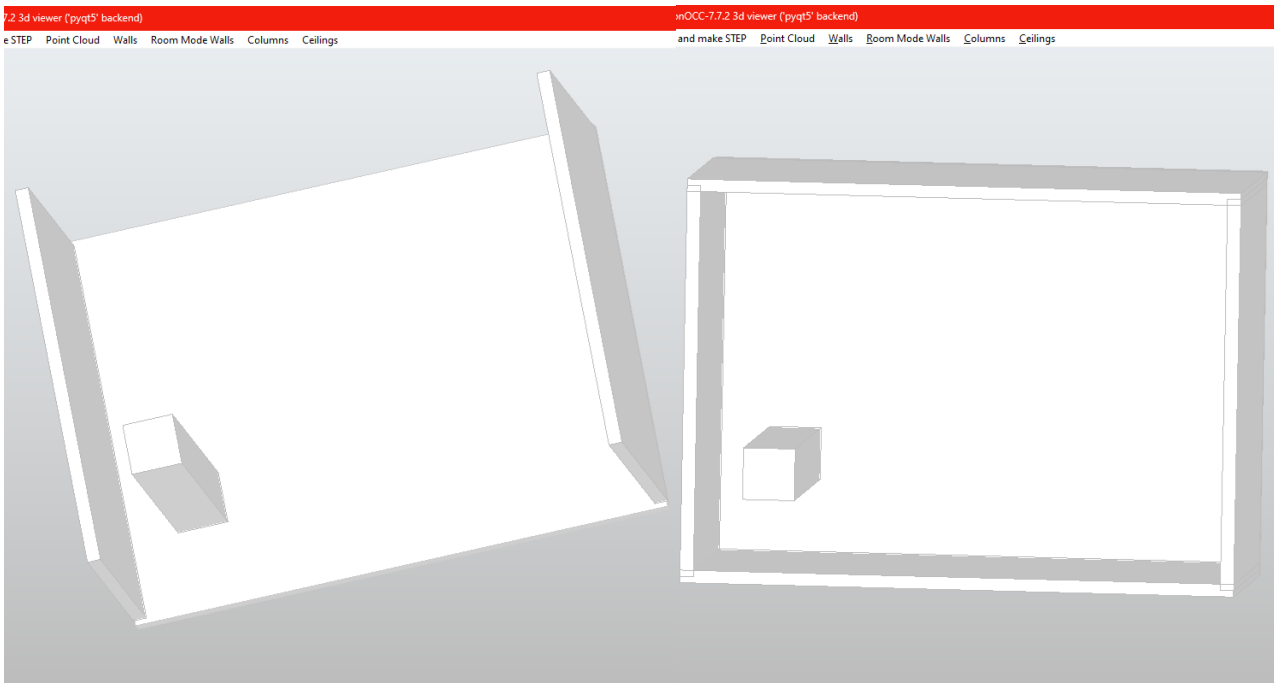

Figure 61: Successful automated generation of multiple horizontal walls at once

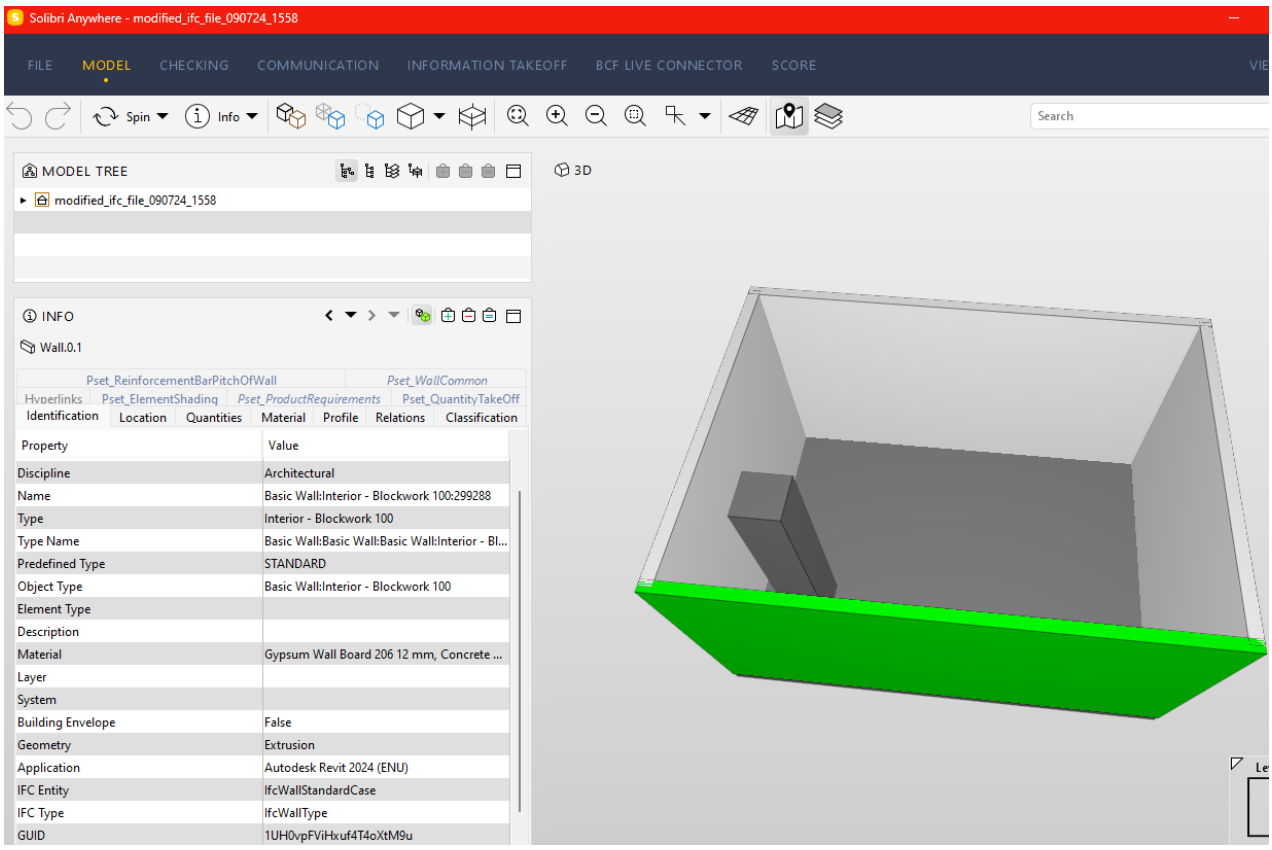

Figure 62: Several types of identification properties properly generated as expected for the new walls

be deleted, and the first option implies that an extra wall was predicted but not executed, or it was demolished. The first option results in a not so realistic (aesthetically) looking alternative, but that is very useful because openings and decomposing elements can be inserted into the wall, e.g. a door. This can help in testing the deletion of decompositions of the wall as well, which should happen, otherwise there would be a wall floating in space at the model, besides the opening elements that are also attributed to the wall and need to be deleted. This first scenario is shown in Figure 63, and the second scenario is shown in Figure 64.

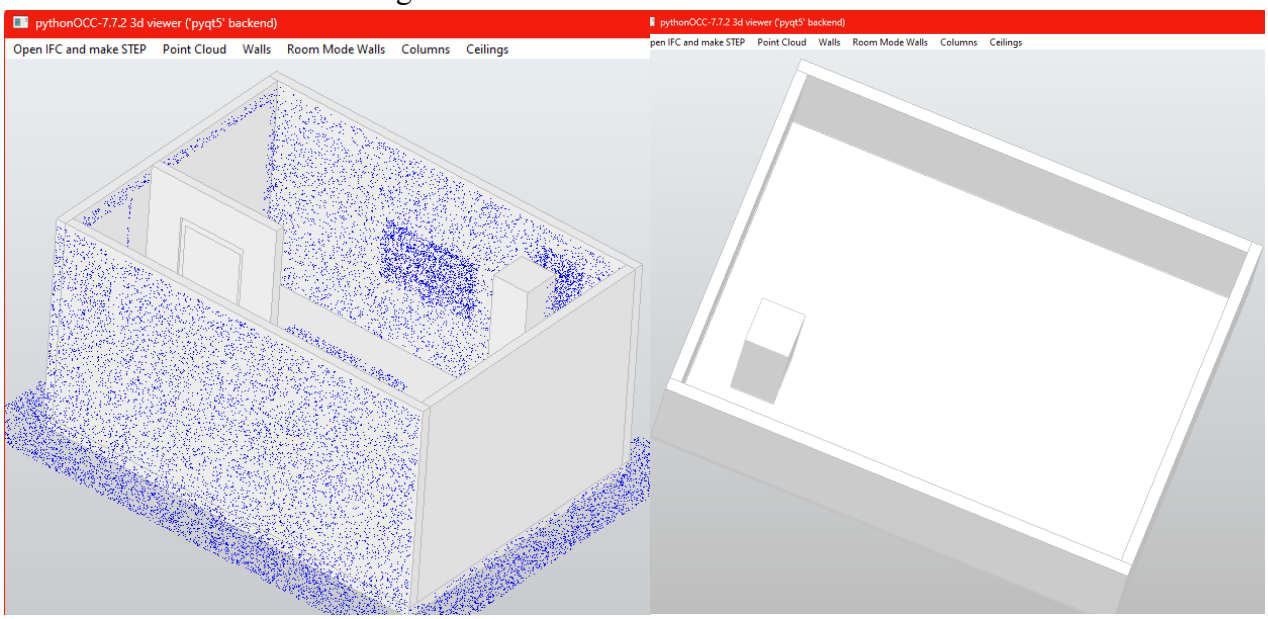

Figure 63: Deletion of a wall and decompositions of a wall (e.g. a door and its opening entity)

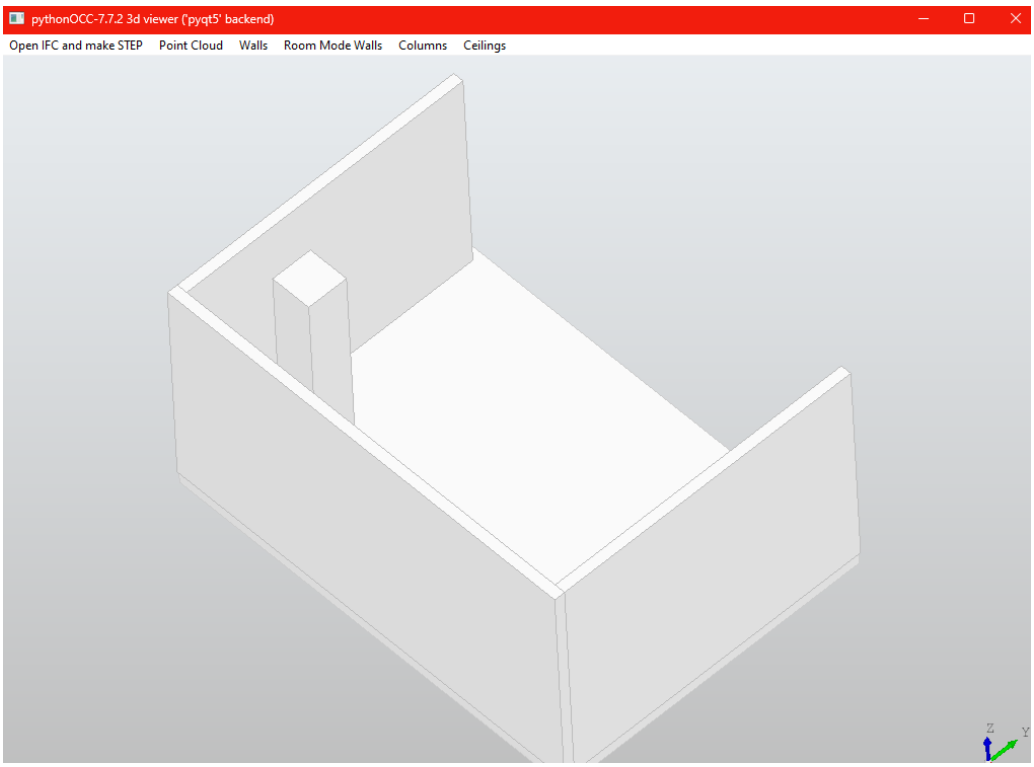

Figure 64: By uploading only 3 of the segmented walls the as-is state is considered as the building only having 3 walls, therefore one of them is deleted by the procedure, as shown above

The deletion of walls using python code is relatively straightforward, the most important principle being deleting first all decompositions of all walls that are going to be deleted, and then deleting the walls themselves. The decompositions of an element are found using the command *ifcopenshell.util.element.get_decomposition(wall)* and the walls are deleted by using the line command *ifcopenshell.api.run("root.remove_product", model, product=wall).* Because the implementation of this functionality is quite short, it is shown in Algorithm 2.

Algorithm 2: Deletion of walls and their decompositions

```
Define function wallDeleterRM(model, walls_to_delete):

    # Step 1: Remove decompositions for each wall in the list of walls to
delete
    for each wall_id in walls_to_delete:
        get the wall object using wall_id from the model
        if wall exists:
               get decompositions related to this wall
               print the decompositions found
               for each decomposition:
                      try to remove this decomposition from the model
                      if successful, print success message
                      if error occurs, print error message

    # Step 2: Remove the walls themselves
    for each wall_id in walls_to_delete:
        get the wall object using wall_id from the model
        if wall exists:
               try to remove this wall from the model
               if successful, print success message
               if error occurs, print error message
```

4.2.2 Larger test

After the creation of walls that are missing in the as-designed file and deletion of walls that are not matched to the point cloud from the IFC file being successful in a smaller scale, those functionalities should be tested in larger projects and buildings. This helps seeing if they still work as they should, and seeing how the thresholds are applied when a larger variety of wall types is employed. Using a larger dataset also allows the testing and verification of the usability of Room Mode, so that scans can be collected of only a section of a large building. The **first test** with this larger dataset shall thus be made to test the wall reconstruction from point cloud geometry, and wall deletion in a large model. This is also a multi-storey IFC project, so this projects helps testing the procedure in a multi-storey perspective as the generation of checks of elements also takes into account the level of the storey an element should be, besides its local elevation relative to that storey. The **second test** of this dataset deals with a more complex and advanced use case. In this test, the improvement of geometry generation is tested at a point where three new walls automatically generated connect to each other.

As mentioned in Section 4.1.2.2, the larger dataset chosen to perform those tests is the Haus30 dataset. This file has many varieties of wall types with several wall thicknesses, which provides a high degree of complexity to test wall updates. For the first test, the area seen in Figure 65 will be considered as the scanned area. The walls shown with a red lightning symbol, at the left side, symbolize walls that were not found when scanning this region of the building, so they should be deleted by the tool. The wall shown in blue, at the right side, symbolizes that, at the current state of the building, the room above and the room below it are now fully separated, by a longer wall. Figure 66 shows the point cloud with a complex shape, that simulates scanning only part of the building, produced by merging three “box-shaped” sections of the complete scan of the building.

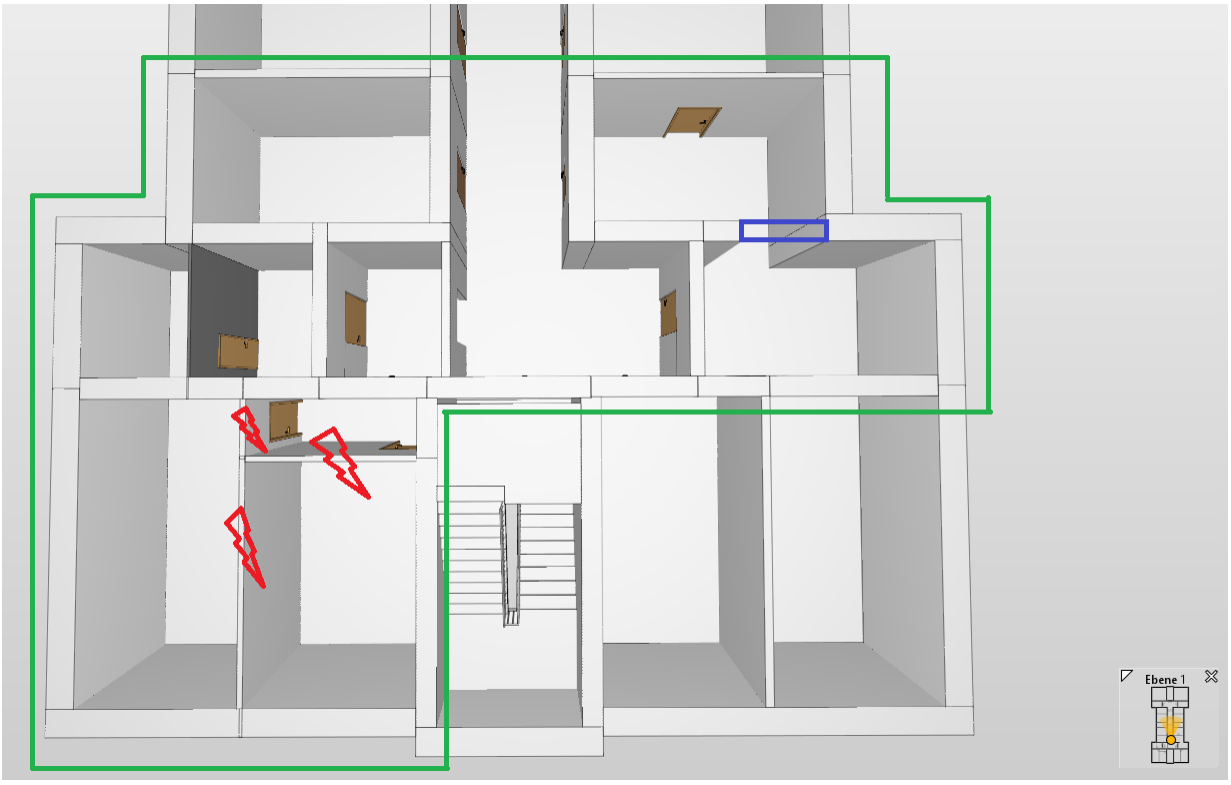

Figure 65: Scanned area outlined in green. Wall to be redefined shown in blue. Walls the be deleted shown in red.

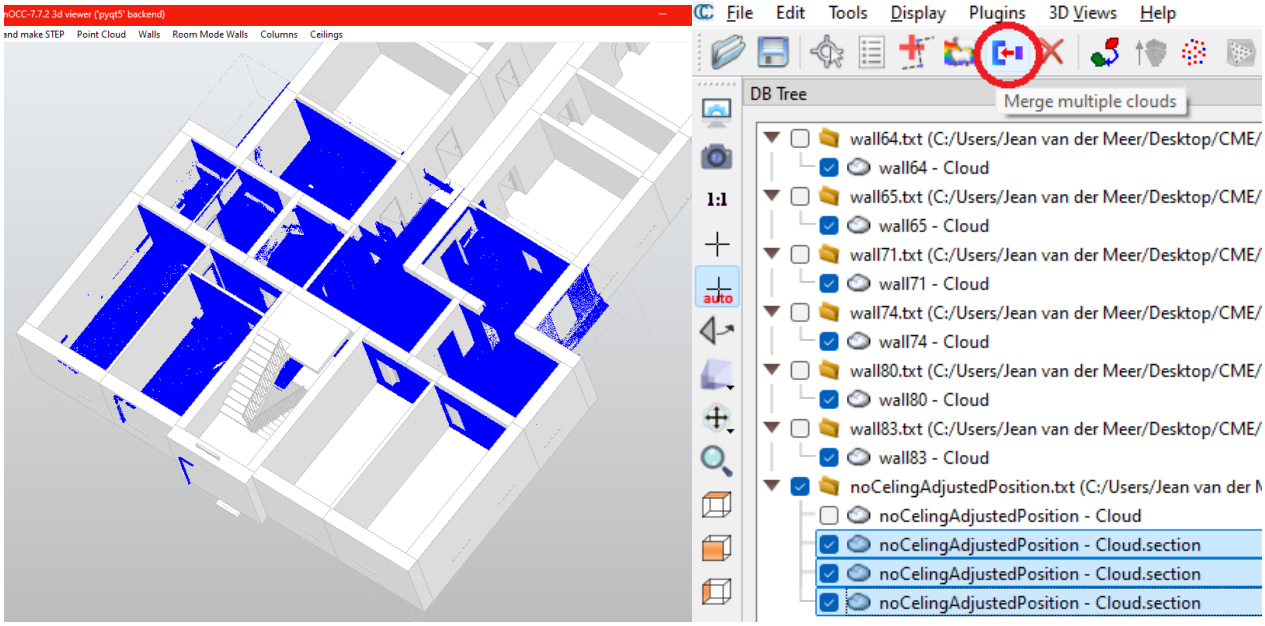

Figure 66: To the left, scan of only a section of the building, not including, for instance, the three rooms at the bottom left corner, neither the areas above in the floorplan. To the right, how a simulation of such a partial scan can be made in CloudCompare, based on the complete scan of the building. A number of box-shaped sections of the scan can be made, and then merged to form a complex scan-shape.

To simulate removing the three walls shown in red, those walls are simply not loaded in the list of segmented walls out of the real building. To simulate a new and longer wall being found in the building, as the one shown in blue in Figure 65, a copy of a scanned wall of similar dimensions can be used and moved to the right place, using CloudCompare as shown in Figure 67.

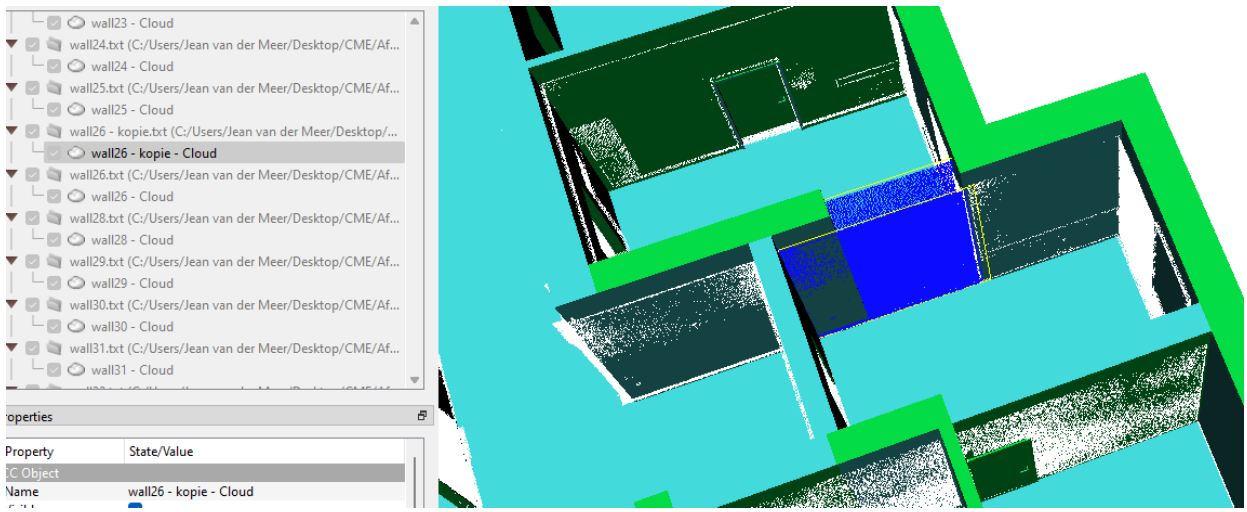

Figure 67: A complete wall that separates both rooms can be simulated even when it was not found in the original point cloud. Here, the wall right beside it was copied, and the copy is moved and has the right number of points selected to fit the necessary area.

Figure 68 Shows the successfully updated file with the non-matched walls deleted, the new wall created, as well as all the other non-scanned walls in the IFC model intact. This shows that the implementation of the Room Mode functionality also worked. This successful update means that the methodology developed also works well for a multi-room perspective, as the data in IFC is located in floor 1 that has an elevation of 3.8m. In this project, walls are created first with a z coordinate of their starting point (their height in space) as the coordinate found at the point cloud, that in a floor located at a level 3.8m tall would render a z coordinate around 3.8 too at the base of the scanned walls. The as-designed IFC walls, however, have their coordinates set to 0.0 and reference their height

as the height relative to their floor, their base point coordinates are then relative to that floor. Because comparisons, updates and analyses of the surroundings of a wall need to be made, it is important to assign the newly created walls into their proper floor. Before this can be done, the function that checks the starting and end point of a wall has a special case to handle IFC walls that do not have a floor assigned to them, and for all other walls it considers the z-value of the base as the elevation of the IfcBuildingStorey pertaining that wall. Another interesting thing to see in Figure 68 is that the aligned shown in red has many walls that have very short extents, which could be a vulnerability of the modelling standard adopted for walls to aid comparisons. Problems could happen when matching walls, when they are this short, because some of those walls are quite thick, implying in large threshold and tolerances when finding the starting point and the end point. With large tolerances, the algorithm could perhaps confuse starting points and end points of walls, when comparing them, and delete walls when it should not, and recreate them in the wrong place or with the wrong geometry. Besides the effectivity in matching IFC walls and point cloud walls in the method developed, that helps avoiding this problem, one key-element that helps avoiding mismatching in those complex scenarios are **dynamic thresholds**, that will be discussed at the description of the second test hereafter. Dynamic thresholds allow flexibility in thresholds used, by making them proportional to wall's thicknesses, besides having a minimum value.

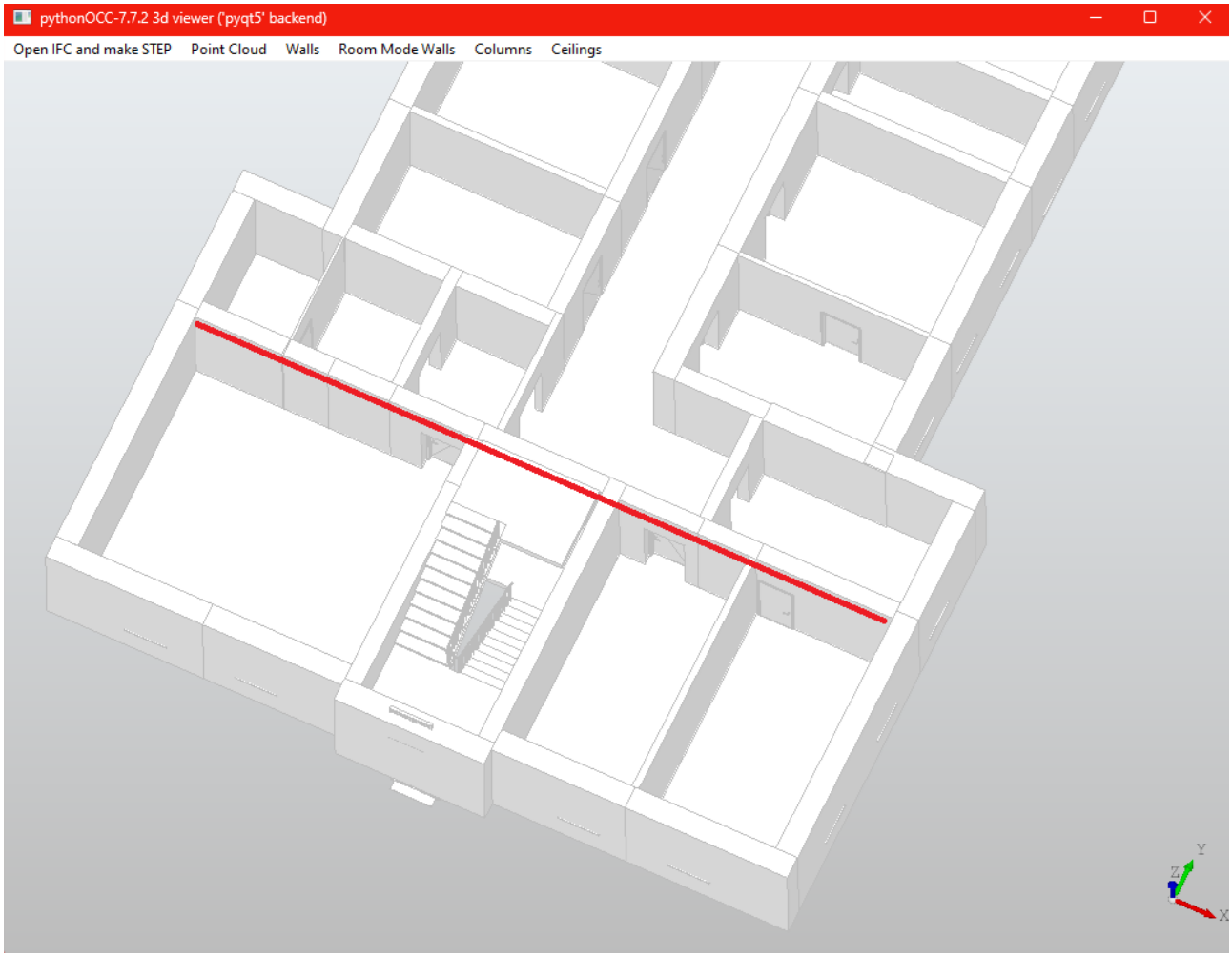

Figure 68: Updated model, with the two rooms that should be fully separated divided by a new wall, and the three walls that are not present in the as-is building deleted from the IFC file. No unexpected changes were made to other elements in the model.

A **second test** is developed aiming to create new walls in complex scenarios using Room Mode, in this large dataset, for the procedure developed. This test is divided in two stages. First, one as-designed IFC model is simulated, where two walls are not at the IFC but are found at the real building, at the point cloud. Those two walls connect to each and to other pre-existing walls in the IFC file. This first stage is shown in Figure 69. In the second part of this test, the as-is designed IFC simulates the scenario where three walls are missing in the outdated IFC files, but present in the current building, so they need to be generated in IFC. They all connect to each other at one point, and also connect to other pre-existing walls in the model.

Figure 69: New as-designed configuration, with walls 7 and 73 from the ground truth mentioned in Figure 54, removed from the as designed stage. They are shown in blue. As point clouds are available for them, if all segmented walls in this area are provided, walls 7 and 73 should be recreated by the procedure. The scanned area, where checks should be performed, is outlined in red.

The creation of new walls is naturally trickier than the simple deletion of walls. Many things went wrong at first, even walls that should be checked as they are present both in as-designed IFC and point cloud, were not matched, and then reconstructed, as seen in the alignment shown in red in Figure 70. The reconstruction rebuilt all walls, but was not very satisfactory, with a good deal of misalignment, overlapping, or indentations at connections, and even a gap between walls that should be connected, seen in the lower right corner. The expected correction of corners and geometry did not work well at this point. The initial diagnostic was that the concave area needed a slightly bigger buffer to check walls, but specially the function that compared point cloud walls and IFC walls needed a bigger threshold, to detect that walls are indeed matched, and leave them intact if it should be so. The function that corrected the geometry also needed a higher threshold to see that there are walls close to a particular wall, that should be connected to it.

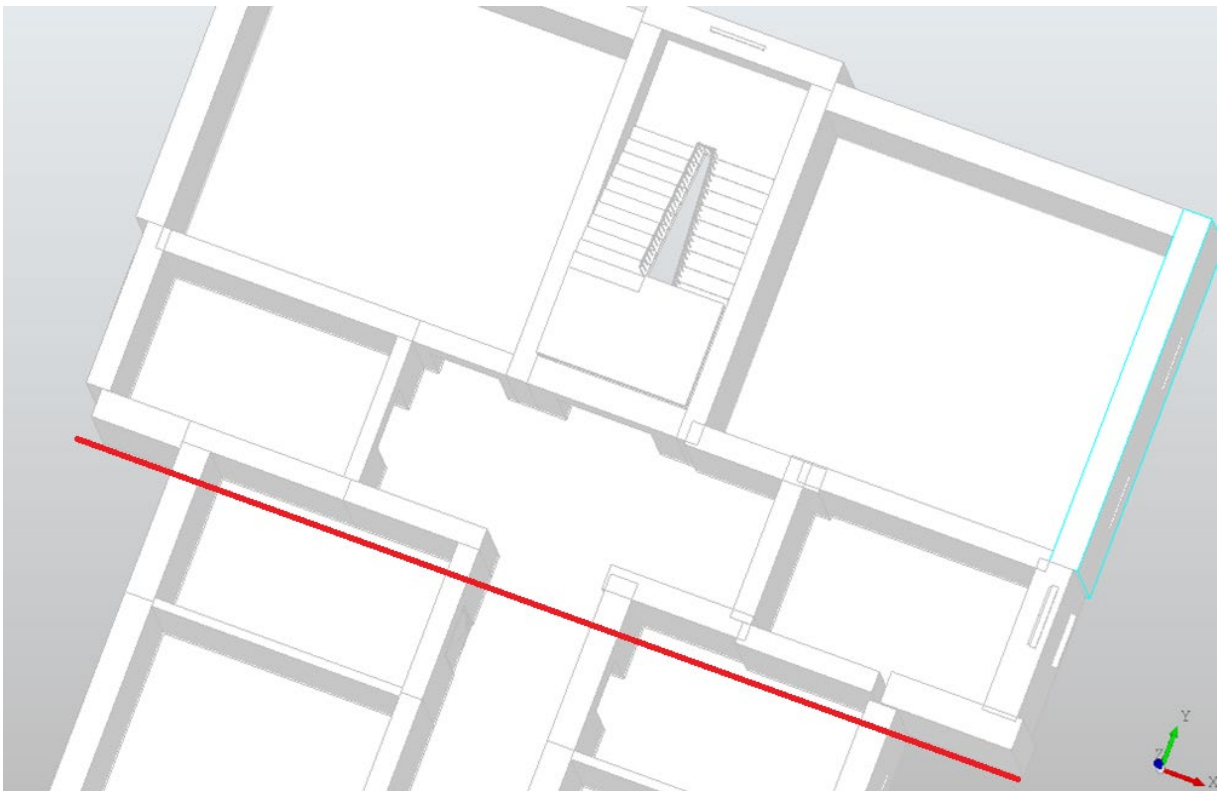

Figure 70: Initial result with limited reconstruction and errors in wall checking

It was then noted that there should be higher thresholds, but applying thresholds of 1m or even more for all walls of the project seems exaggerated and inappropriate. There is a large variety of wall types, ranging from 6cm to 56cm in thickness in the IFC file (and even more in the point cloud, due to the noise at windows). IFC defines the starting point of a wall slightly differently than the segmentation of point clouds does, but for wall that is 6cm thick, there will not be a large difference in position between starting and end points of IFC wall and point cloud wall. For a very thick wall, however, the discrepancies are much larger. This prompted the creation of "dynamic thresholds", where they have a minimum value, but can get larger if the wall is very thick, and are proportional to the wall's thickness. An example is shown in Listing 3:

```
ifc_wall_dim_y = ifc_wall.Representation.Representations[1].Items[0].SweptArea.YDim
dth = max(0.65, 2.5*ifc_wall_dim_y)
```

Listing 3: A definition of dynamic threshold, that can increase depending on the wall's thickness

This solved the problems with some walls not being properly checked, and improved the reconstruction of geometry, but raised attention to a limitation of the algorithm that improved the geometries at connections: if a wall has two (or more) different walls connected to its starting point, the algorithm, with its form up to that point, would pick a random one of the two to adjust the geometry. In complex wall connections, this can generate undesired misalignments, as shown in Figure 71. It would be much more natural for the wall highlighted in green to keep the horizontal alignment, from the horizontal wall at its left side, instead of aligning to the end of the vertical wall to its left. The code that smoothens connections, and starting and ending points, was then changed to give priority to horizontal alignments when a horizontal wall is connected to both a vertical and a horizontal wall, and to give priority to vertical alignments, when a vertical wall is connected to both a vertical and horizontal walls. The result of the improved code is shown in Figure 72.

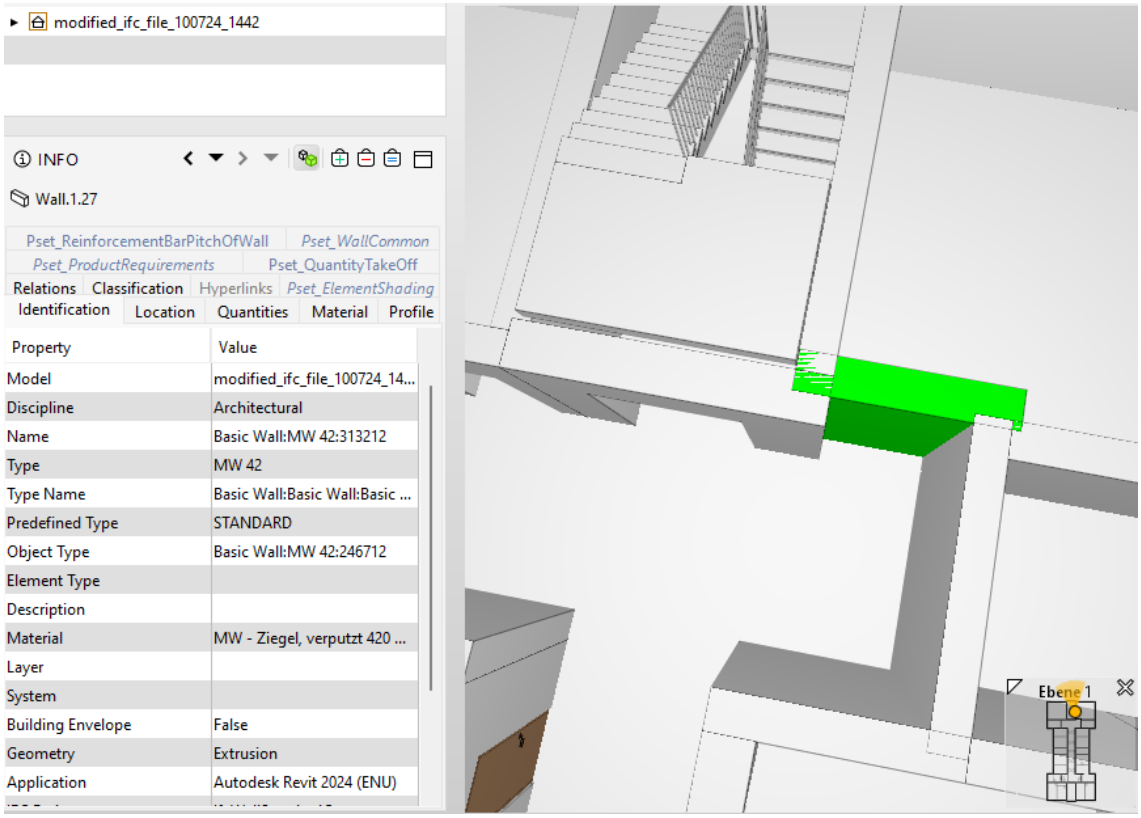

Figure 71: Alignment of walls at random, before giving priority to horizontal or vertical alignments

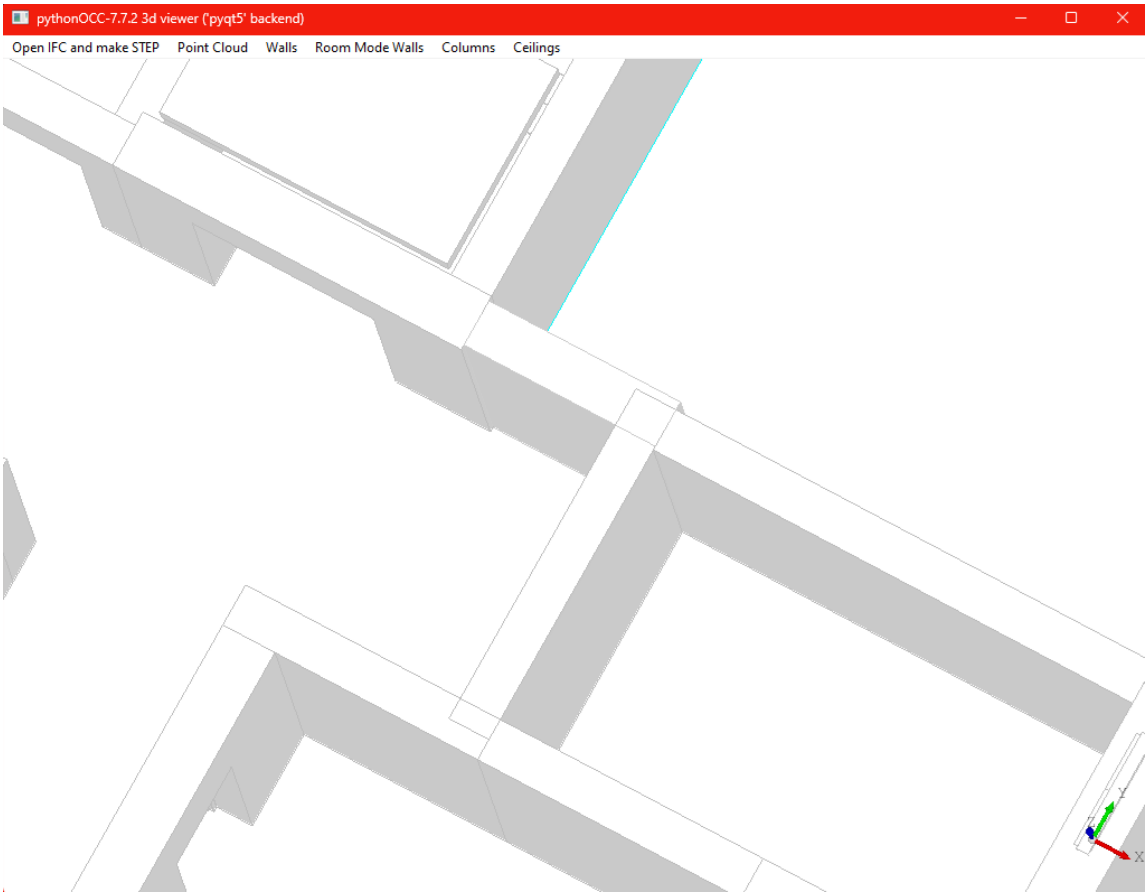

Figure 72: Improvement in the alignment by giving priority to walls of the same orientation

After that, it was time for a yet more challenging scenario**: the second stage of this test**. It was about the simulation of a larger open space in the as-designed area, that should is closed in the scanned/real building, and therefore creates three new walls connected to each other. This is shown in Figure 73.

Because the process that adjusts the geometry at the connections corrects the geometry of the newly created walls, connecting them better to as-designed geometry, it doesn't e.g. extend the end of a new wall to match it with another *new* wall, because this wall is also being corrected based on the as-designed geometry. Maybe the other new wall was not even parsed by the iterating function at that moment yet. This can generate new walls with gaps in between them, or indentations and other such geometric inconsistencies, which is shown at the right side of Figure 73.

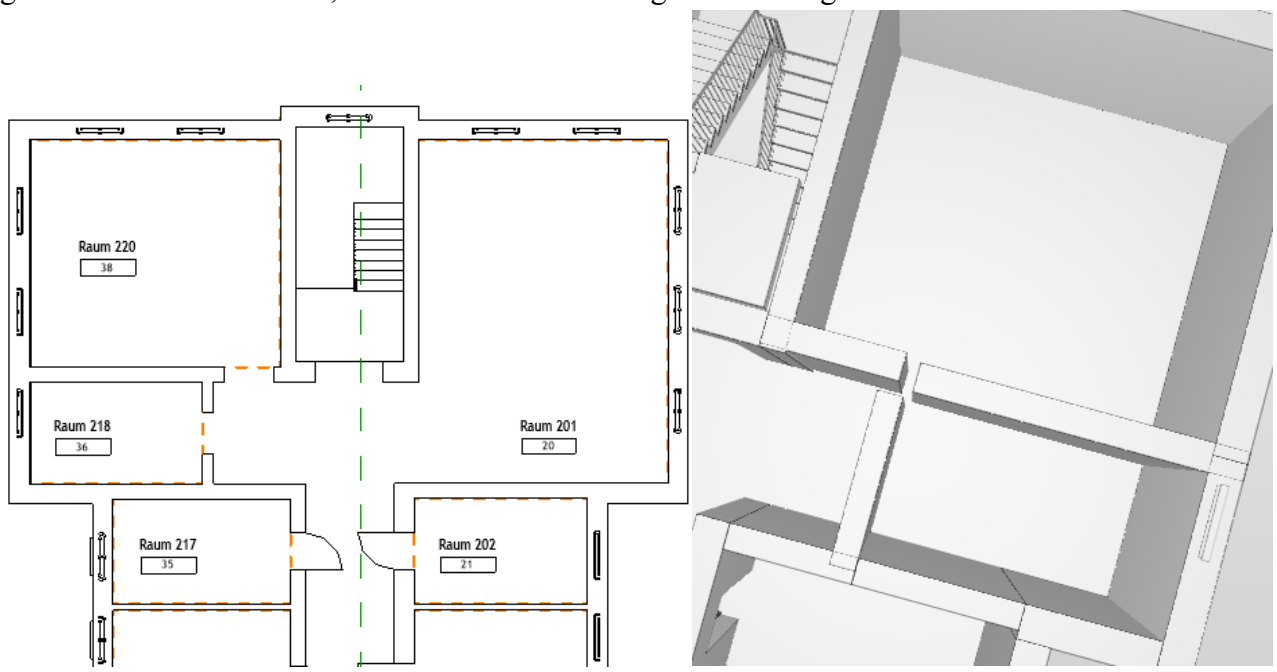

Figure 73: Left: the new configuration of the as-design model used for this third test. Right: When all new walls meet each other, also known as the new wall standoff

The solution to this scenario, is to repeat a similar process to the "corner smoothener" function, after the walls are already corrected based on the as-designed geometry, and then run an adjustment based on new walls, where the starting coordinate of all new walls might be corrected to go closer to either the end or the start of another new wall, and only another new wall, if it is close enough to it. The end of the new walls can also be adjusted, i.e. by adjusting the end of the wall to get closer to either a new wall or as previously existing IFC wall. This process is done twice, as the movement of one wall might further influence the position of new walls around it again, so doing it twice helps obtaining an optimal adjustment. This results in a much better alignment, that is shown in Figure 74.

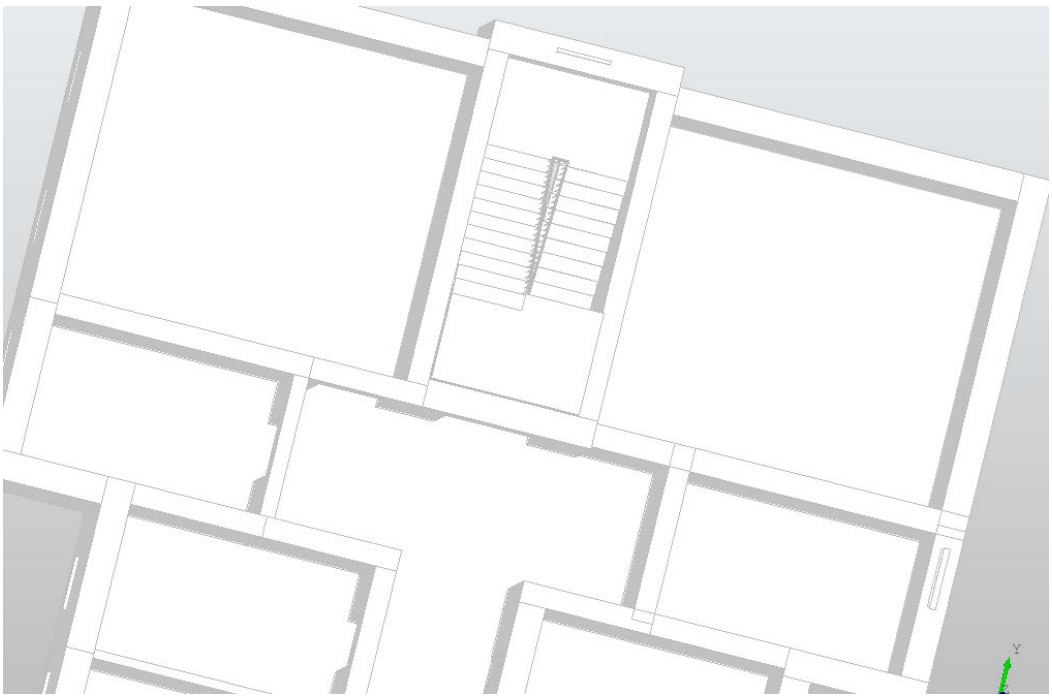

Figure 74: Alignment of new walls in a complex scenario, after all improvements to the algorithm

The later change of starting position of a new wall based only on the geometry of other new walls around it, should only be done if e.g. the new wall being moved is horizontal and there are no horizontal pre-existing IFC walls around it, or if the new wall being moved is vertical and there are no vertical pre-existing IFC walls around it. This is because horizontal-horizontal and vertical-vertical alignments should always have preference.

Evaluation of results

The computation of checks and changes, and generation of updated IFC models is almost immediate in this larger model, using an AMD Ryzen 7 5800H laptop with 16GB of RAM, but the conversion of IFC into STEP for visualization takes about 1 minute and 30 seconds. The correct generation or deletion of IfcWallStandardCase entities worked in all cases, and the geometry correcting algorithm works quite satisfactorily. Because the geometry generated, that initially could have inaccuracies from sensor precision or registration, is later corrected and improved, it can be understood that in about 100% of the cases the geometry generated is congruent with the rest of the building and aligned to it. The horizontal or vertical alignments are however based around a centerline of walls, which is good for when there are many consecutive walls, but for changing wall thicknesses it can generate small indentations, what is shown at the conclusions in Figure 87. Some geometrical overlapping of walls, in their interior side, also occurs sometimes, which is something that could be improved with a more complex algorithm that handles more exceptions.

4.3 Tests on the update of ceilings

A section of the building Ca at the EMC, was used as the dataset to test the update of ceiling height, as discussed in Section 4.1.2.3. This dataset has one ceiling defined by an IfcRectangleProfileDef profile, and one ceiling defined by an IfcArbitraryClosedProfileDef profile. Figure 75 shows a floorplan with the two ceilings mentioned.

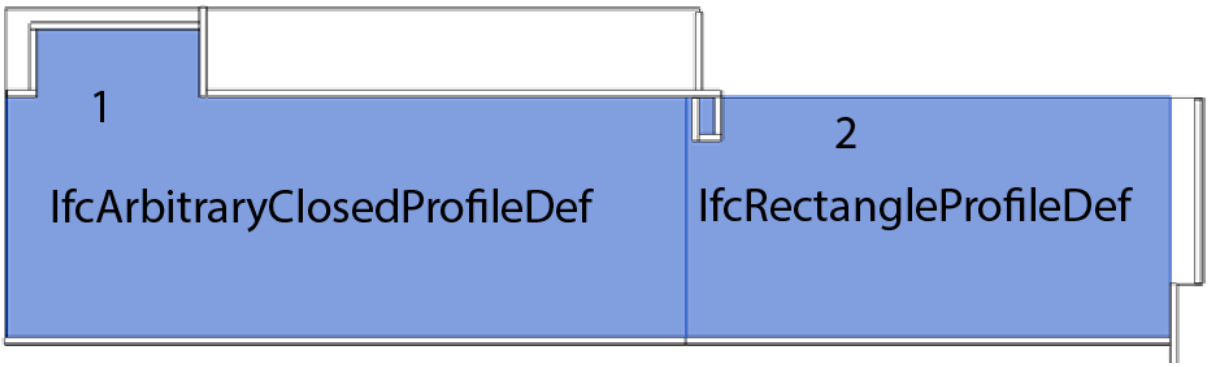

Figure 75: Ceilings present in the dataset

The IFC model used is based on a larger as-built IFC model used by the EMC, and therefore has very precise measurements, that matched quite well with the data scanned by the author. Figure 76 shows a section of the project, with ceiling 2 at the left side and ceiling 1 at the right side, and their respective heights that apply for both IFC file and point cloud.

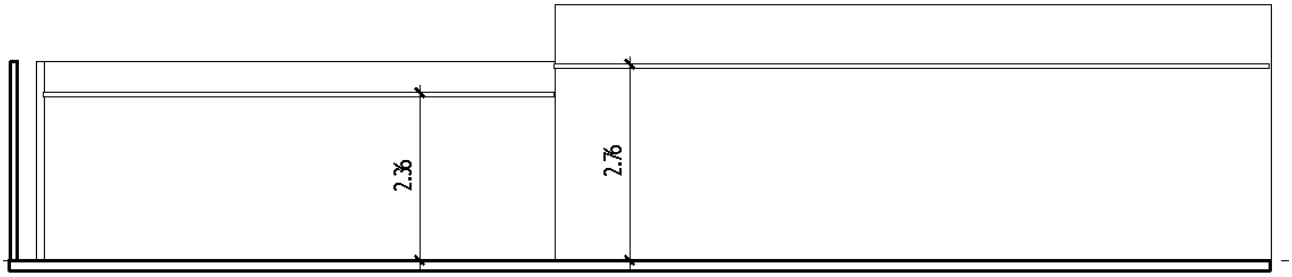

Figure 76: Ceilings heights at the building seen from a section view

Because the placement of the ceilings in the available IFC file were already quite updated, a synthetic IFC file was generated, with the same characteristics, except for different ceiling heights. Ceiling 2 was placed in a lower position (so it should be moved up to match the point cloud data), and Ceiling 1 mas moved into a higher position, so it should be automatically moved down by the "Point Cloud to IFC Updater", in order to match point cloud data and the real height of the building. The new heights are shown in Figure 77, which are now to be considered "the outdated IFC file".

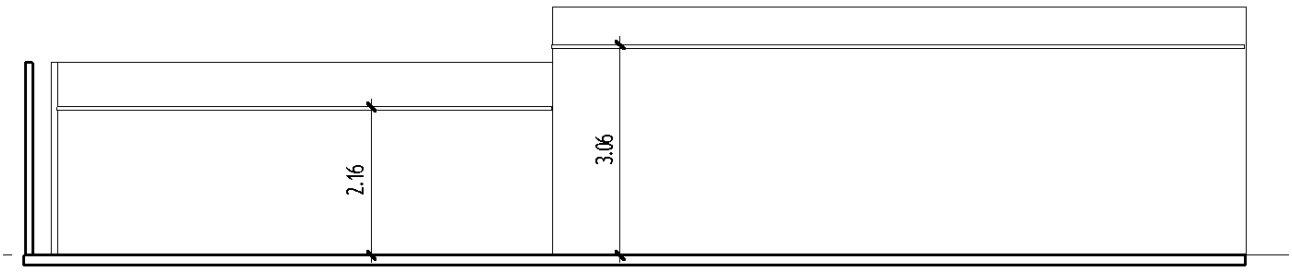

Figure 77: Simulated outdated ceiling heights, that are to be updated by the tool

As discussed in Section 3.4, the matching of IfcArbitraryClosedProfileDef-defined ceilings is handled differently than IfcRectangleProfileDef-defined ceilings, but the matching process and update of IfcCovering elements works well in both scenarios. Having updated ceiling heights can be useful for instance when checking a newly built building, to see if the ceilings were installed at the predicted height, and to know with certainty that the ceilings are at ta given height and there is enough room for e.g. placing installations above the ceiling. Figure 78 shows the initial height difference between point cloud ceiling and (simulated) as-designed IFC ceiling, and below, in the same figure, the successful update of the file can be seen.

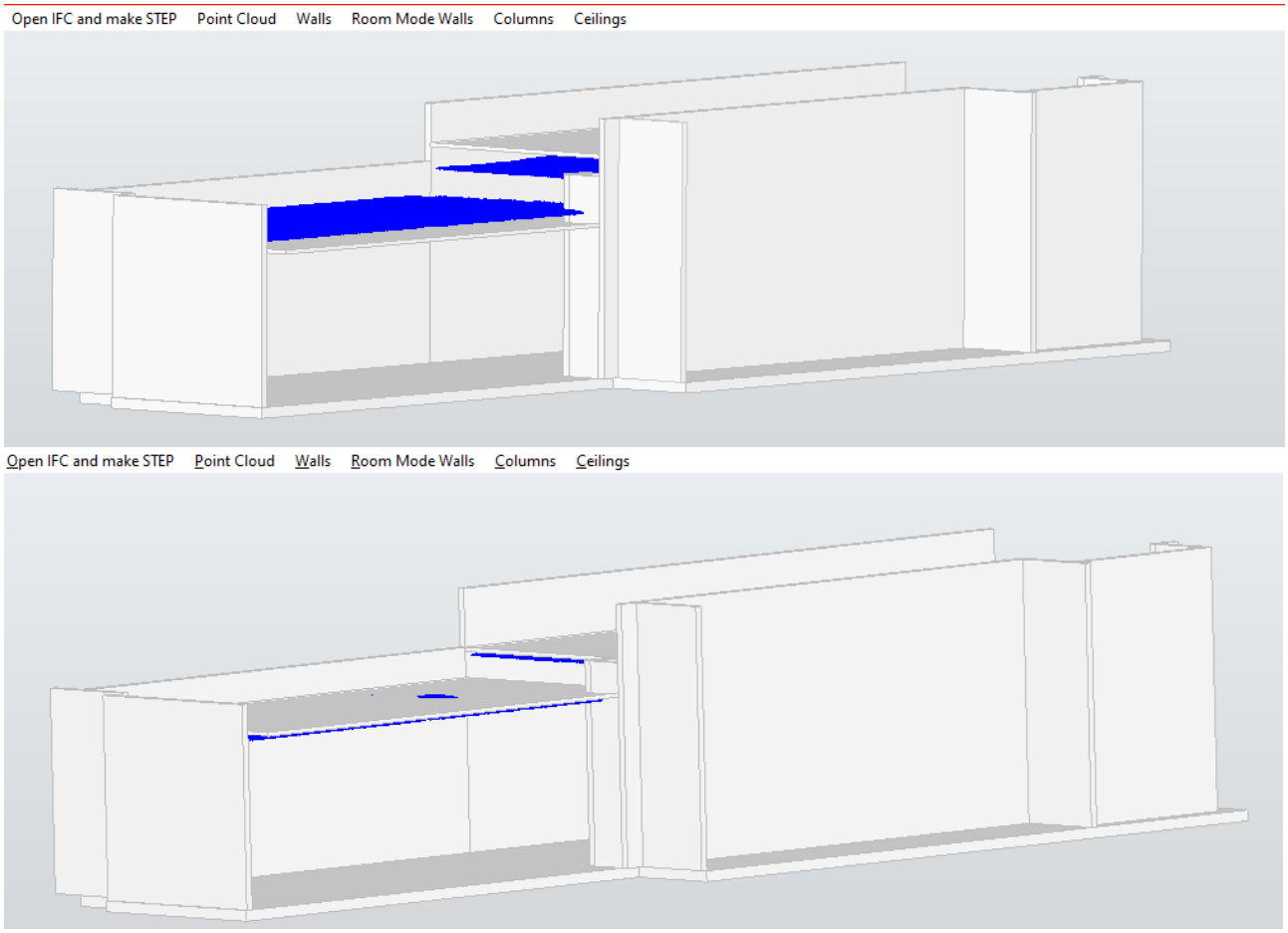

Figure 78: Above, the outdated state of the IFC file with wrong ceiling heights, with the correct height shown by the point cloud in blue. Below, the height of ceilings automatically and successfully updated

Evaluation of results

The updated ceilings were updated to the exact location expected, that is, the height of the segmented ceiling. How precise this update is compared to the real building will depend on the quality of the registration of point cloud and IFC file, quality of the segmentation process, and accuracy of the surveying sensor used, that depends per sensor.

4.4 Tests on the update of columns

Besides the IFC models discussed in Section 4.1.2.1, an array of possible scanned columns were created to test, in the different IFC files, and in different positions, the essential steps of column update, column deletion, column creation, and reports for columns that should not be changed. The columns that should not be changed are the columns fully or partially embedded in walls, as their geometry is uncertain. Very often they cannot be detected at all in the scanned data. Knowing how many columns exist in a real building, and where they are is an important information, for reasons such as structural integrity. Therefore automated reports can be made that can aid the management or renovations of the building. They are leveraged by the information contained in the point cloud, comparing how many columns were detected, distinguishing between embedded and not embedded columns, and how many columns were present in the original IFC file. Tests will start with the first test, done with a room (room 8.304, as per Section 4.1.2.1) with just one column, to test the functioning of column position update in a small scale. Later, a second test will be done with an iteration of room 8.304 shown in Section 4.1.2.1 that has more columns, where moving some, deleting

some, and creating some columns will be tested, as well as detecting an embedded column. The third and last test is with room 8.201, and involves differentiating between columns that are partially embedded and detected, and columns that are fully embedded and not detected.

First test

Test n.1 starts with a case where the column in the real building is much closer to the walls than at the available IFC file. There is only one column in the model, and they are close enough to clearly be the same column. For reasons such as model accuracy, space planning, use of furniture planning and renovations, it is important to know the right place of the column and have the model with the column updated to its right position. The deviation between point cloud and IFC can be seen in Figure 79.

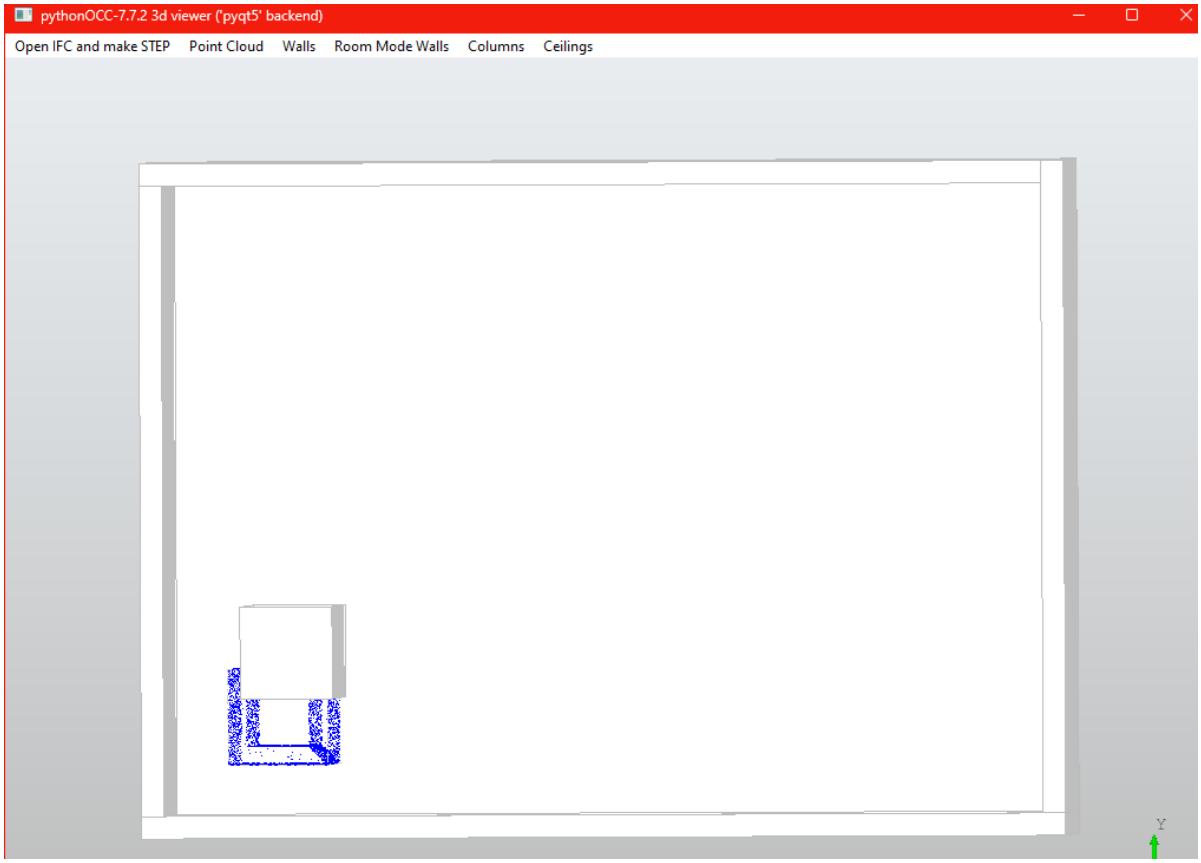

Figure 79: Point cloud column seen in blue, which is at a significant different position compared to the IFC column.

The center of the point cloud columns is found, and the position of the IFC column can be successfully updated. The updated model is shown in Figure 80.

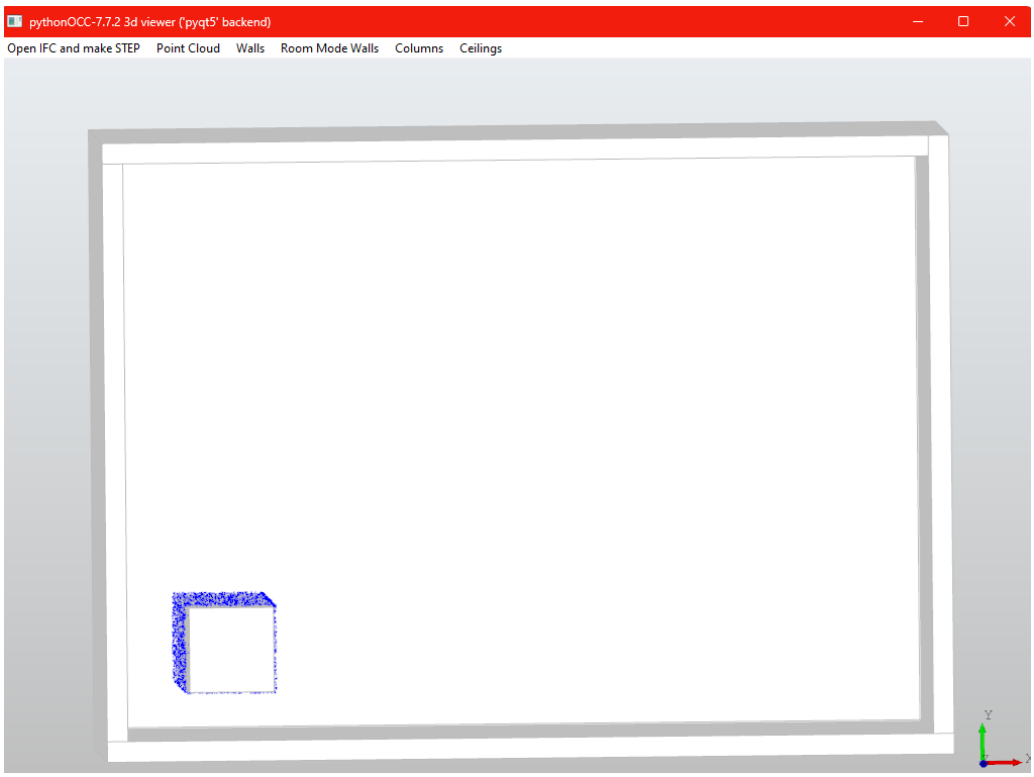

Figure 80: IfcColumn updated to the right position and matching the point cloud column perfectly

Second test

A variation of the first test is made, where three IFC columns are added, and four segmented scanned columns are found. For one of the original columns, present in the outdated IFC file, no point cloud column is found, so it has to be deleted. Another column is partially embedded in a wall, so it should be detected but not changed. One point cloud column is not matched to other columns, so an IfcColumn should be created for it, and other 2 columns should have their position updated. This case can be seen in Figure 81.

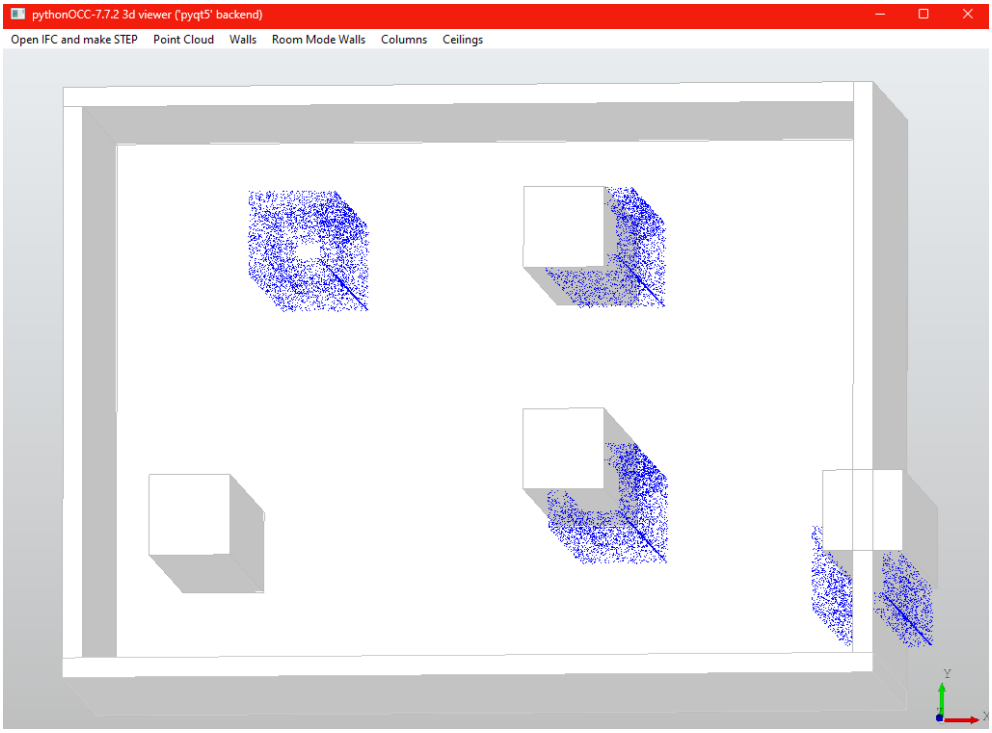

Figure 81: Configuration of scanned columns and IFC columns in the second column test

In this case, a warning is produced, shown in Figure 82, that tells the user that one of the visible IFC columns, from the as-designed model, and that are not embedded in walls, did not find a match in the scanned data of the building, which was the column at the bottom left corner. This column was then deleted. The column at the top left side was created at the right place, and the other two columns that are not embedded also had their position updated. The warning is shown in Figure 82, and the updated IFC model with the point cloud data superimposed can be seen in Figure 83.

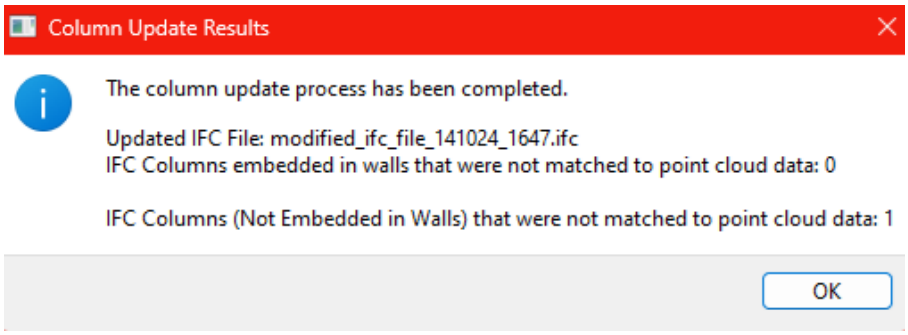

Figure 82: Pop-up warning that tells the user that one column from the original project could not find a match

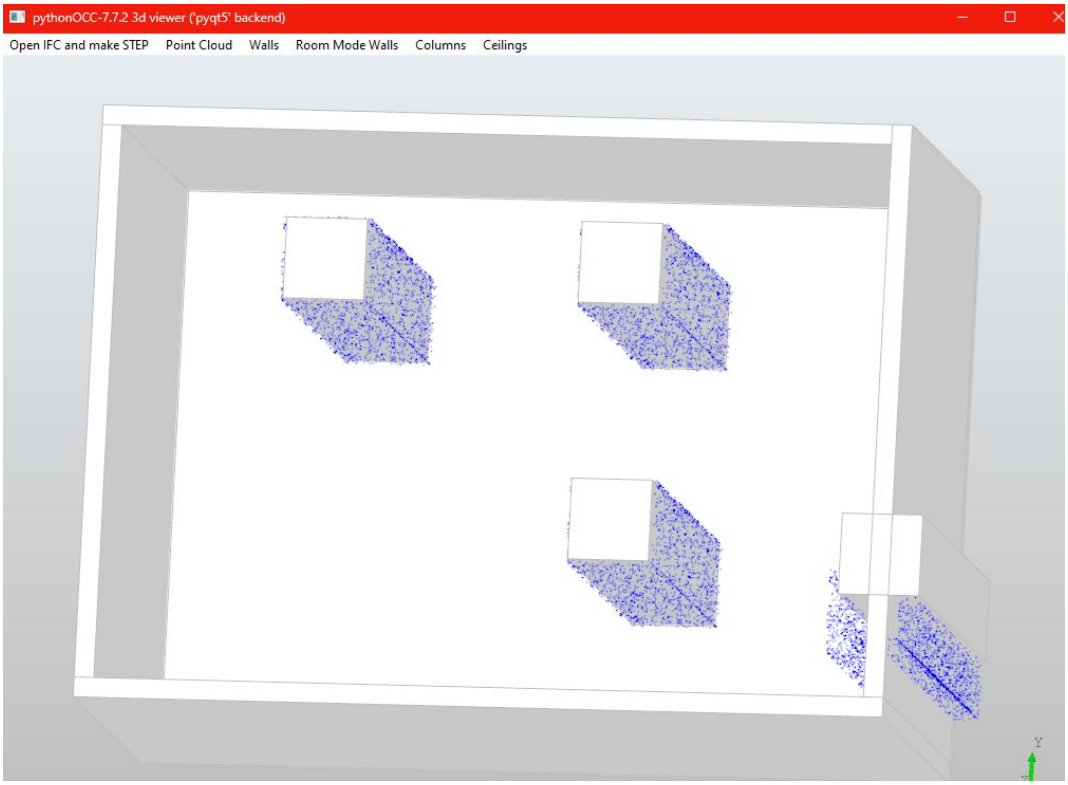

Figure 83: Pop-up warning that tells the user that one column from the original project could not find a match

Third test

The last test is done in a different IFC file, that of room 8.201, which was also customized with some extra columns to test the updating process in a more complex situation. 6 concrete columns are found at the as-designed outdated IFC column, from which 2 are partially embedded columns. Beyond those columns, there are also 3 steel columns with a thinner profile that are fully embedded in the walls and could not be detected by the scanning process. The IFC model with the superimposed point cloud geometry can be seen in Figure 84. Out of the 6 concrete columns in the model, one of the four concrete columns that are not embedded in walls is not found at the current version of the building, so it could not be found at the scan. A warning should be created taking into account that less columns

were found at the building than were at the project, and furthermore that some columns are embedded and could not be checked by point cloud data.

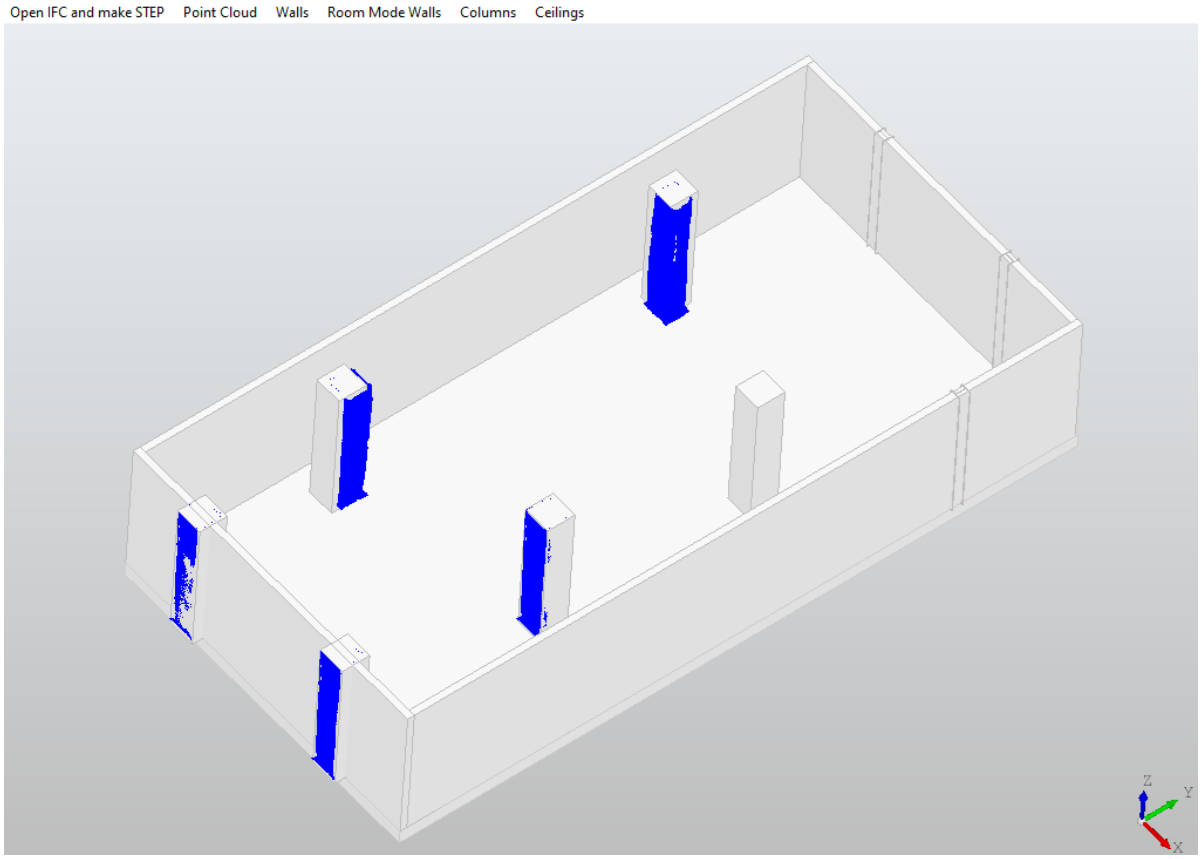

Figure 84: IFC model with two concrete columns partially embedded in a wall, 4 concrete columns not embedded, of which one was not found at the scanning process, and 3 steel columns fully embedded

When running the update, a warning is produced, telling the user that 3 columns are embedded in walls and could not be matched to any point cloud data (the steel columns). The notification also informs that among the IFC columns that are not embedded in walls, one of them could not be found at the real building, asking for extra attention from the responsible engineer. The notification can be seen in Figure 85.

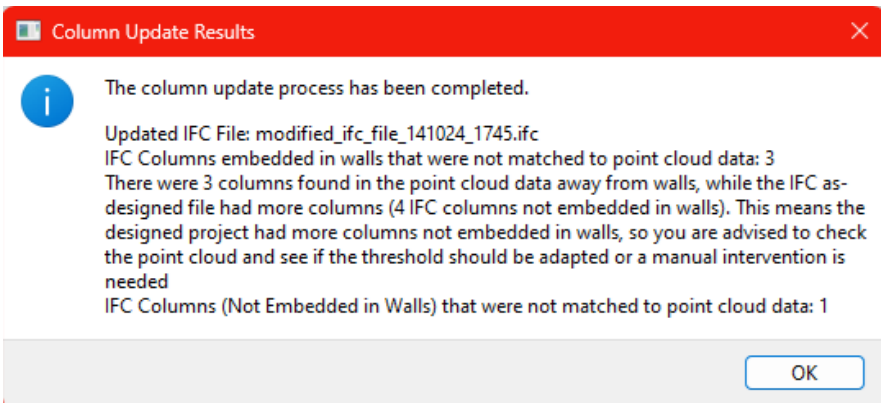

Figure 85: IFC model with two concrete columns partially embedded in a wall, 4 concrete columns not embedded, of which one was not found at the scanning process, and 3 steel columns fully embedded

The model is also updated as expected, with the IfcColumn that found no match at point cloud data, but should have, deleted, and a notification being made about it. Other elements such as embedded columns that could not be detected are kept in the model, but also have their quantity informed. The updated model can be seen in Figure 86.

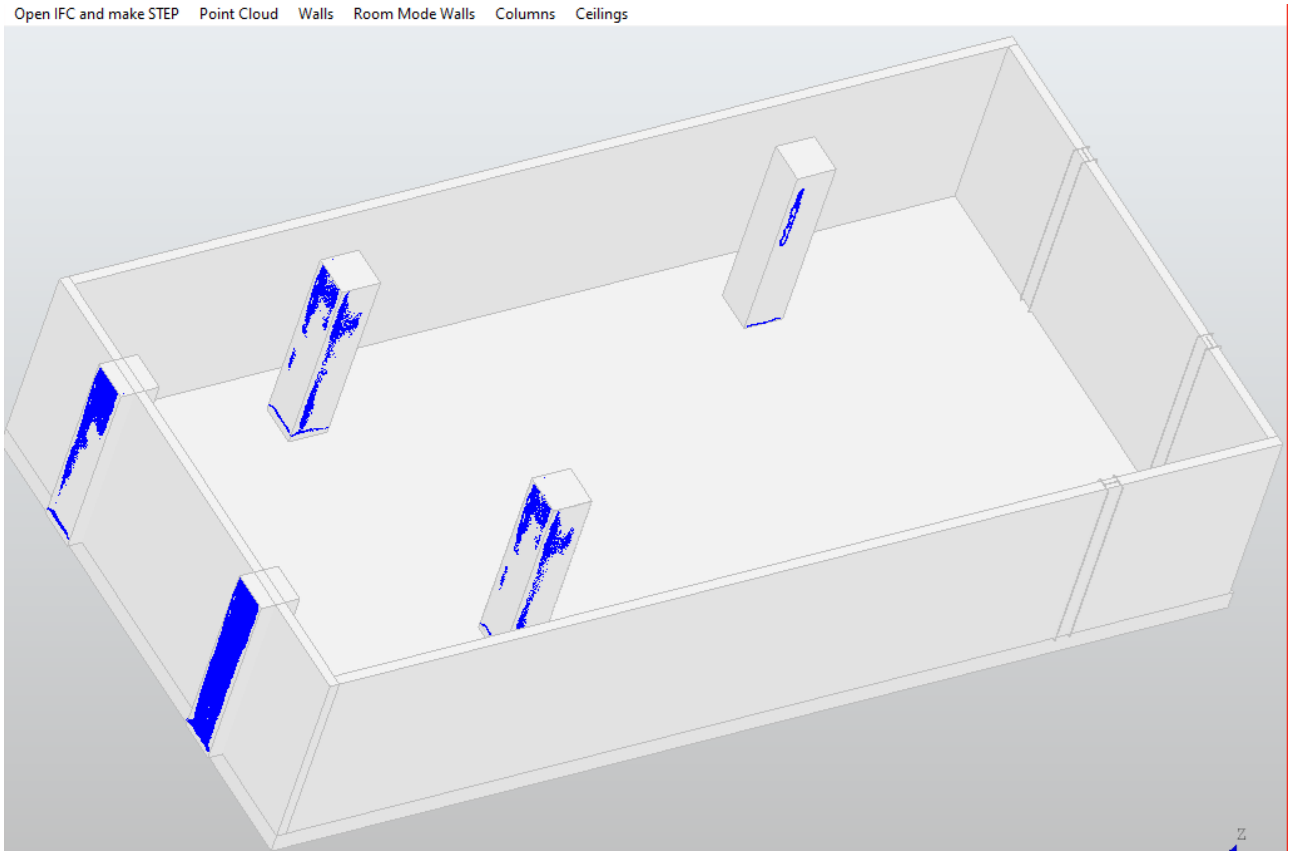

Figure 86: Room 8.201 updated according to the point cloud data and method established

Evaluation of results

All the columns that had to be updated were updated to the exact location expected, that is, the center point of the point cloud column. How precise this update is compared to the real building will depend on the quality of the registration of point cloud and IFC file, and the accuracy of the surveying sensor used, that depends per sensor.

5 Conclusions

5.1 Summary of work, contributions and conclusion of results

The goal of the research done for this thesis was to establish the general process that should be followed when updating an IFC file, making use of segmented point cloud geometry, with an emphasis on the update of walls in a Manhattan-World, multi-room and multi-storey paradigm. To do this, three building element types were chosen to implement the methodology and create a proof-of-concept, namely walls, ceilings, and columns. The general process of updating was divided in (1) Data collection, (2a) Define segmentation standards, (2b) Segment and label building elements, (3) Overlay IFC and point cloud geometry into the same global position, (4) Extract geometric information from segmented elements, (5) Match elements across IFC model and point cloud and (6) Update the model. The special focus of the research was proposing an automated procedure for steps 4 to 6, and demonstrate it for the three building element types chosen. Each building element type requires specific geometric operations to be detected and updated, so those steps can be further broken down into sub-steps that will depend on the element type. Even though the methodology focuses on a Manhattan-World assumption, many principles can also be applied to a non-Manhattan World.

To demonstrate the method created to update models, a python tool was created to implement it, the "Point Cloud to IFC Updater", that is made opensource and available online[3]. It successfully updates the layout of walls in a building receiving as input the segmented walls, and the outdated IFC file. It extracts their geometry, compares that geometry to the geometry of each IFC wall, matching point cloud to IFC walls one by one. The "Point Cloud to IFC Updater" deletes the pre-existing IFC walls that found no match in point cloud data, and creates new IFC walls for point cloud walls that represent geometry that was not found in the outdated model. Besides the creation and deletion process, it creates walls utilizing semantics from the rest of the model, using heuristics such as wall thickness and region of the building the wall is found to look for the best wall type to use as a template, when creating attributes for the new wall. Beyond that, acting after the initial generation of walls, an algorithm was developed to improve the geometry of the new walls, to make them align better with the previously existing geometry of the building. After this alignment is made, an adaptation of this algorithm aligns the new walls to other new walls in case they are connected to each other. On top of that, a special modality of the check and matching of walls, called "Room Mode, was introduced, that allows checking scanned walls by comparing them only to IFC walls of the same region of the building. This allows updating models with only a fragmentary instead of a complete scan of the building. Room Mode was implemented only for walls, but can also be used for other elements. The last developed principle for the update of walls was that of dynamic thresholds, discussed in Section 4.2.2: in buildings with a large variety of wall types, thicknesses and lengths, relatively large thresholds are necessary to match very thick walls, due to differences in definitions of the extents of a wall between IFC and surface-based geometry such as point clouds, and accuracy problems. Very large thresholds can be problematic or inadequate for very thin walls however, so a threshold can be created with a minimum value, applied if walls are thin, and a flexible upper boundary, proportional to the wall's thickness, if the wall is very thick. The highest value is chosen among the two, and this largely improves matching of elements.

[3] https://github.com/jeanvdmeer/The-history-of-the-walls

The research questions were defined as:

1. How does the procedure that automatically updates IFC files of indoor environments based on geometry extracted from segmented and registered point clouds, with an emphasis on the update of walls, and reuse of model semantics, look like?
1.1 How can the implementation of this procedure be made more accessible to users?
1.2 How can the new IFC geometry generated, based on point cloud geometry, be improved and harmonized to the rest of the model?

With pertaining objectives specified as:

1. Prove the feasibility of automated IFC model update based on geometry extracted from segmented point clouds, with emphasis on the update of IfcWallStandardCase elements that follow a Manhattan-World Assumption, and further demonstrations with IfcCovering and IfcColumn elements, using semantics present at the model;
2. Make the implementation of Scan-to-BIM and IFC update more accessible by using data collected from smartphone LiDAR sensors, and reduce the time of surveying by allowing the use of a scanned area of only the section that needs to be updated, instead of a scan of the entire building or floor;
3. Create algorithms that allow the improvement of the initially generated new IFC walls, to harmonize the generated geometry to the previously existing geometry around it e.g. in corners;

Room mode helps in applying cheaper sensors and devices in the use of Scan-to-BIM and IFC model update, allowing a partial scans to be used. As smartphones with LiDAR scanners are usually optimal for scanning a limited area, where they are most accurate, and renovated areas in a project are often limited to only a section of a building, Room Mode helps in solving research question (1.1) and objective (2). The procedure created works just as well for mobile scanners as for professional TLS systems. The corrections of wall geometry, to harmonize their alignment to the previously existing geometry of the building, and compensate for inaccuracies of sensors can solve the research question (1.2) and objective (3). The geometry optimization method can furthermore improve the IFC geometries generated in Scan-to-BIM processes found in the literature such as that of Thomson & Boehm (2015) and Anagnostopoulos et al. (2017).

Much of the literature pertaining creating building elements in IFC takes only a Scan-to-BIM approach, where a fully new IFC file is generated based on point cloud geometry, but all the semantics have to be guessed, or only generic element types are used. When an IFC file is being updated however, it is much more appropriate to take full advantage of the semantics present in the model. When walls were created by the “Point Cloud to IFC Updater”, the best fitting wall already present at the model was used as a template, and all semantics that could be reused from that wall or walls around it were used. Ceilings were updated by correcting their height, and their semantics are kept. An approach was developed to move, create, or delete columns whenever needed. Columns that were encased within walls could not have their position or existence checked, so they were kept in the model. New columns are generated by copying the adequate column type as a template, further preserving semantics. All those approaches fulfil the objectives of research question (1) and objective (1). Out of the 6 steps that define the proposal, steps 4 to 6 were fully automated, which helps to solve the automation element of research question (1). Checks and interventions are further aided by the

reports produced during the column update, that tell the user specific details and results of the test that could affect structural integrity, and by the automated excel report generated by the wall updated process.

One of the contributions of the research was thus a proposed approach of how to deal with the verification and update process of elements that may or may not be encased, or partially encased, in other elements, such as columns. A mixed approach was adopted where encased elements are part of the test, to verify their quantities and produce warnings, while other non-encased elements might be updated. Given that the research available in the update of BIM models is quite limited, and research that goes all the way to implement updates in the IFC schema is even more scarce, some significant contributions could be made to the area of IFC update based on sensing technologies. One of the most relevant researches in the area of IFC update based on sensing technologies is that of Hamledari et al. (2017), where a semi-automated approach was used to update some IFC elements, with the help of user input. The elements updated by that methodology were only elements that had a single geometric representation per element type, such as replacing option (1): a door that has x dimensions, with appearance y, by option (2): a door that has the same x dimensions, with appearance z. Those single representation types could be updated, but the dimensions of the door could not be changed. Other updates done by their methodology are replacing e.g. power outlet types, where one type is replaced by the other. Similarities to this method were that both methods use IfcOpenShell to access the IFC schema, and that the update is performed using the data already contained in the IFC model. This current research could contribute by proposing and demonstrating a procedure to update IfcWallStandardCase entities, where an entity might be represented in many different possible ways, including complex interactions between elements such as connections between three new walls at a single point, plus connections of those walls and other walls, demonstrated in Section 4.2.2. Room Mode, that optimizes the geometry of a point cloud by down-sampling it ant flattening its z-coordinates, to process the geometry of the area scanned and only allow checks to be done in this area, is another important contribution. With Room Mode, the scanning process becomes faster, and therefore cheaper, as the entire building ought not to be scanned, but only the renovated or affected area, and cheaper and more accessible tools such smartphones can be used more easily for scans. Given how much research in Scan-vs-BIM, and even Scan-to-BIM, uses thresholds as one of their fundamental tools, the dynamic thresholds introduced can also be considered as a useful contribution of this research, and for future research and methodologies.

The main contributions can thus be listed as:

1. A method to update IFC elements with multiple possibilities of profile representations for each element type, in this case IfcWallStandardCase entities;
2. Room Mode, a method to perform local checks only in the scanned section of the IFC project;
3. Algorithms to improve the geometry of IFC walls generated and align it to walls around it avoiding e.g. indentations;
4. A method to update IFC columns that takes into account that some of them might be embedded in walls, and a method to update IFC ceiling heights;

Evaluation of Results

For walls, imprecisions in their reconstruction can be caused by either imprecisions of sensors, registration of point cloud, differences of wall definition between surface representations and solid representations, or segmentation. A method was developed to improve the geometry of walls was

however generated, and because it mathematically aligns the walls created to the previously existing walls around it, the alignment technically creates a precision of 100%. The choice of prioritizing vertical and horizontal alignments over orthogonal alignments at corners, described in Section 4.2, greatly helps in making sure that the right alignment is achieved, and indeed it was achieved in all cases tested. However, a central alignment is used, when consecutive walls have different thicknesses, as a choice to have an alignment at one face or the other face would drastically increase complexity and in some cases involve arbitrary choices. A small indentation of half of the difference of the thicknesses of consecutive walls can be present in some cases then, seen in Figure 87.

Figure 87: The alignment algorithm might generate small indentations in some cases, where walls of different thicknesses follow each other

Ceilings and columns have their update based on changing their position, and the center point is found as the center point of the segmented point cloud element. The precision of the update of their location is thus dependent on the precision of the specific sensor used and the precision of registration.

5.2 Limitations and future work

The largest limitation found was the time taken by the python implementation to convert files into STEP and reload them, after changes in the IFC file, or even initially loading the visualization in STEP after opening the IFC file. This could be solved by having an implementation of the procedure that still uses IfcOpenShell in python or C++ to perform the operations in the IFC file, but connect those operations to a web-based visualization, using for instance JavaScript, a language for which IFC visualizers were already developed. Avoiding this intermediary step of converting files of entire models every time an important change is brought to the model would make the implementation much more fluid. A solution could even be made similar to what tools such as Qonic use, where models are edited in real time online, can be accessed anywhere, and have their own engine to process the data based on the IFC schema, but adopt more efficient geometry visualization such as BREP.

There are also some challenges with versioning of elements at the current implementation. If the implementation is based on the paradigm where we have only one IFC file that contains all elements, it still makes some sense to remove non-matched elements from the model. But ideally it would be good to have an online-based implementation, where the non-matched elements are still stored – not visualized as current elements, but stored as elements that are not found at the current version of the building, but can be accessed as older stages of the building.

Another challenge comes from the definition of standards for modelling walls. On one had it makes sense that a model whose geometric representation is designed to take updates into account could be different from a model that is made only to build a new building (as most models now are). In that sense, different definitions for walls that do not make their extents too arbitrary can be good. The definition adopted seems to be one of the best options to make a fair and equivalent comparison between point cloud wall and IFC wall, individually, without taking the entire layout of all walls into

account. Taking the layout of all walls into account can also be tricky if only a part of the building is scanned, and this part is different from the as-designed building. On the other hand, the recommendation to adapt the extents of walls in the model might be hard to implement, and for already existing IFC files it could require changing those files. Both for the case where modelers might design the walls in a standard different than the one specified, and in cases where the IFC file is already existing and does not follow that standard, making the walls conform to the modelling standard could be achieved by an automated script. If the wall entities are defined as walls that have no discontinuities (connections to other walls) at either side of their path, but only at start and end, a script can be made to regenerate the walls, splitting some of them. While it could be argued that this changes the walls and the semantics of the model, the case is made that the semantics are mostly unaffected. Only the number of wall entities changes, and their extents, but the overall mass of walls and wall layout is the same. And at the areas that each wall covers, the properties are still the same. If a wall of layers *x,y* and *z*, with fire-properties *w*, is divided in two, the two parts still have the exact same semantics, and are still located at the exact same place. This is thus mostly an automating problem, which can be solved. Another apparent problem of the method can be seen in Figure 88. When walls are removed, in some cases, an unnecessary division of walls is kept that goes against the very wall modeling definition created. The problem is not too big, as even when the definition of walls are not the same across IFC model and point cloud, when the wall modeling standard is not followed, the procedure still updates the walls and adjusts their geometry well, based on the point cloud geometry and the semantics of the model.

Figure 88: When walls are removed, an unnecessary wall division arises, according to the own standards of wall modelling defined, because the wall that requires that division is not there anymore, as shown in the right side of the figure.

It does seem more ideal though, to have a tool that can automate the standardization of the extents of the walls, and readjust them after the model is updated, or even adapt the procedure to also work with the general layout of walls instead of only a wall-by-wall approach. Taking into account the general layout of the walls can handle some situations, for instance with Room Mode, where only an area of the building is scanned, and the geometry of the other side of the room is not known to help the segmentation tool to follow the standard defined. Taking the general layout of walls into account could be done with help of IFC space boundaries, and the update of IFC spaces can also be handled by using spaces to help take decisions about wall layout. This would help in scaling the method proposed to also include IFC spaces.

Another limitation is that, similarly to the work of Hamledari et al. (2017), the element types for which instances were generated, at least at the implementation of the tool proposed, are only those

that already exist at the model. If a column of different profile is found at the real building, but that profile was not found at the as-designed file, the most similar profile would be used, but not the same. The same applies to walls, if a wall is found at the building site, thicker than those at the IFC model, the one that matches the best will be chosen, but not the exact one. This can be solved by using IfcOpenShel''s API, that allows the creation of generic building elements with customized attributes, for instance, a door of generic and basic appearance and semantics, but that has the dimensions chosen by the user or the script. Another good option is integrating the method with the approach of Valero et al. (2021), where a component database was made, a sort of library of element types that the IFC generating process can use.

The recognition of other elements, and extraction of further semantics of buildings could be done by adding other technologies to the laser-scan process. The integration of the laser-surveying with wire-detector surveying can help in updating or generating IFC models that include embedded elements such as installations. The integration of laser-surveying with thermal analysis, such as proposed by Macher el al. (2019) could also help in detecting and segmenting elements better. And the integration of laser-surveying with 2D photography segmentation can drastically improve detection of some building elements, specially smaller or thinner elements, such as what is done by Apple's RoomPlan. Adding the functionality to update diagonal walls into the current methodology, and better access to the thresholds directly at the interface, so that adjustments can be made more easily, are other suggestions of further research.

6 References

Anagnostopoulos, I., Belsky, M., & Brilakis, I. (Eds.). (2016). Object Boundaries and Room Detection in As-Is BIM Models from Point Cloud Data. *Conference: Proceedings of the 16th International Conference on Computing in Civil and Building Engineering. Osaka, Japan.*

Apple Inc. (2022). *3D Parametric Room Representation with RoomPlan*. Apple Machine Learning Research. https://machinelearning.apple.com/research/roomplan

Apple Inc. (2023). *Explore enhancements to RoomPlan - WWDC23 - Videos - Apple Developer*. Apple Developer. https://developer.apple.com/videos/play/wwdc2023/10192/

Armeni, I., Sener, O., Zamir, A. R., Jiang, H., Brilakis, I., Fischer, M. R., & Savarese, S. (2016). *3D Semantic Parsing of Large-Scale Indoor Spaces*. https://doi.org/10.1109/cvpr.2016.170

Bassier, M., Bonduel, M., Van Genechten, B., & Vergauwen, M. (2017). Segmentation of large unstructured point clouds using octree-based region growing and conditional random fields. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XLII-2/W8*, 25–30. https://doi.org/10.5194/isprs-archives-xlii-2-w8-25-2017

Bassier, M., & Vergauwen, M. (2020). Unsupervised reconstruction of Building Information Modeling wall objects from point cloud data. *Automation in Construction*, *120*, 103338. https://doi.org/10.1016/j.autcon.2020.103338

Becerik-Gerber, B., Jazizadeh, F., Li, N., & Calis, G. (2012). Application areas and data requirements for BIM-Enabled Facilities management. *Journal of the Construction Division and Management*, *138*(3), 431–442. https://doi.org/10.1061/(asce)co.1943-7862.0000433

Beetz, J. , Dietze, S. , Berndt, R. , & Tamke, M. (2013). *Towards The Long-Term Preservation of Building Information Models*. 209–217. http://2013cibw78.civil.tsinghua.edu.cn/

Besl, P. J., & McKay, H. (1992). A method for registration of 3-D shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *14*(2), 239–256. https://doi.org/10.1109/34.121791

BIM Industry Working Group, 2011. *A report for the Government Construction Client Group Building Information Modelling (BIM)*. Working Party Strategy Paper, Communications. London, UK.

BIM Task Group, 2013. Client Guide to 3D Scanning and Data Capture. URL: http://www.bimtaskgroup.org/wp-content/uploads/2013/07/Client-Guide-to-3D-Scanning-and-Data-Capture.pdf

Borrmann, A., König, M., Koch, C., & Beetz, J. (2018). *Building Information Modeling: Technology Foundations and Industry Practice* (1st ed. 2018). Springer.

Bosché, F., Ahmed, M., Turkan, Y., Haas, C. T., and Haas, R. (2015). "The value of integrating Scan-to-BIM and Scan-vs-BIM techniques for construction monitoring using laser scanning and BIM: The case of cylindrical MEP components*." Automation in Construction,* Elsevier B.V., 49, 201–213.

Boulaassal, H., Chevrier, C., & Landes, T. (2010). From laser data to parametric models: towards an automatic method for building façade modelling. In *Lecture Notes in Computer Science* (pp. 42–55). https://doi.org/10.1007/978-3-642-16873-4_4

Bruno, S., De Fino, M., & Fatiguso, F. (2018). Historic Building Information Modelling: performance assessment for diagnosis-aided information modelling and management. *Automation in Construction*, *86*, 256–276. https://doi.org/10.1016/j.autcon.2017.11.009

Cabinet Office, 2011. Government Construction Strategy. Government Construction Strategy. URL:https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/61152/Government-Construction-Strategy_0.pdf

Chase, P., Clarke, K., Hawkes, A., Jabari, S., & Jakus, J. (2022). Apple iPhone 13 Pro LiDAR Accuracy Assessment for Engineering Applications. *2022: The Digital Reality of Tomorrow*. https://doi.org/10.57922/tcrc.645

Chen, J. & Cho, Y. (2018). "Point-to-point Comparison Method for Automated Scan-vs-BIM Deviation Detection." *Proceedings of 17th International Conference on Computing in Civil and Building Engineering, Tampere, Finland*, June 4-7.

Collins, F., Braun, A., & Borrmann, A. (2022). Finding geometric and topological similarities in building elements for large-scale pose updates in Scan-vs-BIM. In *Proc. of the Int. Conf. on Computing in Civil and Building Engineering (ICCCBE).*

Coughlan, J., & Yuille, A. (1999). Manhattan World: compass direction from a single image by Bayesian inference. In Proceedings of the Seventh IEEE International Conference on Computer Vision (pp. 941-947 vol.2). https://doi.org/10.1109/iccv.1999.790349

De Geyter, S., Bassier, M., De Winter, H., & Vergauwen, M. (2022). Review of window and door type detection approaches. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XLVIII-2/W1-2022*, 65–72. https://doi.org/10.5194/isprs-archives-xlviii-2-w1-2022-65-2022

Dehbi, Y., Knechtel, J., Niedermann, B., & Haunert, J. (2022). Incremental constraint-based reasoning for estimating as-built electric line routing in buildings. *Automation in Construction*, *143*, 104571. https://doi.org/10.1016/j.autcon.2022.104571

Díaz-Vilariño, L., Verbree, E., Zlatanova, S., & Diakité, A. A. (2017). Indoor Modelling from SLAM-based laser scanner: door detection to envelope reconstruction. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XLII-2/W7*, 345–352. https://doi.org/10.5194/isprs-archives-xlii-2-w7-345-2017

Domi (2023). *What is the difference between LiDAR and ToF sensor?*. DOMI. https://www.tof-sensor.com/what-is-the-difference-between-lidar-and-tof-sensor-%EF%BC%9F/

Dore, C., & Murphy, M. (2014). Semi-automatic generation of as-built BIM façade geometry from laser and image data. *Journal of Information Technology in Construction (ITcon)*, *19*, 20–46. https://www.itcon.org/paper/2014/2

Dülger, S. (2020). *Using Point Cloud to Automatically Update the BIM 4D Model* (Thesis). https://research.tue.nl/en/studentTheses/using-point-cloud-to-automatically-update-the-bim-4d-model/

El-Din Fawzy, H. (2019). 3D laser scanning and close-range photogrammetry for buildings documentation: A hybrid technique towards a better accuracy. *Alexandria Engineering Journal*, *58*(4), 1191–1204. https://doi.org/10.1016/j.aej.2019.10.003

Esfahani, M. E., Rausch, C. M., Sharif, M. R., Chen, Q., Haas, C. T., & Adey, B. T. (2021). Quantitative investigation on the accuracy and precision of Scan-to-BIM under different modelling scenarios. *Automation in Construction*, *126*, 103686. https://doi.org/10.1016/j.autcon.2021.103686

Frome, D. (2020). *Why Apple chose digital lidar*. Ouster. https://ouster.com/blog/why-apple-chose-digital-lidar/

Frydlewicz, P. (2018, November 10). *LiDAR and ToF Cameras – Technologies explained – ToF-Insights*. https://tof-insights.com/time-of-flight-lidar-and-scanners-technologies-explained/

Furukawa, Y., Curless, B., Seitz, S.M., Szeliski, R., 2009. Reconstructing building interiors from images, in: Cipolla, R., Hebert, M., Tang, X., Yokoya, N. (Eds.), 2009 IEEE 12th International Conference on Computer Vision,. IEEE, Kyoto, Japan, pp. 80–87. DOI: 10.1109/ICCV.2009.5459145

Gyongy, I., Dutton, N., & Henderson, R. (2022). Direct Time-of-Flight Single-Photon Imaging. *IEEE Transactions on Electron Devices*, *69*(6), 2794–2805. https://doi.org/10.1109/ted.2021.3131430

Golparvar-Fard, M., Peña-Mora, F., & Savarese, S. (2009). D4AR–a 4-dimensional augmented reality model for automating construction progress monitoring data collection, processing and communication. *Journal of information technology in construction*, *14*(13), 129-153.

Gourguechon, C., Macher, H., & Landes, T. (2022). Automation of As-Built BIM creation from point cloud: An overview of research works focused on indoor environment. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XLIII-B2-2022*, 193–200. https://doi.org/10.5194/isprs-archives-xliii-b2-2022-193-2022

Hamledari, H., McCabe, B., & Davari, S. (2017). Automated computer vision-based detection of components of under-construction indoor partitions. *Automation in Construction*, *74*, 78–94. https://doi.org/10.1016/j.autcon.2016.11.009

Hamledari, H., Azar, E. R., & McCabe, B. (2018). IFC-Based development of As-Built and As-Is BIMs using construction and facility inspection data: Site-to-BIM Data Transfer Automation. *Journal of Computing in Civil Engineering*, *32*(2). https://doi.org/10.1061/(asce)cp.1943-5487.0000727

Harv, H., Glynn, M., Eckhardt, S., Hadromi, N., Tennyson, R., & Sunesen, S. (2022). *Industry Insight: What do Facility Managers need from BIM? Case 2 – Workplaces in Public Government Workplaces*. buildingSMART International. https://www.buildingsmart.org/standards/bsi-standards/standards-library/#surveys

Hong, S., & Yoshida, J. (2020). Look Inside iPad Pro 11's LiDAR Scanner. *EE Times Asia*. https://www.eetasia.com/look-inside-ipad-pro-11s-lidar-scanner/

Hu, Z., & Brilakis, I. (2023). PriSeg: IFC-Supported Primitive Instance Geometry Segmentation with Unsupervised Clustering. In *Lecture Notes in Computer Science* (pp. 196–211). https://doi.org/10.1007/978-3-031-25082-8_13

Huber, D., Akinci, B., Oliver, A. A., Anil, E., Okorn, B. E., & Xiong, X. (2011, January). Methods for automatically modeling and representing as-built building information models. In *Proceedings of the NSF CMMI Research Innovation Conference* (Vol. 856558). NSF.

Juan, Y., & Hsing, N. (2017). BIM-Based approach to simulate building adaptive performance and life cycle costs for an open building design. *Applied Sciences*, *7*(8), 837. https://doi.org/10.3390/app7080837

Jung, J. H., Stachniss, C., Ju, S., & Heo, J. (2018). Automated 3D volumetric reconstruction of multiple-room building interiors for as-built BIM. *Advanced Engineering Informatics*, *38*, 811–825. https://doi.org/10.1016/j.aei.2018.10.007

Kavaliauskas, P., Fernandez, J. B., McGuinness, K., & Jurelionis, A. (2022). Automation of Construction Progress Monitoring by Integrating 3D Point Cloud Data with an IFC-Based BIM Model. *Buildings*, *12*(10), 1754. https://doi.org/10.3390/buildings12101754

Khalsa, S.J.S., Armstrong, E.M., Hewson, J., Koch, J.F., Leslie, S., Olding, S.W., Doyle, A. (2022). *A Review of Options for Storage and Access of Point Cloud Data in the Cloud*. NASA ESDIS Standards Coordination Office. https://www.earthdata.nasa.gov/s3fs-public/2022-06/ESCO-PUB-003.pdf - https://doi.org/10.5067/DOC/ESO/ESCO-PUB003VERSION1

Khoshelham, K., & Díaz-Vilariño, L. (2014). 3D Modelling of Interior Spaces: Learning the language of Indoor architecture. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XL–5*, 321–326. https://doi.org/10.5194/isprsarchives-xl-5-321-2014

Kim, C., Kim, C., and Son, H. (2013). "Automated construction progress measurement using a 4D building information model and 3D data." *Automation in Construction*, Elsevier B.V., 31, 75–82.

Kloet, F., Hadromi, N., Aiello, A., Fernandez De Sevilla, S., Wei, L., Wagner, F., Assi, R., Hallas, M., & Stonecipher, D. (2022). *Industry Insight: What do Facility Managers need from BIM? Case 5 – Business Catering and openBIM.* buildingSMART International. https://www.buildingsmart.org/standards/bsi-standards/standards-library/#surveys

Lee, T. B. (2020, October 15). *Lidar used to cost $75,000—here's how Apple brought it to the iPhone*. Ars Technica. https://arstechnica.com/cars/2020/10/the-technology-behind-the-iphone-lidar-may-be-coming-soon-to-cars/

Liu, C., Wu, J., & Furukawa, Y. (2018). FloorNet: A Unified Framework for Floorplan Reconstruction from 3D Scans. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1804.00090

Losè, L. T., Spreafico, A., Chiabrando, F., & Tonolo, F. G. (2022). Apple LiDAR Sensor for 3D Surveying: Tests and Results in the Cultural Heritage Domain. *Remote Sensing*, *14*(17), 4157. https://doi.org/10.3390/rs14174157

Macher, H., Landes, T., & Grussenmeyer, P. (2017). From point clouds to building information models: 3D Semi-Automatic reconstruction of indoors of existing buildings. *Applied Sciences*, *7*(10), 1030. https://doi.org/10.3390/app7101030

Macher, H., Boudhaim, M., Grussenmeyer, P., Siroux, M., & Landes, T. (2019). COMBINATION OF THERMAL AND GEOMETRIC INFORMATION FOR BIM ENRICHMENT. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XLII-2/W15*, 719–725. https://doi.org/10.5194/isprs-archives-xlii-2-w15-719-2019

Meyer, T., Brunn, A., & Stilla, U. (2022). Change detection for indoor construction progress monitoring based on BIM, point clouds and uncertainties. *Automation in Construction*, *141*, 104442. https://doi.org/10.1016/j.autcon.2022.104442

Ministerie van Economische Zaken, Landbouw en Innovatie. (2020, January 22). *Maatregelen Klimaatakkoord per sector*. Klimaatverandering | Rijksoverheid.nl. https://www.rijksoverheid.nl/onderwerpen/klimaatverandering/klimaatakkoord/maatregelen-klimaatakkoord-per-sector

Nagel, C., Stadler, A., & Kolbe, T. H. (2009). Conceptual Requirements for the Automatic Reconstruction of Building Information Models from Uninterpreted 3D Models. *ISPRS Archives – Volume XXXVIII-3-4/C3, 2009*, https://www.isprs.org/proceedings/xxxviii/3_4-c3/.

Nikoohemat, S., Peter, M., Elberink, S. O., & Vosselman, G. (2018). Semantic interpretation of mobile laser scanner point clouds in indoor scenes using trajectories. *Remote Sensing*, *10*(11), 1754. https://doi.org/10.3390/rs10111754

Nocerino, E., Lago, F., Morabito, D. D., Remondino, F., Porzi, L., Poiesi, F., Bulò, S. R., Chippendale, P., Locher, A., Havlena, M., Van Gool, L., Eder, M., Fötschl, A., Hilsmann, A., Kausch, L., & Eisert, P. (2017). A SMARTPHONE-BASED 3D PIPELINE FOR THE CREATIVE

INDUSTRY – THE REPLICATE EU PROJECT. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XLII-2/W3*, 535–541. https://doi.org/10.5194/isprs-archives-xlii-2-w3-535-2017

Ochmann, S., Vock, R., Wessel, R., & Klein, R. (2016). Automatic reconstruction of parametric building models from indoor point clouds. *Computers &Amp; Graphics*, *54*, 94–103. https://doi.org/10.1016/j.cag.2015.07.008

Ochmann, S., Vock, R., & Klein, R. (2019). Automatic reconstruction of fully volumetric 3D building models from oriented point clouds. *ISPRS Journal of Photogrammetry and Remote Sensing*, *151*, 251–262. https://doi.org/10.1016/j.isprsjprs.2019.03.017

Oesau, S., Lafarge, F., & Alliez, P. (2014). Indoor scene reconstruction using feature sensitive primitive extraction and graph-cut. *Isprs Journal of Photogrammetry and Remote Sensing*, *90*, 68–82. https://doi.org/10.1016/j.isprsjprs.2014.02.004

Pas, C. C. E. van der (2022). *Achieving the long-term preservation of building data via linked data combined with durable data storage measures* (Thesis). Eindhoven University of Technology. https://research.tue.nl/en/studentTheses/achieving-the-long-term-preservation-of-building-data-via-linked-/

Pereira, F. (2023). *Exploring 3D optical acquisition methods: stereo vision, structured light, time of flight, and LIDAR*. https://www.ml6.eu/blogpost/optical-3d-acquisition-methods-a-comprehensive-guide-part-2

Pintore, G., Mura, C., Ganovelli, F., Fuentes-Perez, L. J., Pajarola, R., & Gobbetti, E. (2020). State-of-the-art in automatic 3D reconstruction of structured indoor environments. *Computer Graphics Forum*, *39*(2), 667–699. https://doi.org/10.1111/cgf.14021

Previtali, M., Díaz-Vilariño, L., & Scaioni, M. (2018). Indoor Building Reconstruction from Occluded Point Clouds Using Graph-Cut and Ray-Tracing. *Applied Sciences*, *8*(9), 1529. https://doi.org/10.3390/app8091529

Quintana, B. H., Prieto, S. A., Adán, A., & Bosché, F. (2018). Door detection in 3D coloured point clouds of indoor environments. *Automation in Construction*, *85*, 146–166. https://doi.org/10.1016/j.autcon.2017.10.016

Rausch, C. M., & Haas, C. T. (2021). Automated shape and pose updating of building information model elements from 3D point clouds. *Automation in Construction*, *124*, 103561. https://doi.org/10.1016/j.autcon.2021.103561

Roman, O. V., Avena, M., Farella, E. M., Remondino, F., & Spano, A. T. (2023). A semi-automated approach to model architectural elements in Scan-to-BIM processes. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XLVIII-M-2–2023*, 1345–1352. https://doi.org/10.5194/isprs-archives-xlviii-m-2-2023-1345-2023

Salazar, G. F., Aboulezz, M., & Alvarez, S. (2019). *Integration of Building Information Modeling and Facilities Management: A Case Study at Worcester Polytechnic Institute*. https://doi.org/10.1061/9780784482421.029

Scherer, R. J., & Katranuschkov, P. (2018). BIMification: How to create and use BIM for retrofitting. *Advanced Engineering Informatics*, *38*, 54–66. https://doi.org/10.1016/j.aei.2018.05.007

Sheik, N. A., Veelaert, P., & Deruyter, G. (2022). Registration of Building Scan with IFC-Based BIM Using the Corner Points. *Remote Sensing*, *14*(20), 5271. https://doi.org/10.3390/rs14205271

Son, H., Bosché, F., and Kim, C. (2015). "As-built data acquisition and its use in production monitoring and automated layout of civil infrastructure: A survey." *Advanced Engineering Informatics*, Elsevier Ltd, 29(2), 172–183.

Spring, A. P. (2020a). A History of Laser Scanning, Part 1: Space and Defense Applications. *Photogrammetric Engineering and Remote Sensing*, *86*(7), 419–429. https://doi.org/10.14358/pers.86.7.419

Spring, A. P. (2020b). History of Laser Scanning, Part 2: The Later Phase of Industrial and Heritage Applications. *Photogrammetric Engineering and Remote Sensing*, *86*(8), 479–501. https://doi.org/10.14358/pers.86.8.479

Tamimi, R. (2022). RELATIVE ACCURACY FOUND WITHIN IPHONE DATA COLLECTION. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XLIII-B2-2022*, 303–308. https://doi.org/10.5194/isprs-archives-xliii-b2-2022-303-2022

Tang, S., Li, X., Zheng, X., Wu, B., Wang, W., & Zhang, Y. (2022). BIM generation from 3D point clouds by combining 3D deep learning and improved morphological approach. *Automation in Construction*, *141*, 104422. https://doi.org/10.1016/j.autcon.2022.104422

Teizer, J., & Kahlmann, T. (2007). Range Imaging as Emerging Optical Three-Dimension Measurement Technology. *Transportation Research Record*, *2040*(1), 19–29. https://doi.org/10.3141/2040-03

Thomson, C., & Boehm, J. (2015). Automatic Geometry Generation from Point Clouds for BIM. *Remote Sensing*, *7*(9), 11753–11775. https://doi.org/10.3390/rs70911753

Thomson, C. (2016). From Point Cloud to Building Information Model: Capturing and Processing Survey Data Towards Automation for High Quality 3D Models to Aid a BIM Process. [Doctoral Thesis, University College London]. DOI:10.13140/RG.2.2.19058.71366/1

Thomson, C. (2018, October 30). *How point clouds are changing BIM, construction & facility maintenance*. Vercator. Retrieved January 25, 2023, from https://info.vercator.com/blog/how-point-clouds-are-changing-bim-construction-and-facility-maintenance

Tran, H., Khoshelham, K., Kealy, A., & Díaz-Vilariño, L. (2019). Shape Grammar approach to 3D modeling of indoor environments using point clouds. *Journal of Computing in Civil Engineering*, *33*(1). https://doi.org/10.1061/(asce)cp.1943-5487.0000800

Tsimpiskakis, G. (2024). *Face-vs-segment comparison: a new method for comparing as-planned and as-built models* (Thesis). https://research.tue.nl/nl/studentTheses/face-vs-segment-comparison

Turkan, Y., Bosche, F., Haas, C. T., & Haas, R. (2012). Automated progress tracking using 4D schedule and 3D sensing technologies. *Automation in Construction,* 22, 414–421. https://doi.org/10.1016/j.autcon.2011.10.003

Tuttas, S., Braun, A., Borrmann, A., & Stilla, U. (2014). Comparision of photogrammetric point clouds with BIM building elements for construction progress monitoring. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences,* XL–3, 341–345. https://doi.org/10.5194/isprsarchives-xl-3-341-2014

Valero, E., Adán, A., & Cerrada, C. (2012). Automatic construction of 3D Basic-Semantic models of inhabited interiors using laser scanners and RFID sensors. *Sensors*, *12*(5), 5705–5724. https://doi.org/10.3390/s120505705

Valero, E., Mohanty, D. D., Ceklarz, M., Tao, B., Bosché, F., Giannakis, G., Fenz, S., Katsigarakis, K., Lilis, G., Rovas, D., & Papanikolaou, A. (2021). An Integrated Scan-to-BIM approach for buildings energy performance evaluation and retrofitting. *38th International Symposium on Automation and Robotics in Construction (ISARC 2021).* https://doi.org/10.22260/isarc2021/0030

Volk, R., Stengel, J., & Schultmann, F. (2014). Building Information Modeling (BIM) for existing buildings — Literature review and future needs. *Automation in Construction*, *38*, 109–127. https://doi.org/10.1016/j.autcon.2013.10.023

Werbrouck, J., Pauwels, P., Bonduel, M., Beetz, J., & Bekers, W. (2020). Scan-to-graph: Semantic enrichment of existing building geometry. *Automation in Construction*, *119*, 103286. https://doi.org/10.1016/j.autcon.2020.103286

Wokke, A. (2017). Google Tango en de Lenovo Phab 2 Pro: eindelijk echte smartphone-innovatie. *Tweakers*. https://tweakers.net/reviews/5369/3/google-tango-en-de-lenovo-phab-2-pro-eindelijk-echte-smartphone-innovatie-tango-op-de-phab-2-pro.html

Xiong, X., Adán, A., Akinci, B., & Huber, D. (2013). Automatic creation of semantically rich 3D building models from laser scanner data. *Automation in Construction*, *31*, 325–337. https://doi.org/10.1016/j.autcon.2012.10.006

Xu, X., Al-Dahle, A., Garg, K. (2020). Shared sensor data across sensor processing pipelines. *U.S. Patent No. 10,671,068.* Washington, DC: U.S. Patent and Trademark Office.

Yang, M. (2022). Apple LIDAR Demystified: SPAD, VCSEL, and Fusion… . *Medium*. https://4sense.medium.com/apple-lidar-demystified-spad-vcsel-and-fusion-aa9c3519d4cb

Appendix I

userInterface.py interface that is run to access all functionalities

```
# This code is part of the Master Thesis of Jean van der Meer presented to the
Eindhoven University of Technology
import os
from PyQt5.QtWidgets import QFileDialog
from OCC.Display.SimpleGui import init_display
from OCC.Extend.DataExchange import read_step_file_with_names_colors
from OCC.Extend.DataExchange import read_step_file_with_names_colors
from OCC.Core.Graphic3d import Graphic3d_ArrayOfPoints
from OCC.Core.AIS import AIS_PointCloud
from OCC.Core.Quantity import Quantity_Color, Quantity_TOC_RGB
import ifcopenshell
import subprocess
# Initialize the display
display, start_display, add_menu, add_function_to_menu = init_display()

# Define global variables
stp_filename = ""
shapes_labels_colors = None


# Function to load and process the STEP file
def load_step_file(file_path):
    global stp_filename, shapes_labels_colors
    stp_filename = file_path
    shapes_labels_colors = read_step_file_with_names_colors(stp_filename)
    print("STEP file loaded successfully:", stp_filename)
    # Always when a new STEP file is loaded the display is cleared and updated
    update_display()

# Function to update the display with the loaded geometry
def update_display():
    global shapes_labels_colors
    display.EraseAll()
    for shape, (_, color) in shapes_labels_colors.items():
        display.DisplayColoredShape(shape, color)

# Function to convert IFC to STEP and load it for visualization of the project
def convert_ifc_to_step_and_load():
    global model
    model = None
    # Use PyQt5 to allow us to find a file with the .ifc extension anywhere in our
computer
    ifc_file_path, _ = QFileDialog.getOpenFileName(None, "Open IFC File", "", "IFC
files (*.ifc)")
    if ifc_file_path:
        # Open the IFC file selected with IfcOpenShell to use and edit information
in it based on the IFC schema
```

```
        ifc_file = ifcopenshell.open(ifc_file_path)
        # Convert the IFC file to a STEP file for later visualization
        step_file_path = ifc_file_path.replace('.ifc', '.stp')
        # To avoid a crash that could be caused if there was already a STEP file
with the same name in the folder
        # (e.g. you are going several tests), replace it automatically
        if os.path.exists(step_file_path):
            os.remove(step_file_path)
        command = f'IfcConvert "{ifc_file_path}" "{step_file_path}"'
        # normally IfcConvert would be run on the command shell, but this can be
automated using subprocess
        subprocess.run(command, shell=True)
        model = ifc_file
        # Load and visualize the converted STEP file
        load_step_file(step_file_path)
        # FitAll adjusts the zoom of the loaded geometry to fit the screen nicely
        display.FitAll()
    return model


def read_point_cloud(file_path):
    # These factors here are just used for the function that visualizes point
clouds in the interface, and not in the semantic processing of point cloud
    # for building update. Because the conversion from IFC to STEP (the latter also
just used for visualization) often changes the units from the IFC file,
    # making the distances much bigger in the step file, usually a order of 1E6 (1
million), here the point cloud is also scaled up by 1E6 to overlap with the
    # BIM geometry data
    scale_factor = 1e6
    points = []
    with open(file_path, 'r') as f:
        for line in f:
            parts = line.split()
            if len(parts) >= 3:  # Ensure there are at least 3 components
                x, y, z = map(float, parts[:3])
                points.append((x * scale_factor, y * scale_factor, z *
scale_factor))
    return points

# Function to display the point cloud
def display_point_cloud(points):
    # This is the function that allows the visualization of point clouds in the
OpenCascade viewer
    n_points = len(points)
    points_3d = Graphic3d_ArrayOfPoints(n_points)
    for point in points:
        x, y, z = point
        points_3d.AddVertex(x, y, z)
    point_cloud = AIS_PointCloud()
    point_cloud.SetPoints(points_3d)
```

```
    #standard color for points is set to blue, other values can be obtained
changing the 0.0, 0.0, 1.0 values below ( that stand for RGB values, respectively)
    blue_color = Quantity_Color(0.0, 0.0, 1.0, Quantity_TOC_RGB)
    point_cloud.SetColor(blue_color)
    point_cloud.SetWidth(5.0) # point size
    ais_context = display.GetContext()
    ais_context.Display(point_cloud, True)
    display.View_Iso()
    display.FitAll()
    print("Point cloud loaded with", len(points), "points.")

def load_point_cloud_file():
    # Initial function accessed from the menu that loads point clouds exclusivelly
for visualization. First PyQt5 allows the user to find the
    # file in their computer, then the points are scaled up to overlap with the
STEP geometry, and then the point cloud is visualized using the OCC
    # function above ( display_point_cloud(points) )

    file_path, _ = QFileDialog.getOpenFileName(None, "Open Point Cloud File", "",
"Point Cloud Files (*.xyz *.txt)")
    if not file_path:
        return

    points = read_point_cloud(file_path)
    display_point_cloud(points)

# Function to load segmented walls
renamed_files = []
def load_segmented_walls():
    # Here point clouds where each one represents a segmented wall are loaded, with
the possibility of multiple walls being loaded at once,
    # to be used in the comparison with as-designed IFC data. This point cloud data
is therefore considered as ground truth and will dictate
    # whether IFC walls are deleted, or if a new IFC wall (or several ones) need to
be added.
    filter = "Text Files (*.txt)"
    file_names, _ = QFileDialog.getOpenFileNames(None, "Select Segmented Walls",
"", filter)
    if not file_names:
        return []
    counter = 1
    for file_name in file_names:
        # All the files are renamed as wall1, wall2, wall3, wall4 etc,
sequentially, for organization reasons and as this naming convention is used
internally
        # in the code. It should be noted however, that it is ok to open walls that
are already named wall1, wall2, wall3, etc, but if e.g. only wall3, wall5
        # and wall5 etc elements are opened/loaded, and there are files named e.g.
wall1 and wall2 at the same folder, the code will try to rename wall3 and wall4
        # into wall1 and wall2 and that will not work, as other files with the same
name already exist in the folder, causing the interface to crash. Therefore
```

```
        # either open all segmented walls, or keep in a different folder those that
already have a name that follow the same convention but shoudln't be opened here
        new_name = f"wall{counter}.txt"
        new_file_path = os.path.join(os.path.dirname(file_name), new_name)
        os.rename(file_name, new_file_path)
        renamed_files.append(new_file_path)
        counter += 1
    print(f"Renamed {len(renamed_files)} files.")
    return renamed_files

# Function to check walls
def check_walls_and_report():
    # Compares point cloud data from segmented walls with the walls of the as-
designed IFC file, and outputs the matched and unmatched walls,
    # producing an Excel report. A simpler version of Room Mode, where the entire
IFC file is checked and liable to updates and deletions
    global model
    from wallChecker import process_seg_walls
    potet1 = process_seg_walls(renamed_files)
    from wallChecker import wallMatcher
    potet2, potet3 = wallMatcher(model = model, wall_dict = potet1)
    from wallChecker import resultsExcel
    resultsExcel(model = model, wall_dict = potet1, ifc_walls_matched = potet2,
point_cloud_walls_matched = potet3)

# Function to update IFC walls
def update_ifc_walls():
    # A simple update of the walls in the model is conducted based on the results
from the check. IFC walls that were not matched to point cloud
    # walls are deleted, and point cloud walls that were not matched to an IFC wall
will generate a new IFC wall. For further explanations check
    # wallMatcher.py and wallCreaTor.py and wallDeleter.py. This produces a simpler
update compared to Room Mode, Room Mode updates the geometry
    # with more adjustments and optimizations, this could be considered a sort of
legacy version of Room Mode. Both here and in Room Mode the walls
    # handled are IfcWallStandardCase entities following a manhattan world
assumption. Diagonal walls might make the script crash.
    global model
    from wallChecker import process_seg_walls
    potet1 = process_seg_walls(renamed_files)
    # match walls to know which ones are matched and therefore which ones should be
deleted (IFC walls) or created (point cloud into ifc)
    from wallChecker import wallMatcher
    potet2, potet3 = wallMatcher(model=model, wall_dict=potet1)

    from wallRemover import wallDeleter
    # first delete all unmatched walls, so that new walls only get connected to
validated pre existing walls
    wallDeleter(model=model, ifc_walls_matched=potet2)
```

```
    # Now we can create new walls based on the point cloud geometry, for walls that
did not exist yet in the IFC model
    # or walls that need a corrected position
    from wallUpdaTor import wallCreaTor
    potet4 = wallCreaTor(model=model, wall_dict=potet1, ifc_walls_matched=potet2,
point_cloud_walls_matched=potet3)

    # Update step file and give it a name with the date and time at the time of
update
    from datetime import datetime
    current_datetime = datetime.now()
    formatted_datetime = current_datetime.strftime("%d%m%y_%H%M")
    step_new_filename = f"updated_model_{formatted_datetime}.stp"

    command = f'IfcConvert "{potet4}" "{step_new_filename}"'
    subprocess.run(command, shell=True)
    load_step_file(step_new_filename)
    display.FitAll()

# Function to load point cloud file and generate alpha hull (the concave hull that
envolves only the scanned area in Room Mode)
def load_total_scanned_area():
    global alpha_hull
    # find the point cloud of the scanned area in any folder
    file_path, _ = QFileDialog.getOpenFileName(None, "Open Point Cloud File", "",
"Point Cloud Files (*.xyz *.txt)")
    if not file_path:
        return
    # here in the read_point_cloud2 a scale of 10E6 (1 million up) is NOT used,
unlike for visualization, because the point cloud is at the same
    # scale as the IFC file, unlike the STEP file that is being visualized
    from wallCheckerRM import read_point_cloud2
    from wallCheckerRM import compute_2d_concave_hull_and_extrude
    points = read_point_cloud2(file_path)
    display_point_cloud(points)
    # Generate the alpha hull
    alpha = 0.5 # Adjust alpha as needed it is a factor that can look for more or
less concavities in the data, 0.5 works in the vast majority of cases
    alpha_hull = compute_2d_concave_hull_and_extrude(points, alpha)
    print("Alpha hull generated.")

# Function to check walls against alpha hull for Room Mode
# Here ifc walls are checked to see if they match point cloud data only within the
volume/region scanned
def check_RM_walls_and_report():
    global model, alpha_hull
    if not alpha_hull:
        print("Alpha hull not generated.")
        return
```

```
    from wallCheckerRM import process_seg_wallsRM, wallMatcherRM, extrPoints,
is_within_alpha_hull
    point_cloud_walls = process_seg_wallsRM(renamed_files)
    # This time, the alpha hull is also used as a an argument for the function, as
the check of walls is only done in the region comprised by the alpha hull
    # furthermore, a buffer size is added, that creates a tolerance around the
scanned region to accept a possible wall start or end that was just outside the
scanned area
    ifc_walls_matched, point_cloud_walls_matched, ifc_walls_to_delete =
wallMatcherRM(model, point_cloud_walls, alpha_hull, buffer_size=0.70)

    from wallCheckerRM import resultsExcel
    # Here an excel report is made of the walls that had to be deleted, had to be
created, and the walls that were kept/matched
    resultsExcel(model = model, wall_dict = point_cloud_walls, ifc_walls_matched =
ifc_walls_matched, point_cloud_walls_matched = point_cloud_walls_matched,
alpha_hull = alpha_hull)
    print("IFC walls to delete:", ifc_walls_to_delete)

# Function to update IFC walls based on alpha hull for Room Mode
# Based on the check to see which IFC walls are inside the alpha hull, those walls
are checked against point cloud data and liable to being matched or deleted
# If necessary, new IFC walls are created based on point cloud data, and they are
enrichted with semantics from the model based on several heuristic principles
# This check and update of geometry at this Room Mode version is much more complex
than the other one and handles many more exceptions and optimizations
def update_RM_ifc_walls():
    global model, alpha_hull
    from wallCheckerRM import process_seg_wallsRM
    point_cloud_walls = process_seg_wallsRM(renamed_files)
    from wallCheckerRM import wallMatcherRM
    ifc_walls_matched, point_cloud_walls_matched, ifc_walls_to_delete =
wallMatcherRM(model, point_cloud_walls, alpha_hull, buffer_size=0.70)
    # The wallMatcherRM function is a bit different from the older wallMatcher
function, and here it also produces an "ifc_walls_to_delete" list,
    # which makes that the wallDeleter function also works a bit differently and
does not parse walls from the entire project but just the preselected ones
    from wallRemoverRM import wallDeleterRM
    wallDeleterRM(model=model, ifc_walls_to_delete = ifc_walls_to_delete)
    from wallUpdaTor import wallCreaTor
    potet4 = wallCreaTor(model=model, wall_dict=point_cloud_walls,
ifc_walls_matched=ifc_walls_matched,
point_cloud_walls_matched=point_cloud_walls_matched)
    from datetime import datetime
    current_datetime = datetime.now()
    formatted_datetime = current_datetime.strftime("%d%m%y_%H%M")
    step_new_filename = f"updated_model_{formatted_datetime}.stp"
    command = f'IfcConvert "{potet4}" "{step_new_filename}"'
    subprocess.run(command, shell=True)
    load_step_file(step_new_filename)
    display.FitAll()
```

```
# Here the ceiling block starts, and similarly as with walls several segmented
ceiling files, in the form of point clouds, can be loaded at once,
# and each file of a ceiling is renamed as ceiling1, ceiling2, ceiling3, etc.
renamed_ceilings = []
def load_segmented_ceilings():
    filter = "Text Files (*.txt)"
    file_names, _ = QFileDialog.getOpenFileNames(None, "Select Segmented Ceilings",
"", filter)
    if not file_names:
        return []
    counter = 1
    for file_name in file_names:
        new_name = f"ceiling{counter}.txt"
        new_file_path = os.path.join(os.path.dirname(file_name), new_name)
        os.rename(file_name, new_file_path)
        renamed_ceilings.append(new_file_path)
        counter += 1
    print(f"Renamed {len(renamed_ceilings)} files.")
    return renamed_ceilings


def check_ceilings_and_update():
    global model
    from ceilingUpdaTor import check_and_update_ceilings, process_seg_ceilings
    # first the segmented ceilings are parsed to extract relevant geometrical
information about them and make a dictionary with each ceiling as an item and
    # relevant data attached to that ceiling also in the dictionary
    pc_ceilings = process_seg_ceilings(renamed_ceilings)
    # then the matching and update of ceiling heights is done based on the geometry
extracted from the point clouds, for more information check ceilingUpdaTor.py
    new_model2 = check_and_update_ceilings(model=model, pc_ceilings=pc_ceilings)
    from datetime import datetime
    current_datetime = datetime.now()
    formatted_datetime = current_datetime.strftime("%d%m%y_%H%M")
    step_new_filename2 = f'updated_model_{formatted_datetime}.stp'
    command = f'IfcConvert "{new_model2}" "{step_new_filename2}"'
    subprocess.run(command, shell=True)
    load_step_file(step_new_filename2)
    display.FitAll()


# Here the column block starts, and similarly as with walls, several segmented
column files, in the form of point clouds, can be loaded at once,
# and each file of a column is renamed as column1, column2, column3, etc
renamed_columns = []
def load_segmented_columns():
    filter = "Text Files (*.txt)"
    file_names, _ = QFileDialog.getOpenFileNames(None, "Select Segmented Columns",
"", filter)
```

```
    if not file_names:
        return []
    counter = 1
    for file_name in file_names:
        new_name = f"column{counter}.txt"
        new_file_path = os.path.join(os.path.dirname(file_name), new_name)
        os.rename(file_name, new_file_path)
        renamed_columns.append(new_file_path)
        counter += 1
    print(f"Renamed {len(renamed_columns)} files.")
    return renamed_columns


# This function deals with the check and update of columns. Columns are a tricky
building element because columns might be embedded in a wall,
# and because current laser scanning techniques see about as much as we can see,
i.e. visible building elements, elements encased in walls or other such
# spaces are not found in scanned data. To take that into account, the check is
done in several stages, first seeing if there are columns in the point
# cloud data that can be matched to encased columns in the as-designed project, but
opting to not change the as-designed data of columns encased in walls
# anyways, but having an idea of how many columns are encased in the ifc project,
and how many were found at the point cloud side, and also how many columns
# are found in the ifc project and how many columns are found in the point cloud
that are reasonably distant from walls, and focusing on that last part,
# perform an update on the position and possibly the quantity of columns that are
not embedded in walls. Because columns are important for the stability
# of a building, a series of reports and warnings are made in the form of pop-up
messages, if for instance less columns are found in the real building,
# comparing it to the as-designed project. Depending on the case, the user is
advised to look at the superimposion of point cloud and IFC geometry and
# possibly do a manual intervention, apart from the automated update
def check_columns_and_update():
    from PyQt5.QtWidgets import QMessageBox, QFileDialog
    global model
    from columnUpdaTor import check_and_update_columns, process_seg_columns

    # Process the segmented columns
    pc_columns = process_seg_columns(renamed_columns)

    # Perform the column check, update and get the results and warnings
    update_results = check_and_update_columns(model=model, pc_columns=pc_columns)

    # Extract results from the dictionary returned by check_and_update_columns
    new_filename = update_results['new_filename']
    num_ifc_emb_columns_no_match = update_results['num_ifc_emb_columns_no_match']
    message2 = update_results['message']
    num_unmatched_free_columns = update_results['num_unmatched_free_ifc_columns']

    # Convert the updated IFC file to STEP format
    from datetime import datetime
```

```
    current_datetime = datetime.now()
    formatted_datetime = current_datetime.strftime("%d%m%y_%H%M")
    step_new_filename3 = f'updated_model_{formatted_datetime}.stp'
    command = f'IfcConvert "{new_filename}" "{step_new_filename3}"'
    subprocess.run(command, shell=True)

    # Load the new STEP file into the viewer
    load_step_file(step_new_filename3)
    display.FitAll()

    # Display a message box with the results
    msg = QMessageBox()
    msg.setIcon(QMessageBox.Information)
    msg.setWindowTitle("Column Update Results")
    msg.setText("The column update process has been completed.")
    msg.setInformativeText(
        f"Updated IFC File: {new_filename}\n"
        f"IFC Columns embedded in walls that were not matched to point cloud data:
{num_ifc_emb_columns_no_match}\n"
        f"{message2}\n"
        f"IFC Columns (Not Embedded in Walls) that were not matched to point cloud
data: {num_unmatched_free_columns}"
    )
    msg.setStandardButtons(QMessageBox.Ok)
    msg.exec_()



if __name__ == "__main__":
    # Show initial instructions pop-up message before starting the display,
otherwise it only shows when you close the display window
    from PyQt5.QtWidgets import QMessageBox, QFileDialog
    msg = QMessageBox()
    msg.setIcon(QMessageBox.Information)
    msg.setWindowTitle("Instructions")
    msg.setText("Instructions:")
    msg.setInformativeText(
        "* As the first step, always open the IFC file at the first menu.\n"
        "* Depending on what you want to update, choose a menu and follow all the
steps of the submenus in the order they are presented at the menu.\n"
        "* The menu 'Point Cloud' serves for visualization of point clouds; they
are not loaded to perform operations and changes in this menu. You can open as many
point clouds at the same time as you wish here."
    )
    msg.setStandardButtons(QMessageBox.Ok)
    msg.exec_()

    # Add menus and functions as submenus
    add_menu('Open IFC and make STEP')
    add_function_to_menu('Open IFC and make STEP', convert_ifc_to_step_and_load)
```

```
add_menu('Point Cloud')
add_function_to_menu('Point Cloud', load_point_cloud_file)
add_menu('Walls')
add_function_to_menu('Walls', load_segmented_walls)
add_function_to_menu('Walls', check_walls_and_report)
add_function_to_menu('Walls', update_ifc_walls)
add_menu('Room Mode Walls')
add_function_to_menu('Room Mode Walls', load_total_scanned_area)
add_function_to_menu('Room Mode Walls', load_segmented_walls)
add_function_to_menu('Room Mode Walls', check_RM_walls_and_report)
add_function_to_menu('Room Mode Walls', update_RM_ifc_walls)
add_menu('Columns')
add_function_to_menu('Columns', load_segmented_columns)
add_function_to_menu('Columns', check_columns_and_update)
add_menu('Ceilings')
add_function_to_menu('Ceilings', load_segmented_ceilings)
add_function_to_menu('Ceilings', check_ceilings_and_update)

start_display()
```

Appendix II

wallChecker.py checks walls when Room Mode is not used

```
# This code is part of the Master Thesis of Jean van der Meer presented to the
Eindhoven University of Technology
import ifcopenshell
import ifcopenshell.geom
import numpy as np
import math
import os
import pandas as pd
import glob


# here the IFC file of the as-designed project is opened, to be compared with the
point clouds,
# have its outdated elements pointed out in a report, and be (later, in a different
import) updated into a new IFC file

##########################################################################################
#####
# First step: find beginning and end coordinates of the walls in the Point cloud
file  #
# It is necessary to classify them in horizontal or vertical in order to be able
to    #
# know which maximum and minimum values to seek and how they will translate into
start #
# and end coordinates of the
wall                                                    #
##########################################################################################
#####

# Here all walls that were segmented are read into the script and saved in a
dictionary
# the segmentation method assumes that each wall is in a separate text file (both
sides of the wall).
# some other methodologies have a point cloud with one of the values per line
pointing out that
# that specific point is a point that belongs to a given wall, or that it belongs
to some other type
# of building element. It is possible to adapt the code into such a methodology,
but that is not the
#  methodology currently adopted. Some reasons for that are discussed in the thesis
report

# point cloud walls are assumed to be named wall1, wall2, wall3, etc for each wall
and each text file
# here, point clouds are used in txt format following the xyz standard, with no
header in the file
# a point cloud in xyz can be used by changing the (prefix + '*.txt') below to
(prefix + '*.xyz')
```

```
# Because the only information relevant for the heuristics used here is the x,y,z geometric information,
# an accompanying file named pcdSimplifier2.py is added to this repository that can clean point cloud files
# from other data and make the files lighter

# In the interface edition of the code, walls are automatically named in the right standard when loaded from
# a folder

import glob
import pandas as pd
import numpy as np
import os

def process_seg_walls(files2):
    data_dict = {}
    wall_dict = {}

    for file in files2:
        # Read the point clouds and store them in data_dict
        data_dict[file] = pd.read_csv(file, sep=' ', header=None, names=['X', 'Y', 'Z', 'R', 'G', 'B'])

    for file, data in data_dict.items():
        x = data['X'].values
        y = data['Y'].values
        z = data['Z'].values
        x_diff = np.abs(x.max() - x.min())
        y_diff = np.abs(y.max() - y.min())

        wall_name = os.path.splitext(os.path.basename(file))[0] # Extract the base name without the extension, as the walls are handled internally as wall1, wall2 etc and not wall1.txt, wall2.txt etc

        if x_diff <= 0.22:
            # this threshold difference at x_diff can be changed, it is mainly placed to estimate the thickness of a wall, and assumes that if the x coordinates
            # of a wall stay constant and within a range that can be considered the thickness of a wall, and the y coordinates change a lot, the wall is vertical
            # seen from a plan view, that is, longitudinally grows along the y axis
            wall_dict[wall_name] = {
                'type': 'vertical',
                'base point': (float(x.mean()), float(y.min()), float(z.min())),
                'end point': (float(x.mean()), float(y.max()), float(z.min())),
                'height': float(z.max()-z.min()),
                'thickness': float(x.max()-x.min()),
                'length': float((y.max() - y.min()))
            }
        elif y_diff <= 0.22:
```

```
            wall_dict[wall_name] = {
                'type': 'horizontal',
                'base point': (float(x.min()), float(y.mean()), float(z.min())),
                'end point': (float(x.max()), float(y.mean()), float(z.min())),
                'height': float(z.max() - z.min()),
                'thickness': float(y.max()-y.min()),
                'length': float((x.max() - x.min()))
            }
        else:
            # Here is where walls that follow a non-manhattan world assumption can
be handled. Their start and end points can be found by a linear approximation of
the
            # x and y coordinates of the segmented wall
            wall_dict[wall_name] = {
                'type': 'diagonal wall',
                'base point': (),
                'end point': (),
                'height': float(z.max()-z.min())
            }

    return wall_dict


# print(wall_dict['wall1']['type'])

##########################################################################################
##########
# Second step: find beginning and end coordinates of the walls in the as-designed
IFC file  #
# Depending on the RefDirection that orients the starting point of a wall, which
can be     #
# random (starting at beginning or end), the end coordinate will be found, based on
the     #
# shape representation polyline that gives the length of the wall; the local
placement, that#
# gives the start coordinate of the wall; and the axis and Reference Direction
of           #
# the wall that transform the local axis into the global
axis                                #
##########################################################################################
##########

import ifcopenshell
import ifcopenshell.util.placement

# Function to extract a wall's start and end points
def extrPoints(wall):
    local_placement = wall.ObjectPlacement
    if local_placement:
        location = local_placement.RelativePlacement.Location.Coordinates
```

```
        # new walls created in the tool may not have an elevation yet at some
checkpoints
        if wall.ContainedInStructure[0].RelatingStructure.Elevation:
            z_coord = wall.ContainedInStructure[0].RelatingStructure.Elevation
        else:
            # newly created walls start with a global z height in their "location"
z coordinate and later get assigned
            # to a floor and don't need a global z height anymore (it is corrected
to relative z), as their height
            # is then referenced by the floor
            z_coord = local_placement.RelativePlacement.Location.Coordinates[2]
        # the length of the wall helps finding the end point of it, that is
implicit in IFC. Points[1] is the second index,
        # so the end point, and the length is given in the x axis, the first axis,
so Coordinates[0]
        lengthW =
wall.Representation.Representations[0].Items[0].Points[1].Coordinates[0]
        # For some horizontal walls RefDirection can be None, so they will not have
an individual DirectionRatio as they
        # already follow the direction of the floor they are located in
        if local_placement.RelativePlacement.RefDirection is not None:
            axis2 = local_placement.RelativePlacement.RefDirection.DirectionRatios
            axis1 = local_placement.RelativePlacement.Axis.DirectionRatios
            # A direction ratio of (-1,0,0) means the wall is horizontal (x axis =
1) but the point stored as
            # location point of the wall is at the end of the wall, when looking at
global coordinates
            if axis1 == (0,0,1) and axis2 == (-1,0,0):
                end_coordinate = ((location[0] - lengthW), location[1],
location[2])
            # A direction ratio of (0,1,0) and (0,-1,0) mean the wall is vertical
(y axis = 1 or -1) for (0,1,0) the
            # location point is in a lower global y value compared to the end of
the wall, and for (0,-1,0) the location/base
            # point of the wall is located on a higher global y value and the end
point at a lower y value than the base point
            elif axis1 == (0,0,1) and axis2 == (0,1,0):
                end_coordinate = (location[0], (location[1] + lengthW),
location[2])
            elif axis1 == (0,0,1) and axis2 == (0,-1,0):
                end_coordinate = (location[0], (location[1] - lengthW),
location[2])
        else:
            end_coordinate = (location[0] +
wall.Representation.Representations[0].Items[0].Points[1].Coordinates[0],
location[1], location[2])
        return (location[0], location[1], z_coord), (end_coordinate[0],
end_coordinate[1], z_coord)
    return None
```

```
##########################################################################################
###########
# Third step: Iterate over walls in the point cloud and check whether any of the
walls       #
# in the IFC file matches this point cloud wall. This is done by checking whether
there is   #
# an absolute difference smaller than e.g. ~0.22m for the Y and X coordinates, of
either     #
# Base Point of Point cloud and base point of ifc wall, AND end point of point
cloud and end #
# point of ifc wall, OR a match within 0.22m of the base point of point cloud and
end point  #
# of ifc wall, AND end point of point cloud and base point of ifc wall, because the
ifc      #
# file does not necessarily consider the wall as always starting from left to right
and      #
# bottom to top as the point cloud files
do.                                                    #
##########################################################################################
###########
# in Room Mode this 0.22 threshold is higher and is also a dynamic threshold,
depending on the thickness of the walls being compared and a minimum value
def wallMatcher(model, wall_dict):
    # from inter5 import model
    point_cloud_walls_matched = []
    ifc_walls_matched = []

    for wall in wall_dict:
        wall_matched = False
        for ifc_wall in model.by_type("IfcWallStandardCase"):
            if (
                (
                    (wall_dict[wall]['base point'][0] < (extrPoints(ifc_wall)[0][0]
+ 0.22) and wall_dict[wall]['base point'][0] > (extrPoints(ifc_wall)[0][0] - 0.22))
and
                    (wall_dict[wall]['base point'][1] < (extrPoints(ifc_wall)[0][1]
+ 0.22) and wall_dict[wall]['base point'][1] > (extrPoints(ifc_wall)[0][1] - 0.22))
and
                    (
                        (wall_dict[wall]['end point'][0] <
(extrPoints(ifc_wall)[1][0] + 0.22) and wall_dict[wall]['end point'][0] >
(extrPoints(ifc_wall)[1][0] - 0.22)) and
                        (wall_dict[wall]['end point'][1] <
(extrPoints(ifc_wall)[1][1] + 0.22) and wall_dict[wall]['end point'][1] >
(extrPoints(ifc_wall)[1][1] - 0.22))
                    )
                ) or
                (
                    (
```

```
                        (wall_dict[wall]['base point'][0] <
(extrPoints(ifc_wall)[1][0] + 0.22) and wall_dict[wall]['base point'][0] >
(extrPoints(ifc_wall)[1][0] - 0.22)) and
                        (wall_dict[wall]['base point'][1] <
(extrPoints(ifc_wall)[1][1] + 0.22) and wall_dict[wall]['base point'][1] >
(extrPoints(ifc_wall)[1][1] - 0.22)) and
                        (
                            (wall_dict[wall]['end point'][0] <
(extrPoints(ifc_wall)[0][0] + 0.22) and wall_dict[wall]['end point'][0] >
(extrPoints(ifc_wall)[0][0] - 0.22)) and
                            (wall_dict[wall]['end point'][1] <
(extrPoints(ifc_wall)[0][1] + 0.22) and wall_dict[wall]['end point'][1] >
(extrPoints(ifc_wall)[0][1] - 0.22))
                        )
                    )
                )
            ):
                print(f'Wall {wall} at the point cloud has matched wall
{ifc_wall.GlobalId} at the IFC file')
                wall_matched = True
                point_cloud_walls_matched.append(wall)
                ifc_walls_matched.append(ifc_wall.GlobalId)

        if not wall_matched:
            # The walls present in the point cloud (as-is / as-built) that were not
matched with the IFC model are
            # new walls or walls with a new configuration, that needs to be
modelled. A report is made, and later in
            # the code, they are updated into the IFC file for some of the use
cases
            print(f'Wall {wall} at the point cloud did not find a match in the IFC
file. It needs to be modeled in the IFC file.')

    for ifc_wall in model.by_type("IfcWallStandardCase"):
        if ifc_wall.GlobalId not in ifc_walls_matched:
            # Walls present in the as-designed model, but that are not found in the
current building, should be deleted from the IFC project
            print(f'Wall {ifc_wall.GlobalId} in the IFC file did not find a match
in the point cloud. It needs to be deleted from the IFC file.')
    return ifc_walls_matched, point_cloud_walls_matched


############################
# Export results to Excel #
# Here, an excel file is generated that has as-designed IFC walls, their global ids
and names to say which ones
# are present in the real building, conforming to the design concept of the
building, and which ones are not.
# furthermore, walls in the as-built/as-is condition of the building that are
present in the point cloud but
```

```
# were not present in the IFC file have their global location outputted so a new
wall can be drawn automatically,
# or by a modeller, if necessary, and checked.

def resultsExcel(model, wall_dict, ifc_walls_matched, point_cloud_walls_matched):

    import openpyxl
    from openpyxl.styles import PatternFill
    from openpyxl.utils import get_column_letter

    # Create a new workbook
    wb = openpyxl.Workbook()
    ws = wb.active

    # Set column headers
    ws['A1'] = 'IFC Wall Name'
    ws['B1'] = 'IFC Wall GUID'
    ws['C1'] = 'Status'

    # Set column widths
    ws.column_dimensions['A'].width = 20
    ws.column_dimensions['B'].width = 40
    ws.column_dimensions['C'].width = 40

    # Set initial row index
    row_index = 2

    # Iterate over IFC walls
    for ifc_wall in model.by_type("IfcWallStandardCase"):
        if ifc_wall.GlobalId in ifc_walls_matched:
            # Match found, set status and fill cell with green color for IFC walls
that found a match
            ws.cell(row=row_index, column=1, value=ifc_wall.Name)
            ws.cell(row=row_index, column=2, value=ifc_wall.GlobalId)
            ws.cell(row=row_index, column=3, value=f'Matched with
{point_cloud_walls_matched[ifc_walls_matched.index(ifc_wall.GlobalId)]}')
            ws.cell(row=row_index, column=3).fill =
PatternFill(start_color="00FF00", end_color="00FF00", fill_type="solid")
        else:
            # No match found, set status and fill cell with red color
            ws.cell(row=row_index, column=1, value=ifc_wall.Name)
            ws.cell(row=row_index, column=2, value=ifc_wall.GlobalId)
            ws.cell(row=row_index, column=3, value='No match found, delete wall')
            ws.cell(row=row_index, column=3).fill =
PatternFill(start_color="FF0000", end_color="FF0000", fill_type="solid")

        row_index += 1

    # Add table of point cloud walls that are matched or not
    row_index += 2
    ws.cell(row=row_index, column=1, value='Point Cloud Wall Name')
```

```
    ws.cell(row=row_index, column=2, value='Matched IFC Wall')
    ws.cell(row=row_index, column=3, value='Coordinates to build new wall, if
needed')

    row_index += 1

    for wall in wall_dict:
        ws.cell(row=row_index, column=1, value=wall)
        if wall in point_cloud_walls_matched:
            ws.cell(row=row_index, column=2,
value=ifc_walls_matched[point_cloud_walls_matched.index(wall)])
        else:
            ws.cell(row=row_index, column=2, value='No match found')
            ws.cell(row=row_index, column=2).fill =
PatternFill(start_color="FF0000", end_color="FF0000", fill_type="solid")
        # round start and end point coordinates to limit cell size
        start_point = [round(coord, 6) for coord in wall_dict[wall]["base point"]]
        end_point = [round(coord, 6) for coord in wall_dict[wall]["end point"]]
        ws.cell(row=row_index, column=3, value=f'Start: {start_point}, End:
{end_point}')

        row_index += 1

    # Set number format for coordinate columns
    for col in ['C']:
        for row in range(2, row_index):
            cell = ws[f'{col}{row}']
            cell.number_format = '0.000000'

    # Save the workbook
    wb.save('wall_matching_results.xlsx')
```

Appendix III

wallCheckerRM.py creates a bounding volume around the scanned are and checks IFC walls therein

```
# This code is part of the Master Thesis of Jean van der Meer presented to the
Eindhoven University of Technology
# Room Mode version of the Wall Checker. The Excel workbook generating function
also has some changes to work with Room Mode.
# The Room Mode functionality is defined here, to check whether the ifc walls that
ought to be checked belong
# to the scanned area (that can be checked) or not.

import numpy as np
import alphashape
from shapely.geometry import Polygon, Point
import pandas as pd
import os
from scipy.spatial import Delaunay
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection

# Function to read point cloud from a file and use it to later find the volume that
bounds the point cloud
def read_point_cloud2(file_path):
    points = []
    with open(file_path, 'r') as f:
        for line in f:
            parts = line.split()
            if len(parts) >= 3:  # Ensure there are at least 3 components
                # for every split line divide the first 3 values as x, y and z and
later assign the tuple of them as a point into a list of points
                x, y, z = map(float, parts[:3])
                points.append((x, y, z))
    return np.array(points)

# Function to downsample the point cloud using voxel grid filtering
# Some point clouds have a very high point density, which makes computation of the
bounding volume of the point cloud slow. The same computation can be
# achieved at a lesser density of points, so that is done here.
def voxel_grid_downsample(points, voxel_size):
    coords_min = np.min(points, axis=0)
    coords_max = np.max(points, axis=0)
    dims = np.ceil((coords_max - coords_min) / voxel_size).astype(int)

    # Calculate voxel indices
    indices = np.floor((points - coords_min) / voxel_size).astype(int)

    # Use np.unique to find unique voxel indices and corresponding points
    _, unique_indices = np.unique(indices, axis=0, return_index=True)

    return points[unique_indices]
```

```
# Function to compute the 2D concave hull, buffer it, and extrude it
# Instead of calculating a bounding volume (hull) around the entire point cloud,
which would again be very computationally expensive,
# the point cloud is "flattened", that is, only the x and y coordinates are used to
calculate a bounding area of the scanned area.
# because people mostly only walk in 2 dimensions in indoor environments, the data
for most indoor environment layouts is expressed in x, y,
# and can be mostly just extruded in the z direction, with ceilings usually having
similar heights. Skipping the computation of this third dimension
# can speed enormously the computation process, from more than 40 minutes
(sometimes even crashes) to around a minute or less for the largest
# project used. In some very specific cases where height of ceilings change a lot,
as in floors where one area has a double height ceiling and the
# other area does not, and only that floor is scanned and other areas shouldn't be
checked or compared to the IFC data of non scanned areas of
# the next floor, a solution with voxelization of the point cloud data to create a
voxelized volume could speed up the computation and still
# take variations in the z coordinate into account.
# A buffer is made around the calculated volume, to find wall starting and end
points that might have fallen just outside the scanned area due
# to imprecisions of the scanning process of discrepancies from as-designed and as-
built measurements. Alpha is a factor that determines how
# small are the concavities that the algorithm should look for when creating a
volume that bounds the points
def compute_2d_concave_hull_and_extrude(points, alpha=1.0, buffer_size=0.4):
    # Project points onto the XY plane (flatten the Z coordinate)
    voxel_size = 0.5
    points = voxel_grid_downsample(points, voxel_size)
    # just the downsampled x and y coordinates of points are used to make the
bounding area
    points_2d = points[:, :2]

    # Compute the 2D concave hull
    hull = alphashape.alphashape(points_2d, alpha)
    hull_polygon = Polygon(hull.exterior.coords)

    # Apply buffer to the hull
    expanded_hull_polygon = hull_polygon.buffer(buffer_size)
    expanded_hull_points = np.array(expanded_hull_polygon.exterior.coords)

    # Get the Z range for extrusion
    z_min = np.min(points[:, 2]) - 0.3 # make sure that the base points of walls
are checked even if the scan is a bit higher than the ifc floor
    z_max = np.max(points[:, 2]) - 0.3 # make sure that the walls of the next floor
are not considered as belonging to the scanned floor

    # Plot the 3D extruded concave hull
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')

    # Create the sides of the extruded shape
```

```
    for i in range(len(expanded_hull_points) - 1):
        x = [expanded_hull_points[i, 0], expanded_hull_points[i + 1, 0],
expanded_hull_points[i + 1, 0], expanded_hull_points[i, 0]]
        y = [expanded_hull_points[i, 1], expanded_hull_points[i + 1, 1],
expanded_hull_points[i + 1, 1], expanded_hull_points[i, 1]]
        z = [z_min, z_min, z_max, z_max]
        ax.add_collection3d(Poly3DCollection([list(zip(x, y, z))], color='cyan',
alpha=0.5))

    # Create the top and bottom faces
    for z in [z_min, z_max]:
        x = expanded_hull_points[:, 0]
        y = expanded_hull_points[:, 1]
        ax.add_collection3d(Poly3DCollection([list(zip(x, y, [z]*len(x)))],
color='cyan', alpha=0.5))

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    # Set default axis limits for visualization
    ax.set_xlim(-25.0, 25.0)
    ax.set_ylim(-25.0, 25.0)
    ax.set_zlim(-12.0, 12.0)
    # visualize bounding hull
    plt.show()

    return expanded_hull_polygon

# Function to check if a point is within the alpha hull
def is_within_alpha_hull(point, alpha_hull_polygon, buffer_size=0.55):
    point_3d = Point(point[:3])
    buffered_polygon = alpha_hull_polygon.buffer(buffer_size)
    return buffered_polygon.contains(point_3d)

# Function to extract a wall's start and end points
def extrPoints(wall):
    local_placement = wall.ObjectPlacement
    if local_placement:
        location = local_placement.RelativePlacement.Location.Coordinates
        # new walls created in the tool may not have an elevation yet at some
checkpoints
        if wall.ContainedInStructure[0].RelatingStructure.Elevation:
            z_coord = wall.ContainedInStructure[0].RelatingStructure.Elevation
        else:
            # newly created walls start with a global z height in their "location"
z coordinate and later get assigned
            # to a floor and don't need a global z height anymore (it is corrected
to relative z), as their height
            # is then referenced by the floor
            z_coord = local_placement.RelativePlacement.Location.Coordinates[2]
```

```
        # the length of the wall helps finding the end point of it, that is
implicit in IFC. Points[1] is the second index,
        # so the end point, and the length is given in the x axis, the first axis,
so Coordinates[0]
        lengthW =
wall.Representation.Representations[0].Items[0].Points[1].Coordinates[0]
        # For some horizontal walls RefDirection can be None, so they will not have
an individual DirectionRatio as they
        # already follow the direction of the floor they are located in
        if local_placement.RelativePlacement.RefDirection is not None:
            axis2 = local_placement.RelativePlacement.RefDirection.DirectionRatios
            axis1 = local_placement.RelativePlacement.Axis.DirectionRatios
            # A direction ratio of (-1,0,0) means the wall is horizontal (x axis =
1) but the point stored as
            # location point of the wall is at the end of the wall, when looking at
global coordinates
            if axis1 == (0,0,1) and axis2 == (-1,0,0):
                end_coordinate = ((location[0] - lengthW), location[1],
location[2])
            # A direction ratio of (0,1,0) and (0,-1,0) mean the wall is vertical
(y axis = 1 or -1) for (0,1,0) the
            # location point is in a lower global y value compared to the end of
the wall, and for (0,-1,0) the location/base
            # point of the wall is located on a higher global y value and the end
point at a lower y value than the base point
            elif axis1 == (0,0,1) and axis2 == (0,1,0):
                end_coordinate = (location[0], (location[1] + lengthW),
location[2])
            elif axis1 == (0,0,1) and axis2 == (0,-1,0):
                end_coordinate = (location[0], (location[1] - lengthW),
location[2])
        else:
            end_coordinate = (location[0] +
wall.Representation.Representations[0].Items[0].Points[1].Coordinates[0],
location[1], location[2])
        return (location[0], location[1], z_coord), (end_coordinate[0],
end_coordinate[1], z_coord)
    return None

# Function to process walls in Room Mode
def process_seg_wallsRM(files2):
    data_dict = {}
    wall_dict = {}

    for file in files2:
        # Read the point clouds and store them in data_dict
        data_dict[file] = pd.read_csv(file, sep=' ', header=None, names=['X', 'Y',
'Z', 'R', 'G', 'B'])

    for file, data in data_dict.items():
        x = data['X'].values
```

```
        y = data['Y'].values
        z = data['Z'].values
        x_diff = np.abs(x.max() - x.min())
        y_diff = np.abs(y.max() - y.min())

        wall_name = os.path.splitext(os.path.basename(file))[0] # Extract the base
name without the extension, as the walls are handled internally as wall1, wall2 etc
and not wall1.txt, wall2.txt etc

        if x_diff <= 0.78 and x_diff < y_diff:
            # The threshold here at room mode is much higher than at the older
wallChecker. that is mainly due to walls tested at bigger projects being often
wider.
            # This threshold difference at x_diff can be changed, it is mainly
placed to estimate the thickness of a wall, and assumes that if the x coordinates
            # of a wall stay constant and within a range that can be considered the
thickness of a wall, and the y coordinates change a lot, the wall is vertical
            # seen from a plan view, that is, longitudinally grows along the y axis
            wall_dict[wall_name] = {
                'type': 'vertical',
                'base point': (float(x.mean()), float(y.min()), float(z.min())),
                'end point': (float(x.mean()), float(y.max()), float(z.min())),
                'height': float(z.max()-z.min()),
                'thickness': float(x.max()-x.min()),
                'length': float((y.max() - y.min()))
            }
        elif y_diff <= 0.78 and y_diff < x_diff:
            wall_dict[wall_name] = {
                'type': 'horizontal',
                'base point': (float(x.min()), float(y.mean()), float(z.min())),
                'end point': (float(x.max()), float(y.mean()), float(z.min())),
                'height': float(z.max() - z.min()),
                'thickness': float(y.max()-y.min()),
                'length': float((x.max() - x.min()))
            }
        else:
            # Here is where walls that follow a non-manhattan world assumption can
be handled. Their start and end points can be found by a linear approximation of
the
            # x and y coordinates of the segmented wall
            wall_dict[wall_name] = {
                'type': 'diagonal wall',
                'base point': (),
                'end point': (),
                'height': float(z.max()-z.min())
            }

    return wall_dict

# Function to match walls in Room Mode
def wallMatcherRM(model, wall_dict, alpha_hull, buffer_size=0.55):
```

```
    point_cloud_walls_matched = []
    ifc_walls_matched = []
    ifc_walls_to_delete = []

    for wall in wall_dict:
        wall_matched = False
        for ifc_wall in model.by_type("IfcWallStandardCase"):
            if
ifc_wall.Representation.Representations[1].Items[0].SweptArea.is_a('IfcRectangleProfileDef'):
                start_point, end_point = extrPoints(ifc_wall)
                # Here a dynamic threshold is created and used to match IFC and
point cloud walls. Because the distance between the start point of a wall
                # in an IFC file and the start point of the same wall in a point
cloud can be quite considerable, taken deviations of measeurement, as-designed
                # vs as-built differences, and differences in how wall connections
and starting points are defined (discussed in the report), this distance
                # can be of around a metre or even a bit more for walls that are 70
cm thick, for instance. It would not be realistic to use a threshold of
                # more than 70 cm for 10cm thick walls however, so a minimum
threshold is defined, that is used for most thin walls, and a threshold
                # proportional to wall thickness is used for walls that are very
thick. To save space the threshold is named dth (Dynamic ThresHold).

                ifc_wall_dim_y =
ifc_wall.Representation.Representations[1].Items[0].SweptArea.YDim
                dth = max(0.65, 2.5*ifc_wall_dim_y)
                # The big OR conditional below defines whether walls are matched
start to start and end to end OR start to end and end to start, because
                # IFC can structure the global coordinates of their wall starts and
ends in a counterintuitive direction, which is converted in IFC
                # by a Reference Direction. Because point clouds always work in the
same coordinate system we need to check for both possibilities though
                # to make sure a match is fully checked
                if is_within_alpha_hull(start_point, alpha_hull, buffer_size) and
is_within_alpha_hull(end_point, alpha_hull, buffer_size):
                    if (
                        (
                            (wall_dict[wall]['base point'][0] < (start_point[0] +
dth) and wall_dict[wall]['base point'][0] > (start_point[0] - dth)) and
                            (wall_dict[wall]['base point'][1] < (start_point[1] +
dth) and wall_dict[wall]['base point'][1] > (start_point[1] - dth)) and
                            (wall_dict[wall]['end point'][0] < (end_point[0] + dth)
and wall_dict[wall]['end point'][0] > (end_point[0] - dth)) and
                            (wall_dict[wall]['end point'][1] < (end_point[1] + dth)
and wall_dict[wall]['end point'][1] > (end_point[1] - dth))
                        ) or
                        (
                            (wall_dict[wall]['base point'][0] < (end_point[0] +
dth) and wall_dict[wall]['base point'][0] > (end_point[0] - dth)) and
```

```
                            (wall_dict[wall]['base point'][1] < (end_point[1] +
dth) and wall_dict[wall]['base point'][1] > (end_point[1] - dth)) and
                            (wall_dict[wall]['end point'][0] < (start_point[0] +
dth) and wall_dict[wall]['end point'][0] > (start_point[0] - dth)) and
                            (wall_dict[wall]['end point'][1] < (start_point[1] +
dth) and wall_dict[wall]['end point'][1] > (start_point[1] - dth))
                        )
                    ):
                        print(f'Wall {wall} at the point cloud has matched wall
{ifc_wall.GlobalId} at the IFC file')
                        wall_matched = True
                        point_cloud_walls_matched.append(wall)
                        ifc_walls_matched.append(ifc_wall.GlobalId)
        if not wall_matched:
            print(f'Wall {wall} at the point cloud did not find a match in the IFC
file. It needs to be modeled in the IFC file.')


    for ifc_wall in model.by_type("IfcWallStandardCase"):
        start_point, end_point = extrPoints(ifc_wall)
        if is_within_alpha_hull(start_point, alpha_hull, buffer_size) and
is_within_alpha_hull(end_point, alpha_hull, buffer_size):
            if ifc_wall.GlobalId not in ifc_walls_matched:
                print(f'Wall {ifc_wall.GlobalId} in the IFC file did not find a
match in the point cloud. It needs to be deleted from the IFC file.')
                ifc_walls_to_delete.append(ifc_wall.GlobalId)
    # here a direct list of walls to be deleted is created, as only walls that are
not checked withing the scanned area should be removed,
    # instead of walls of the entire model that don't find a match with point cloud
data
    return ifc_walls_matched, point_cloud_walls_matched, ifc_walls_to_delete



############################
# Export results to Excel #
# Here, an excel file is generated that has as-designed IFC walls, their global ids
and names to say which ones
# are present in the real building, conforming to the design concept of the
building, and which ones are not.
# furthermore, walls in the as-built/as-is condition of the building that are
present in the point cloud but
# were not present in the IFC file have their global location outputted so a new
wall can be drawn automatically
# or by a modeller, if necessary, and checked.

def resultsExcel(model, wall_dict, ifc_walls_matched, point_cloud_walls_matched,
alpha_hull, buffer_size=0.55):
    # from inter5 import model
    import openpyxl
    from openpyxl.styles import PatternFill
```

```
    from openpyxl.utils import get_column_letter

    # Create a new workbook
    wb = openpyxl.Workbook()
    ws = wb.active

    # Set column headers
    ws['A1'] = 'IFC Wall Name'
    ws['B1'] = 'IFC Wall GUID'
    ws['C1'] = 'Status'

    # Set column widths
    ws.column_dimensions['A'].width = 20
    ws.column_dimensions['B'].width = 40
    ws.column_dimensions['C'].width = 40

    # Set initial row index
    row_index = 2

    # Iterate over IFC walls
    for ifc_wall in model.by_type("IfcWallStandardCase"):
        if
ifc_wall.Representation.Representations[1].Items[0].SweptArea.is_a('IfcRectanglePro
fileDef'):
                start_point, end_point = extrPoints(ifc_wall)
                # only walls within the alpha hull should be marked as not checked,
so the test is done again
                if is_within_alpha_hull(start_point, alpha_hull, buffer_size) and
is_within_alpha_hull(end_point, alpha_hull, buffer_size):

                    if ifc_wall.GlobalId in ifc_walls_matched:
                        # Match found, set status and fill cell with green color
for IFC walls that found a match
                        ws.cell(row=row_index, column=1, value=ifc_wall.Name)
                        ws.cell(row=row_index, column=2, value=ifc_wall.GlobalId)
                        ws.cell(row=row_index, column=3, value=f'Matched with
{point_cloud_walls_matched[ifc_walls_matched.index(ifc_wall.GlobalId)]}')
                        ws.cell(row=row_index, column=3).fill =
PatternFill(start_color="00FF00", end_color="00FF00", fill_type="solid")
                    else:
                        # No match found, set status and fill cell with red color
                        ws.cell(row=row_index, column=1, value=ifc_wall.Name)
                        ws.cell(row=row_index, column=2, value=ifc_wall.GlobalId)
                        ws.cell(row=row_index, column=3, value='No match found,
delete wall')
                        ws.cell(row=row_index, column=3).fill =
PatternFill(start_color="FF0000", end_color="FF0000", fill_type="solid")

                    row_index += 1

    # Add table of point cloud walls that are matched or not
```

```
    row_index += 2
    ws.cell(row=row_index, column=1, value='Point Cloud Wall Name')
    ws.cell(row=row_index, column=2, value='Matched IFC Wall')
    ws.cell(row=row_index, column=3, value='Coordinates to build new wall, if
needed')

    row_index += 1

    for wall in wall_dict:
        ws.cell(row=row_index, column=1, value=wall)
        if wall in point_cloud_walls_matched:
            ws.cell(row=row_index, column=2,
value=ifc_walls_matched[point_cloud_walls_matched.index(wall)])
        else:
            ws.cell(row=row_index, column=2, value='No match found')
            ws.cell(row=row_index, column=2).fill =
PatternFill(start_color="FF0000", end_color="FF0000", fill_type="solid")
        start_point = [round(coord, 6) for coord in wall_dict[wall]["base point"]]
        end_point = [round(coord, 6) for coord in wall_dict[wall]["end point"]]
        ws.cell(row=row_index, column=3, value=f'Start: {start_point}, End:
{end_point}')

        row_index += 1

    # Set number format for coordinate columns
    for col in ['C']:
        for row in range(2, row_index):
            cell = ws[f'{col}{row}']
            cell.number_format = '0.000000'

    # Save the workbook
    wb.save('wall_matching_results.xlsx')
```

Appendix IV

wallRemover.py removes unmatched IFC walls when Room Mode is not used

```
# This code is part of the Master Thesis of Jean van der Meer presented to the
Eindhoven University of Technology
import ifcopenshell
import pandas as pd
import numpy as np
import os as os
import ifcopenshell.api

# Here the deletion of non matched IFC walls is handled for the case where the
entire building is scanned. If the entire building was scanned
# then all walls in the model that are not in the list ifc_walls_matched can be
deleted, which is done here. In Room Mode, where only a specific
# section of the building is scanned, a unique list of walls to be deleted is
produced at the wall matching step, and this list is used at that
# version of the wall deleter function.

def wallDeleter(model, ifc_walls_matched):
    for wall in model.by_type("IfcWallStandardCase"):
        # we want to delete all IfcWalls that did not find a match with a point
cloud wall
        if wall.GlobalId not in ifc_walls_matched:
            # before deleting the wall we need to delete all decompositions of the
wall such as doors, windows and openings it had
            # otherwise if the wall were deleted first we wouldn't be able to
connect their existence to a wall and they
            # would just be unconnnected entities hard to find and making the model
less consistent
            extras2 = ifcopenshell.util.element.get_decomposition(wall)
            print(extras2)
            for extra in extras2:
                # delete the decompositions of the not-matched wall being parsed
                ifcopenshell.api.run("root.remove_product", model, product=extra)

    for wall in model.by_type("IfcWallStandardCase"):
        # after deleting the decompositions the walls without a match can be
deleted
        if wall.GlobalId not in ifc_walls_matched:
            ifcopenshell.api.run("root.remove_product", model, product=wall)
```

Appendix V

wallRemoverRM.py removes unmatched IFC walls in Room Mode

```
# This code is part of the Master Thesis of Jean van der Meer presented to the
Eindhoven University of Technology
import ifcopenshell
import pandas as pd
import numpy as np
import os as os
import ifcopenshell.util.element
import ifcopenshell.api

def wallDeleterRM(model, ifc_walls_to_delete):
    #First, remove all decompositions of the walls
    for wall_id in ifc_walls_to_delete:
        wall = model.by_id(wall_id)
        if wall:
            extras2 = ifcopenshell.util.element.get_decomposition(wall)
            print(f"Decompositions for wall {wall_id}: {extras2}")
            # before deleting the wall we need to delete all decompositions of the
wall such as doors, windows and openings it had
            # otherwise if the wall were deleted first we wouldn't be able to
connect their existence to a wall and they
            # would just be unconnnected entities hard to find and making the model
less consistent
            for extra in extras2:
                try:
                    ifcopenshell.api.run("root.remove_product", model,
product=extra)
                    print(f"Removed decomposition product: {extra}")
                except Exception as e:
                    print(f"Error removing decomposition product {extra}: {e}")

    #Then, remove the walls themselves
    for wall_id in ifc_walls_to_delete:
        wall = model.by_id(wall_id)
        if wall:
            try:
                ifcopenshell.api.run("root.remove_product", model, product=wall)
                print(f"Removed wall: {wall}")
            except Exception as e:
                print(f"Error removing wall {wall_id}: {e}")
```

Appendix VI

wallUpdaTor.py creates new IFC walls whether or not Room Mode is used. Chonky.

```
# This code is part of the Master Thesis of Jean van der Meer presented to the
Eindhoven University of Technology
import uuid
from datetime import datetime
import numpy as np
import ifcopenshell.util.element



def wallCreaTor(model, wall_dict, ifc_walls_matched, point_cloud_walls_matched):
    import math
    from wallCheckerRM import extrPoints
    import ifcopenshell
    import time
    import datetime
    # create datetime data to use in the owner history of the changed items, to
signal when they were edited/created
    dt = datetime.datetime.now()
    dti = int(dt.strftime('%Y%m%d'))

    # Create a new IfcOwnerHistory for new elements to be added to the model
    owner_history = model.create_entity('IfcOwnerHistory')
    owner_history.OwningUser = model.create_entity('IfcPersonAndOrganization')
    owner_history.OwningUser.ThePerson = model.create_entity('IfcPerson')
    owner_history.OwningUser.TheOrganization =
model.create_entity('IfcOrganization')
    owner_history.OwningApplication = model.create_entity('IfcApplication')
    owner_history.State = 'READWRITE'
    # point out that building elements with this owner history were modified
    owner_history.ChangeAction = 'MODIFIED'
    owner_history.LastModifiedDate = int(dti)
    owner_history.LastModifyingUser = owner_history.OwningUser
    owner_history.LastModifyingApplication = owner_history.OwningApplication
    owner_history.LastModifyingApplication.ApplicationFullName = 'Python Scan to
Ifc Updater'
    owner_history.CreationDate = int(dti)

    # Create list for the new walls that will be added into the model, to keep
track of which walls in the
    # model were pre-existing and which ones are new
    new_walls = []

    # Iterate over point cloud walls
    for wall_name, wall_properties in wall_dict.items():
        # Skip if the wall was matched with a as-designed IFC wall
        if wall_name in point_cloud_walls_matched:
```

```
            continue
        # Define functions that will help testing the most appropriate wall in the
model to use as template.
        # The idea is finding a wall of similar thickness nearby, and if that is
not available, look for a wall
        # of similar thickness in the entire model
        def create_bounding_box(center, x_range, y_range, z_range):
            """Create a bounding box around a center point with given ranges. This
can be used to look for IFC
            walls of similar thickness, so a similar walltype, close to the place
where an IFC wall should be created
            on data of a point cloud wall"""
            return {
                'xmin': center[0] - x_range,
                'xmax': center[0] + x_range,
                'ymin': center[1] - y_range,
                'ymax': center[1] + y_range,
                'zmin': center[2] - z_range,
                'zmax': center[2] + z_range
            }

        def is_within_bounding_box(point, box):
            """Check if a point is within the bounding box."""
            return (box['xmin'] <= point[0] <= box['xmax'] and
                    box['ymin'] <= point[1] <= box['ymax'] and
                    box['zmin'] <= point[2] <= box['zmax'])

        # here a bounding box of 4 x 4 x 0,6 m is created around start and end of
the point cloud wall, to look for walls
        # around it and find the one with the closest thickness to the point cloud
wall
        base_box = create_bounding_box(wall_properties['base point'], 2, 2, 0.3)
        end_box = create_bounding_box(wall_properties['end point'], 2, 2, 0.3)

        candidate_walls = []

        for ifc_wall in model.by_type('IfcWallStandardCase'):
            # We want to check whether the wall that might be used as a template to
create a new wall is represented by a rectangular profile,
            # as this is the type of wall that we will create
            if
ifc_wall.Representation.Representations[1].Items[0].SweptArea.is_a('IfcRectanglePro
fileDef'):
                ifc_wall_start, ifc_wall_end = extrPoints(ifc_wall)
                # check if the start or the end of the ifc wall is in the bounding
box around the start of the point cloud wall
                if is_within_bounding_box(ifc_wall_start, base_box) or
is_within_bounding_box(ifc_wall_end, base_box):
                    candidate_walls.append(ifc_wall)
                # check if the start or the end of the ifc wall is in the bounding
box around the end of the point cloud wall
```

```
                elif is_within_bounding_box(ifc_wall_start, end_box) or
is_within_bounding_box(ifc_wall_end, end_box):
                    candidate_walls.append(ifc_wall)
            else:
                continue

        # Select the closest matching wall based on thickness
        closest_wall = None
        # min thickness diff is started as a very high number (infinity) and
interatively uptated to the smallest difference among walls
        min_thickness_diff = float('inf')

        for candidate_wall in candidate_walls:
            candidate_thickness =
candidate_wall.Representation.Representations[1].Items[0].SweptArea.YDim
            thickness_diff = abs(candidate_thickness -
wall_properties['thickness'])
                # is is adopted that an IFC wall that has a difference in thickness
of less than 6 cm should be found around the point cloud wall,
                # otherwise another more fitting wall is searched in the entire
model
            if thickness_diff <= 0.06 and thickness_diff < min_thickness_diff:
                min_thickness_diff = thickness_diff
                closest_wall = candidate_wall

        # if any fitting if wall is found close to it... look at the entire model
        if not closest_wall:
            # If no wall is found within the bounding boxes, find the closest
thickness wall in the entire model
            for ifc_wall in model.by_type('IfcWallStandardCase'):
                if
ifc_wall.Representation.Representations[1].Items[0].SweptArea.is_a('IfcRectanglePro
fileDef'):
                    candidate_thickness =
ifc_wall.Representation.Representations[1].Items[0].SweptArea.YDim
                    thickness_diff = abs(candidate_thickness -
wall_properties['thickness'])
                    if thickness_diff < min_thickness_diff:
                        min_thickness_diff = thickness_diff
                        closest_wall = ifc_wall
                else: continue
        existing_wall = closest_wall

        if existing_wall:
            # Copy a previously existing wall, but most attributes will be empty.
To copy all attributes,
            # ifcopenshell.util.element.copy_deep could be used, but many
attributes would have to be replaced anyways
            # copy_deep is used in the column update module if the user wants to
see an example of the use
            new_wall = ifcopenshell.util.element.copy(model, existing_wall)
```

```
            # Assign a new GUID and the new owner history to the new wall
            new_wall.GlobalId = ifcopenshell.guid.compress(uuid.uuid1().hex)
            new_wall.OwnerHistory = owner_history

            # Determine whether the wall is vertical or horizontal, as different
geometric operations are used for the creation of each one
            dx = max(wall_properties['end point'][0], wall_properties['base
point'][0]) - min(wall_properties['end point'][0], wall_properties['base
point'][0])
            dy = max(wall_properties['end point'][1], wall_properties['base
point'][1]) - min(wall_properties['end point'][1], wall_properties['base
point'][1])
            # also works:
            # if wall_properties['type'] == 'horizontal':
            if abs(dx) > abs(dy):

                ##########################################
                #######  The wall is HORIZONTAL  ##########
                ##########################################

                # Create a Local Placement and fill it with data
                new_placement = model.create_entity('IfcLocalPlacement')
                new_placement.RelativePlacement =
model.create_entity('IfcAxis2Placement3D')
                # Fill the local placement with data. The further entities for
Reference Direction are not created for horizontal walls, as all
                # horizontal walls created by this script have the same (1.0, 0.0,
0.0) direction of the floor the wall is located in, which
                # makes reference directions superfluous, being then set to None.
                new_placement.RelativePlacement.Axis = None # for vertical walls:
model.create_entity('IfcDirection'), shown later

                new_placement.RelativePlacement.RefDirection = None

                # Here the location of the new IFC wall is set as the same as the
point cloud wall. At this point the z coordinate of the wall's location
                # is still the same as the point cloud wall, in global coordinates,
but later it will be set to a height relative to the floor the wall is located
                # located in. That is also why the extrPoints() function has a
special case to handle the position of a wall when it does not yet have a floor
                # assigned to it. Later the x and y coordinates of the location are
also slightly adjusted to smoothen the connection of the wall to other walls.
                new_placement.RelativePlacement.Location =
model.create_entity('IfcCartesianPoint')
                new_placement.RelativePlacement.Location.Coordinates =
wall_properties['base point']

                # set the local placement entities created as the ObjectPlacement
of the new wall
                new_wall.ObjectPlacement = new_placement
```

```
                new_wall.ObjectPlacement.PlacementRelTo =
existing_wall.ObjectPlacement.PlacementRelTo # these semantics can be reused from
the existing template wall

                # Wall Representation
                # The representation of the wall usually has two separate shape
representations. One represents the profile, and the other can be used to represent
                # the outline of the wall, as a polyline. Therefore two
IfcShapeRepresentation entities need to be created and populated with data
                new_representation =
model.create_entity('IfcProductDefinitionShape')
                new_shape_representation1 =
model.create_entity('IfcShapeRepresentation')
                new_shape_representation2 =
model.create_entity('IfcShapeRepresentation')

                # Initialize the Representations attribute as an empty tuple
                new_representation.Representations = ()
                # Add the new shape representations into the tuple of
representations, this needs to be done as a sum of a tuple into the current tuple
value
                new_representation.Representations =
new_representation.Representations + (new_shape_representation1,)
                new_representation.Representations =
new_representation.Representations + (new_shape_representation2,)


                # First geometric representation

                #this first item ContextOfItems can be used from an existing wall
                new_representation.Representations[0].ContextOfItems =
existing_wall.Representation.Representations[0].ContextOfItems
                new_representation.Representations[0].RepresentationIdentifier =
'Axis'
                new_representation.Representations[0].RepresentationType =
'Curve2D'

                new_polyline = model.create_entity('IfcPolyline')
                #initialize the tuple and add the ifc polyline that will represent
the wall
                new_representation.Representations[0].Items = ()
                new_representation.Representations[0].Items =
new_representation.Representations[0].Items + (new_polyline,)
                new_representation.Representations[0].Items[0].Points = ()
                # this new polyline is defined by ifc points, the first one is a
(0,0) internal coordinate that can be used from
                # the existing wall, and the second point represents the length of
the wall internally, on the x coordinate
                new_point1 = model.create_entity('IfcCartesianPoint')
```

```
                new_representation.Representations[0].Items[0].Points =
new_representation.Representations[0].Items[0].Points +
(existing_wall.Representation.Representations[0].Items[0].Points[0],)
                new_representation.Representations[0].Items[0].Points =
new_representation.Representations[0].Items[0].Points + (new_point1,)
                # the first point, after being created and assigned to the tuple of
points, can have a tuple of its coordinates assigned to its coordinates
                new_representation.Representations[0].Items[0].Points[1].Coordinate
s = (wall_properties['length'], 0.0)


                # Second geometric representation
                new_representation.Representations[1].ContextOfItems =
existing_wall.Representation.Representations[1].ContextOfItems
                new_representation.Representations[1].RepresentationIdentifier =
'Body'
                new_representation.Representations[1].RepresentationType =
'SweptSolid'
                new_representation.Representations[1].Items = ()
                # here again the entity for an extruded area representation needs
to be assigned as a tuple adition into the representation items
                new_extruded_area_solid1 =
model.create_entity('IfcExtrudedAreaSolid')
                new_representation.Representations[1].Items =
new_representation.Representations[1].Items + (new_extruded_area_solid1,)
                new_representation.Representations[1].Items[0].Depth =
wall_properties['height'] # the height is refined later to equal that of walls
around it if it should
                #use the same extruded direction from an existing wall, to not have
to create a new ifc entity instance, as they are the same for all walls
                new_representation.Representations[1].Items[0].ExtrudedDirection =
existing_wall.Representation.Representations[1].Items[0].ExtrudedDirection
                #definition of the profile area of the wall
                new_representation.Representations[1].Items[0].SweptArea =
model.create_entity('IfcRectangleProfileDef')
                new_representation.Representations[1].Items[0].SweptArea.ProfileTyp
e = 'AREA'
                new_representation.Representations[1].Items[0].SweptArea.XDim =
float(wall_properties['length'])
                new_representation.Representations[1].Items[0].SweptArea.YDim =
float(wall_properties['thickness'])
                new_representation.Representations[1].Items[0].SweptArea.Position =
model.create_entity('IfcAxis2Placement2D')
                new_representation.Representations[1].Items[0].SweptArea.Position.L
ocation = model.create_entity('IfcCartesianPoint')
                #this rectangle that represents the wall has its internal location
point in the centre of the rectangle, so length divided by 2
                new_representation.Representations[1].Items[0].SweptArea.Position.L
ocation.Coordinates = (float(wall_properties['length'])/2, 0.0)
                new_representation.Representations[1].Items[0].SweptArea.Position.R
efDirection =
```

```
existing_wall.Representation.Representations[1].Items[0].SweptArea.Position.RefDire
ction
                new_representation.Representations[1].Items[0].Position =
existing_wall.Representation.Representations[1].Items[0].Position #is this right?
update: yea ig


                #creation of an IfcStyledItem entity, it is not part of the wall,
but mentions the wall in its attributes
                new_styled_item = model.create_entity('IfcStyledItem')
                #here under, mention in the IfcStyledItem entity is made to the
representation of the new wall
                new_styled_item.Item =
new_representation.Representations[1].Items[0]
                new_styled_item.Styles = ()
                new_styles1 = model.create_entity('IfcPresentationStyleAssignment')
                new_styled_item.Styles = new_styled_item.Styles + (new_styles1,)
                new_styled_item.Styles[0].Styles = ()
                new_surface_style1 = model.create_entity('IfcSurfaceStyle')
                new_styled_item.Styles[0].Styles = new_styled_item.Styles[0].Styles
+ (new_surface_style1,)
                #from here on, the attributes of the styledItem are identical to
other walls of the same type, so they can be copied from an existing wall
                new_styled_item.Styles[0].Styles =
existing_wall.Representation.Representations[1].Items[0].StyledByItem[0].Styles[0].
Styles

                new_wall.Representation = new_representation
                #new_wall.Tag = 'Tag'


                # A number of wall attributes are found in wall.IsDefinedBy, that
are usually shared among all internal walls
                # that includes visualization properties, attributes that determine
whether the wall connects to the ceiling
                # or not, whether the wall is an interior wall or not, loadbearing,
etc. Those attributes can be copied from the
                # exinsting wall being used as reference, but if a given value or
parameter needs to be added to new walls,
                # the code can be changed here to include this option. There are a
number of IfcRelDefinesByProperties
                # instances that connect those attributes to one of the existing
walls. What the code does here is to include
                # the new wall into the tuple of walls (previously just one) that
are receiving each attribute. So in a way
                # the definition of some generic properties of the template wall
are expanded to apply also to the new wall.
                # To expand this code and make it more complete it could create
individual instances  of those definitions to
                # say if the wall is load bearing or not, etc. But as this is not
important for many purposes those properties
```

```
                # were copied for the sake of brevity.

                # An iteration is done in reverse order, as it was observed that if
the iteration altering each item is done in normal
                # order, the index of the altered item changes to the last index of
the list, messing the orders of indexes and not updating all items.
                # Therefore, if the count starts at the last index, it remains
last, then it goes to index n-1, and sends n-1 into last position
                # but it keeps feeding the parsing process with indexes whose order
was not yet scrambled and allows all items to be changed only once

                # Iterate over the IsDefinedBy attributes of the existing wall in
reverse order
                for i in range(len(existing_wall.IsDefinedBy)-1, -1, -1):
                    # Create a new tuple that includes all the existing objects
plus the new object
                    new_related_objects =
existing_wall.IsDefinedBy[i].RelatedObjects + (new_wall,)
                    # Assign the new tuple to the RelatedObjects attribute
                    existing_wall.IsDefinedBy[i].RelatedObjects =
new_related_objects

                #wall material association, add new wall to the list of other walls
with the same material
                existing_wall.HasAssociations[0].RelatedObjects =
existing_wall.HasAssociations[0].RelatedObjects + (new_wall,)


            else:
                ##################################################
                # ########## The wall is VERTICAL ############# #
                ##################################################

                #Local Placement
                new_placement = model.create_entity('IfcLocalPlacement')
                new_placement.RelativePlacement =
model.create_entity('IfcAxis2Placement3D')
                # for vertical walls there shall be axes that give it a Reference
Direction to convert the internal coordinates into the
                # coordinates of the context, of the floor the wall is located in
                new_placement.RelativePlacement.Axis =
model.create_entity('IfcDirection')
                new_placement.RelativePlacement.Axis.DirectionRatios = (0., 0., 1.)
                new_placement.RelativePlacement.RefDirection =
model.create_entity('IfcDirection')
                # (0.0, 1.0, 0.0) is chosen as the standard reference direction for
vertical walls as walls that start at the lowest global y coordinate
                # and end at the highest global y coordinate
                new_placement.RelativePlacement.RefDirection.DirectionRatios = (0.,
1., 0.)
```

```
                new_placement.RelativePlacement.Location =
model.create_entity('IfcCartesianPoint')
                new_placement.RelativePlacement.Location.Coordinates =
wall_properties['base point']
                #new_placement.PlacesObject[0] = (new_wall,) -> not necessary

                new_wall.ObjectPlacement = new_placement



                new_wall.ObjectPlacement.PlacementRelTo =
existing_wall.ObjectPlacement.PlacementRelTo # can be used as the same of an
existing wall

                # Wall Representation
                # The representation of the wall usually has two separate shape
representations. One represents the profile, and the other can be used to represent
                # the outline of the wall, as a polyline. Therefore two
IfcShapeRepresentation entities need to be created and populated with data
                new_representation =
model.create_entity('IfcProductDefinitionShape')
                new_shape_representation1 =
model.create_entity('IfcShapeRepresentation')
                new_shape_representation2 =
model.create_entity('IfcShapeRepresentation')
                # Initialize the Representations attribute as an empty tuple
                new_representation.Representations = ()
                # Add the new shape representations into the tuple of
representations, this needs to be done as a sum of a tuple into the current tuple
value
                new_representation.Representations =
new_representation.Representations + (new_shape_representation1,)
                new_representation.Representations =
new_representation.Representations + (new_shape_representation2,)


                # First geometric representation

                #this first item ContextOfItems can be used from an existing wall
                new_representation.Representations[0].ContextOfItems =
existing_wall.Representation.Representations[0].ContextOfItems
                new_representation.Representations[0].RepresentationIdentifier =
'Axis'
                new_representation.Representations[0].RepresentationType =
'Curve2D'
                new_polyline = model.create_entity('IfcPolyline')
                new_representation.Representations[0].Items = ()
                new_representation.Representations[0].Items =
new_representation.Representations[0].Items + (new_polyline,)
                new_representation.Representations[0].Items[0].Points = ()
```

```
                # this new polyline is defined by ifc points, the first one is a
(0,0) internal coordinate that can be used from
                # the existing wall, and the second point represents the length of
the wall internally, on the x coordinate
                new_point1 = model.create_entity('IfcCartesianPoint')
                new_representation.Representations[0].Items[0].Points =
new_representation.Representations[0].Items[0].Points +
(existing_wall.Representation.Representations[0].Items[0].Points[0],)
                new_representation.Representations[0].Items[0].Points =
new_representation.Representations[0].Items[0].Points + (new_point1,)
                new_representation.Representations[0].Items[0].Points[1].Coordinate
s = (wall_properties['length'], 0.0)

                # second geometric representation
                new_representation.Representations[1].ContextOfItems =
existing_wall.Representation.Representations[1].ContextOfItems
                new_representation.Representations[1].RepresentationIdentifier =
'Body'
                new_representation.Representations[1].RepresentationType =
'SweptSolid'
                new_representation.Representations[1].Items = ()
                new_extruded_area_solid1 =
model.create_entity('IfcExtrudedAreaSolid')
                # here again the entity for an extruded area representation needs
to be assigned as a tuple adition into the representation items
                new_representation.Representations[1].Items =
new_representation.Representations[1].Items + (new_extruded_area_solid1,)
                new_representation.Representations[1].Items[0].Depth =
wall_properties['height']  # the height is refined later to equal that of walls
around it if it should
                #use the same extruded direction from an existing wall, to not have
to create a new ifc entity instance, as they are the same for all walls
                new_representation.Representations[1].Items[0].ExtrudedDirection =
existing_wall.Representation.Representations[1].Items[0].ExtrudedDirection
                # definition of the profile area of a wall
                new_representation.Representations[1].Items[0].SweptArea =
model.create_entity('IfcRectangleProfileDef')
                new_representation.Representations[1].Items[0].SweptArea.ProfileTyp
e = 'AREA'
                new_representation.Representations[1].Items[0].SweptArea.XDim =
float(wall_properties['length'])
                new_representation.Representations[1].Items[0].SweptArea.YDim =
float(wall_properties['thickness'])
                new_representation.Representations[1].Items[0].SweptArea.Position =
model.create_entity('IfcAxis2Placement2D')
                new_representation.Representations[1].Items[0].SweptArea.Position.L
ocation = model.create_entity('IfcCartesianPoint')
                #this rectangle that represents the wall has its internal location
point in the centre of the rectangle, so length divided by 2
                new_representation.Representations[1].Items[0].SweptArea.Position.L
ocation.Coordinates = (float(wall_properties['length'])/2, 0.0)
```

```
                new_representation.Representations[1].Items[0].SweptArea.Position.R
efDirection =
existing_wall.Representation.Representations[1].Items[0].SweptArea.Position.RefDire
ction
                new_representation.Representations[1].Items[0].Position =
existing_wall.Representation.Representations[1].Items[0].Position

                #creation of an IfcStyledItem entity, it is not part of the wall,
but mentions the wall in its attributes
                new_styled_item = model.create_entity('IfcStyledItem')
                #here under, mention in the IfcStyledItem entity is made to the
representation of the new wall
                new_styled_item.Item =
new_representation.Representations[1].Items[0]
                new_styled_item.Styles = ()
                new_styles1 = model.create_entity('IfcPresentationStyleAssignment')
                new_styled_item.Styles = new_styled_item.Styles + (new_styles1,)
                new_styled_item.Styles[0].Styles = ()
                new_surface_style1 = model.create_entity('IfcSurfaceStyle')
                new_styled_item.Styles[0].Styles = new_styled_item.Styles[0].Styles
+ (new_surface_style1,)
                #from here on, the attributes of the styledItem are identical to
other walls of the same type, so they can be copied from an existing wall
                new_styled_item.Styles[0].Styles =
existing_wall.Representation.Representations[1].Items[0].StyledByItem[0].Styles[0].
Styles



                new_wall.Representation = new_representation







                # A number of wall attributes are found in wall.IsDefinedBy, that
are usually shared among all internal walls
                # that includes visualization properties, attributes that determine
whether the wall connects to the ceiling
                # or not, whether the wall is an interior wall or not, loadbearing,
etc. Those attributes can be copied from the
                # exinsting wall being used as reference, but if a given value or
parameter needs to be added to new walls,
                # the code can be changed here to include this option. There are a
number of IfcRelDefinesByProperties
                # instances that connect those attributes to one of the existing
walls. What the code does here is to include
                # the new wall into the tuple of walls (previously just one) that
are receiving each attribute. So in a way
                # the definition of some generic properties of the template wall
are expanded to apply also to the new wall.
```

```
                # To expand this code and make it more complete it could create
individual instances  of those definitions to
                # say if the wall is load bearing or not, etc. But as this is not
important for many purposes those properties
                # were copied for the sake of brevity.

                # An iteration is done in reverse order, as it was observed that if
the iteration altering each item is done in normal
                # order, the index of the altered item changes to the last index of
the list, messing the orders of indexes and not updating all items.
                # Therefore, if the count starts at the last index, it remains
last, then it goes to index n-1, and sends n-1 into last position
                # but it keeps feeding the parsing process with indexes whose order
was not yet scrambled and allows all items to be changed only once

                # Iterate over the IsDefinedBy attributes of the existing wall in
reverse order
                for i in range(len(existing_wall.IsDefinedBy)-1, -1, -1):
                    # Create a new tuple that includes all the existing objects
plus the new object
                    new_related_objects =
existing_wall.IsDefinedBy[i].RelatedObjects + (new_wall,)
                    # Assign the new tuple to the RelatedObjects attribute
                    existing_wall.IsDefinedBy[i].RelatedObjects =
new_related_objects


                # if the methodology of updating the indexes in the way below,
commented, were to be used, some errors are obtained
                # because the indexes in IsDefinedBy are rearranged for each line
performed. In the end, some of the original
                # indexes end up with too much new information (new_wall assigned
to it multiple times), and some with no information
                    # wrong methodology:
                # existing_wall.IsDefinedBy[0].RelatedObjects =
existing_wall.IsDefinedBy[0].RelatedObjects + (new_wall,)
                # existing_wall.IsDefinedBy[1].RelatedObjects =
existing_wall.IsDefinedBy[1].RelatedObjects + (new_wall,)
                # existing_wall.IsDefinedBy[2].RelatedObjects =
existing_wall.IsDefinedBy[2].RelatedObjects + (new_wall,)
                # existing_wall.IsDefinedBy[3].RelatedObjects =
existing_wall.IsDefinedBy[3].RelatedObjects + (new_wall,)
                # existing_wall.IsDefinedBy[4].RelatedObjects =
existing_wall.IsDefinedBy[4].RelatedObjects + (new_wall,)
                # existing_wall.IsDefinedBy[5].RelatedObjects =
existing_wall.IsDefinedBy[5].RelatedObjects + (new_wall,)

                #wall material association, add new wall to the list of other walls
with the same material
                existing_wall.HasAssociations[0].RelatedObjects =
existing_wall.HasAssociations[0].RelatedObjects + (new_wall,)
```

```
            # Modify the other properties of the new wall and improve geometry

            #Add new wall into the elements listed in its Level
            # Get the z-coordinate of the base point of the new wall
            z_new_wall = float(wall_properties['base point'][2])

            # Iterate over all existing walls
            for existing_wall in model.by_type('IfcWallStandardCase'):
                if new_wall != existing_wall and existing_wall not in new_walls:
#new_walls are not assigned to a floor yet so we do not want to use those as
template

                    if
existing_wall.Representation.Representations[1].Items[0].SweptArea.is_a('IfcRectang
leProfileDef'):
                    # Check if the existing wall has a relative placement
coordinate
                        if existing_wall.ObjectPlacement:
                            # Get the z-coordinate of the relative placement
coordinate
                            # Worth mentioning that in extrPoints, for existing
walls, the z coordinate retrieved is not the one from
ObjectPlacement.RelativePlacement.Location.Coordinates but the one
                            # in
wall.ContainedInStructure[0].RelatingStructure.Elevation. This is done so that the
"global height" of the existing wall can be compared
                            # to the height of the new wall, that until then was in
global coordinates, and then after the comparison, here, the new wall gets the
right
                            # z coordinate assigned to its ...Location.Coordinates,
which will be around 0.0 instead of e.g. 3.8 or 4.0 etc.
                            # The extrPoints function checks if a wall is new or
existing, the existing wall gets checked as just mentioned for the z coordinate
                            # and a new wall that has no ContainedInStructure gets
the z coordinate from its ...Location.Coordinates, as it still has its global Z
there until that point
                            z_existing_wall = extrPoints(existing_wall)[0][2]
                            # Check if the existing wall starting point elevation
coordinate is close to the new wall
                            if abs(z_new_wall - z_existing_wall ) <= 0.35:
                                #add new wall to the list of walls in the same
level
                                existing_wall.ContainedInStructure[0].RelatedElemen
ts = existing_wall.ContainedInStructure[0].RelatedElements + (new_wall,)
                                #copy the same z coordinate to ensure the new wall
starts at the same level as other walls around it, see third value of the tuple ->
```

```
                                new_wall.ObjectPlacement.RelativePlacement.Location
.Coordinates = (new_wall.ObjectPlacement.RelativePlacement.Location.Coordinates[0],
new_wall.ObjectPlacement.RelativePlacement.Location.Coordinates[1],
existing_wall.ObjectPlacement.RelativePlacement.Location.Coordinates[2])
                                # set the same height of the wall into walls around
it if the difference in height is not too great, to have the geometry of the walls
aligned on top
                                if
abs(new_wall.Representation.Representations[1].Items[0].Depth -
existing_wall.Representation.Representations[1].Items[0].Depth) <= 0.3:
                                    new_wall.Representation.Representations[1].Item
s[0].Depth = existing_wall.Representation.Representations[1].Items[0].Depth

                                break
                    else: continue



            # it is important to add connections only after all new walls are
created because some connections
            # might be with new walls that otherwise did not exist yet at the time
of the iteration
            new_walls.append(new_wall)



    # Code to refine the geometry of wall connections
    import math

    def euclidean_distance(point1, point2):
        return math.sqrt(sum((a - b) ** 2 for a, b in zip(point1, point2)))

    # The code here aims to improve the geometry of the newly created walls, to
compensate for imprecisions of the LiDAR scanner and differences
    # in how the start and end of a wall are defined in a point cloud and in an IFC
file. The walls at the IFC file are defined as extrusions of profiles,
    # as a solid, usually and extruded rectangle, and the walls in a point cloud
are segmented as planes of the visible surfaces of the wall, usually just
    # the two faces of a wall. This difference in definition between plane
representations of geometry and solid representations of geometry creates some
    # ambiguities in defining where a wall starts and ends, which are further
discussed in the report. But in order to have smothened corners at connections
    # we can perform improvements at the wall geometry. Those improvements are done
mainly in 8 steps for walls following a manhattan world assumption.
    #
    # The first 4 cases deal with the connections at the starting point of the
wall, when the wall that is being updated is horizontal and is being connected
    # to another horizontal wall, when the wall that is being updated is horizontal
and is being connected to a vertical wall, when the wall that is being
    # updated is vertical and is connected to another vertical wall, when the wall
that is being updated is vertical and is connected into a horizontal wall.
```

```
    # The last 4 cases deal with updating the end point of the wall in a similar
fashion to the one described above. The main difference is that the first 4
    # cases update the actual position of the wall, with the freedom to move it in
the x and y directions to generate smooth geometry, and the last 4 cases
    # only change the lenght of the wall. That happens for two reasons: one of them
is that the end point of a wall is only defined implicitly, based off the
    # length of the wall, and the second reason is that because the start of the
wall is already aligned to the walls around it, we don't want to move it
    # and disalign that side just to align the other end. So at the end point the
algorithm looks for the best length that will generate the best alignment
    # to walls around it. In a horizontal wall that would be the optimal adition of
subtraction of the length of the wall in the x direction, the direction
    # of a horizontal wall, and for a vertical wall the challenge is finding the
best alignment in the y direction (that is however represented as the
    # internal length of the wall in the internal x direction).

    for new_wall in new_walls:
        new_wall_is_horizontal = False
        new_wall_is_vertical = False
        new_wall_has_hor_connection = False
        new_wall_has_ver_connection = False
        new_wall_start, new_wall_end = extrPoints(new_wall)
        # YDim, the thickness of the wall, is quite useful in aligning a wall being
studied to another wall orthogonal to it, as the connection should either
        # be aligned to the closest face of the wall or to the opposite face of the
orthogonal wall
        new_wall_dim_y =
new_wall.Representation.Representations[1].Items[0].SweptArea.YDim
        new_wall_is_horizontal =
new_wall.ObjectPlacement.RelativePlacement.RefDirection is None
        # if the direction ratios are not mentioned in a wall, the code would break
trying to look for them in walls that don't have them
        # so we mention them as an exception for walls that do have them
        if new_wall.ObjectPlacement.RelativePlacement.RefDirection is not None:
            new_wall_is_horizontal =
new_wall.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios == (-1.,
0., 0.)
            new_wall_is_vertical =
new_wall.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios in [(0.,
1., 0.), (0., -1., 0.)]
        for existing_wall in model.by_type('IfcWallStandardCase'):
            # first we match the new walls only to previously existing walls in the
model, as not all new walls were corrected yet. Later on
            # a similar code section will also update (or try to) new walls
relative to other possible new walls around it
            if existing_wall not in new_walls and
existing_wall.Representation.Representations[1].Items[0].SweptArea.is_a('IfcRectang
leProfileDef'):

                if
new_wall.ObjectPlacement.RelativePlacement.Location.Coordinates[2] ==
```

```
existing_wall.ObjectPlacement.RelativePlacement.Location.Coordinates[2]:  # Check
if on the same floor, possibly a small threshold here could be useful for walls
with slightly different elevations
                    existing_wall_is_vertical = False
                    existing_wall_is_horizontal = False
                    existing_wall_start, existing_wall_end =
extrPoints(existing_wall)
                    existing_wall_dim_y =
existing_wall.Representation.Representations[1].Items[0].SweptArea.YDim
                    existing_wall_is_horizontal =
existing_wall.ObjectPlacement.RelativePlacement.RefDirection is None
                    if existing_wall.ObjectPlacement.RelativePlacement.RefDirection
is not None:
                        existing_wall_is_horizontal =
existing_wall.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios == (-
1., 0., 0.)
                        existing_wall_is_vertical =
existing_wall.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios in
[(0., 1., 0.), (0., -1., 0.)]

                    # Here an interim check is done to see if horizontal walls have
horizontal connections and vertical walls have vertical connections
                    # as, if true, those connections should take preference in
determining the new wall position to keep a better geometrical alignment
                    if new_wall_is_horizontal and existing_wall_is_horizontal:
                        # Check if either the start of the new wall is close enough
to determine a connection to the start of another wall, or to the end of another
wall
                        # Here a dynamic threshold is used, so the highest value
between 0.35 and 3 times the wall thickness. For most cases 0.35 should be fine,
but if a wall
                        # is 0.7 m thick the point defined as start or end of the
neaby connected wall might be much more far away
                        if euclidean_distance(new_wall_start, existing_wall_start)
<= max(0.55, 2.5*new_wall_dim_y) or euclidean_distance(new_wall_start,
existing_wall_end) <= max(0.55, 2.5*new_wall_dim_y):
                            new_wall_has_hor_connection = True

                    elif new_wall_is_vertical and existing_wall_is_vertical:
                        if euclidean_distance(new_wall_start, existing_wall_start)
<= max(0.55, 2.5*new_wall_dim_y) or euclidean_distance(new_wall_start,
existing_wall_end) <= max(0.55, 2.5*new_wall_dim_y):
                            new_wall_has_ver_connection = True


        for existing_wall in model.by_type('IfcWallStandardCase'):
            if existing_wall not in new_walls and
existing_wall.Representation.Representations[1].Items[0].SweptArea.is_a('IfcRectang
leProfileDef'):
```

```
                if
new_wall.ObjectPlacement.RelativePlacement.Location.Coordinates[2] ==
existing_wall.ObjectPlacement.RelativePlacement.Location.Coordinates[2]:  # Check
if on the same floor
                    existing_wall_is_vertical = False
                    existing_wall_is_horizontal = False
                    existing_wall_start, existing_wall_end =
extrPoints(existing_wall)
                    existing_wall_dim_y =
existing_wall.Representation.Representations[1].Items[0].SweptArea.YDim
                    existing_wall_is_horizontal =
existing_wall.ObjectPlacement.RelativePlacement.RefDirection is None
                    if existing_wall.ObjectPlacement.RelativePlacement.RefDirection
is not None:
                        existing_wall_is_horizontal =
existing_wall.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios == (-
1., 0., 0.)
                        existing_wall_is_vertical =
existing_wall.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios in
[(0., 1., 0.), (0., -1., 0.)]
                    if new_wall_is_horizontal:
                        # A change of position in a horizontal wall gives
preference in aligning it into another horizontal wall connected to it
                        if new_wall_has_hor_connection:
                            # Update start points (Cases 1-4)
                            # check if the walls are close enough to be considered
as connected
                            if existing_wall_is_horizontal and
(euclidean_distance(new_wall_start, existing_wall_start) <= max(0.55,
2.5*new_wall_dim_y) or euclidean_distance(new_wall_start, existing_wall_end) <=
max(0.55, 2.5*new_wall_dim_y)):
                                if euclidean_distance(new_wall_start,
existing_wall_start) < euclidean_distance(new_wall_start, existing_wall_end):
                                    # if there is a gap between the two horizontal
walls, or an overlap, the start of one is aligned to the end of the other
                                    # or start to start, if there is a horizontal
wall with (-1.0, 0.0, 0.0) reference direction connected to the new wall
                                    new_wall_start = existing_wall_start
                                else:
                                    new_wall_start = existing_wall_end
                            else:
                                # if there is any horizontal wall connected to the
horizontal wall, look for it, an alignment based on vertical wall connection (case
2) is only
                                # applied when no horizontal connection exist (that
is why the else option is used, else is only parsed if the "if" condition is false)
                                continue
                        else:
                            #Case 2
                            if new_wall_is_horizontal and
existing_wall_is_vertical:
```

```
                                if euclidean_distance(new_wall_start,
existing_wall_start) <= max(0.55, 2.5*new_wall_dim_y) or
euclidean_distance(new_wall_start, existing_wall_end) <= max(0.55,
2.5*new_wall_dim_y):
                                    if euclidean_distance(new_wall_start,
existing_wall_start) < euclidean_distance(new_wall_start, existing_wall_end):
                                        # the existing_wall start or end coordinate
aligns to its longitudinal axis, so using half the thickness of the existing wall
aligns the new wall
                                        # to one of the faces of the existing wall
                                        new_wall_start = (existing_wall_start[0] -
0.5 * existing_wall_dim_y, existing_wall_start[1], new_wall_start[2])
                                    else:
                                        new_wall_start = (existing_wall_end[0] -
0.5 * existing_wall_dim_y, existing_wall_end[1], new_wall_start[2])

                    elif new_wall_is_vertical:
                        if new_wall_has_ver_connection:
                            #Case 3 - here again, give preference to vertical walls
connected to other vertical walls to keep the alignment
                            if existing_wall_is_vertical and
(euclidean_distance(new_wall_start, existing_wall_start) <= max(0.55,
2.5*new_wall_dim_y) or euclidean_distance(new_wall_start, existing_wall_end) <=
max(0.55, 2.5*new_wall_dim_y)):
                                # whichever end (end or start) of the nearby wall
is the closest to the start of the new wall, use it to correct the position and
align
                                if euclidean_distance(new_wall_start,
existing_wall_start) < euclidean_distance(new_wall_start, existing_wall_end):
                                    new_wall_start = existing_wall_start
                                else:
                                    new_wall_start = existing_wall_end
                            else:
                                continue
                        else:
                            #Case 4
                            if new_wall_is_vertical and
existing_wall_is_horizontal:
                                if euclidean_distance(new_wall_start,
existing_wall_start) <= max(0.55, 2.5*new_wall_dim_y) or
euclidean_distance(new_wall_start, existing_wall_end) <= max(0.55,
2.5*new_wall_dim_y):
                                    # if the starting points of both walls are the
closest to each other and they are connected
                                    if euclidean_distance(new_wall_start,
existing_wall_start) < euclidean_distance(new_wall_start, existing_wall_end):
                                        if
existing_wall.ObjectPlacement.RelativePlacement.RefDirection is None:
                                            # if refDirection is none the existing
wall conencting to the new wall follows the global coordinates and is at the right
side
```

```
                                            # of the vertical wall, so moving the
new wall to the right can ensure alignment and no indentation at the connection
                                            new_wall_start =
(existing_wall_start[0] + 0.5 * new_wall_dim_y, existing_wall_start[1],
new_wall_start[2])
                                        else:
                                            new_wall_start =
(existing_wall_start[0] - 0.5 * new_wall_dim_y, existing_wall_start[1],
new_wall_start[2])
                                    # start to end conenction. RefDirection None
gives the cue of whether the horizontal connected wall is at the right or left side
of the vertical wall
                                    else:
                                        if
existing_wall.ObjectPlacement.RelativePlacement.RefDirection is None:
                                            new_wall_start = (existing_wall_end[0]
- 0.5 * new_wall_dim_y,existing_wall_start[1], new_wall_start[2])
                                        else:
                                            new_wall_start = (existing_wall_end[0]
+ 0.5 * new_wall_dim_y, existing_wall_start[1], new_wall_start[2])

            # Now, update the start point of the new wall
            # new_wall.ObjectPlacement.RelativePlacement.Location.Coordinates =
new_wall_start
            new_wall.ObjectPlacement.RelativePlacement.Location.Coordinates =
(new_wall_start[0], new_wall_start[1],
new_wall.ObjectPlacement.RelativePlacement.Location.Coordinates[2])



    # Update end points (Cases 5-8)
    for new_wall in new_walls:
        new_wall_is_horizontal = False
        new_wall_is_vertical = False
        new_wall_start, new_wall_end = extrPoints(new_wall)
        new_wall_dim_y =
new_wall.Representation.Representations[1].Items[0].SweptArea.YDim
        new_wall_length =
new_wall.Representation.Representations[0].Items[0].Points[1].Coordinates[0]
        new_wall_is_horizontal =
new_wall.ObjectPlacement.RelativePlacement.RefDirection is None
        if new_wall.ObjectPlacement.RelativePlacement.RefDirection is not None:
            new_wall_is_horizontal =
new_wall.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios == (-1.,
0., 0.)
            new_wall_is_vertical =
new_wall.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios in [(0.,
1., 0.), (0., -1., 0.)]

        for existing_wall in model.by_type('IfcWallStandardCase'):
```

```
            if existing_wall not in new_walls and
existing_wall.Representation.Representations[1].Items[0].SweptArea.is_a('IfcRectang
leProfileDef'):
                if
new_wall.ObjectPlacement.RelativePlacement.Location.Coordinates[2] ==
existing_wall.ObjectPlacement.RelativePlacement.Location.Coordinates[2]:  # Check
if on the same floor
                    existing_wall_is_horizontal = False
                    existing_wall_is_vertical = False
                    existing_wall_start, existing_wall_end =
extrPoints(existing_wall)
                    existing_wall_dim_y =
existing_wall.Representation.Representations[1].Items[0].SweptArea.YDim
                    existing_wall_is_horizontal =
existing_wall.ObjectPlacement.RelativePlacement.RefDirection is None
                    if existing_wall.ObjectPlacement.RelativePlacement.RefDirection
is not None:
                        existing_wall_is_horizontal =
existing_wall.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios == (-
1., 0., 0.)
                        existing_wall_is_vertical =
existing_wall.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios in
[(0., 1., 0.), (0., -1., 0.)]

                    # here the alignment of horizontal-horizontal and vertical-
vertical is not as important as only the lenght of the wall is being changed
anyways
                    # case5
                    if new_wall_is_horizontal and existing_wall_is_horizontal:
                        # The minimum value at the dynamic threshold is higher at
the wall end because the start point was just moved, possibly
                        # making distances to another wall even greater without,
until here, a change in the wall length
                        if euclidean_distance(new_wall_end, existing_wall_start) <=
max(0.75, 2.8*new_wall_dim_y) or euclidean_distance(new_wall_end,
existing_wall_end) <= max(0.75, 2.8*new_wall_dim_y):
                            if euclidean_distance(new_wall_end,
existing_wall_start) < euclidean_distance(new_wall_end, existing_wall_end):
                                new_wall_length = existing_wall_start[0] -
new_wall_start[0]
                            else:
                                new_wall_length = existing_wall_end[0] -
new_wall_start[0]
                    # case 6
                    elif new_wall_is_horizontal and existing_wall_is_vertical:
                        if euclidean_distance(new_wall_end, existing_wall_start) <=
max(0.75, 2.8*new_wall_dim_y) or euclidean_distance(new_wall_end,
existing_wall_end) <= max(0.75, 2.8*new_wall_dim_y):
                            if euclidean_distance(new_wall_end,
existing_wall_start) < euclidean_distance(new_wall_end, existing_wall_end):
```

```
                                new_wall_length = existing_wall_start[0] -
new_wall_start[0] + 0.5 * existing_wall_dim_y
                            else:
                                new_wall_length = existing_wall_end[0] -
new_wall_start[0] + 0.5 * existing_wall_dim_y
                    # case 7
                    elif new_wall_is_vertical and existing_wall_is_vertical:
                        if euclidean_distance(new_wall_end, existing_wall_start) <=
max(0.75, 2.8*new_wall_dim_y) or euclidean_distance(new_wall_end,
existing_wall_end) <= max(0.75, 2.8*new_wall_dim_y):
                            if euclidean_distance(new_wall_end,
existing_wall_start) < euclidean_distance(new_wall_end, existing_wall_end):
                                # If the end of the new wall is conencted to the
start of another vertical wall just above it, the ideal length of the wall
                                # should be the distance between this start of the
existing wall bordering it, and the start of the new wall
                                new_wall_length = existing_wall_start[1] -
new_wall_start[1]
                            else:
                                # here the case is handled for when IFC decides to
name the point of the existing wall above our new vertical wall as an end point, so
end point
                                # is connected to end point, but the length of the
wall is the distance from this end point of the exsisting wall, ideally just
touching
                                # the end point of the new wall, to the start of
the new wall
                                new_wall_length = existing_wall_end[1] -
new_wall_start[1]
                    # case 8
                    elif new_wall_is_vertical and existing_wall_is_horizontal:
                        if euclidean_distance(new_wall_end, existing_wall_start) <=
max(0.75, 2.8*new_wall_dim_y) or euclidean_distance(new_wall_end,
existing_wall_end) <= max(0.75, 2.8*new_wall_dim_y):
                            new_wall_length = float(existing_wall_start[1] -
new_wall_start[1]) + 0.5 * existing_wall_dim_y

            # Update the new wall's length and size of the profile that defines the
wall
            new_wall.Representation.Representations[0].Items[0].Points[1].Coordinat
es = (new_wall_length, 0.0)
            new_wall.Representation.Representations[1].Items[0].SweptArea.XDim =
float(new_wall_length)
            new_wall.Representation.Representations[1].Items[0].SweptArea.Position.
Location.Coordinates = (float(new_wall_length / 2), 0.0)
```

```
    # Repurposing the code to correct connections in points where multiple new
walls connect to each other
    # Repeating the process helps solving the alingments that were not solved in
the previous step, specially
    # because now we are matching only connections of new walls to other new walls.
Previously their connection
    # to the old walls of the model was improved, and now it is improved among
other new walls, if they connect to each other
    import math

    def euclidean_distance(point1, point2):
        return math.sqrt(sum((a - b) ** 2 for a, b in zip(point1, point2)))


    for new_wall in new_walls:
        new_wall_is_horizontal = False
        new_wall_is_vertical = False
        new_wall_has_hor_connection = False
        new_wall_has_ver_connection = False
        new_wall_start, new_wall_end = extrPoints(new_wall)
        new_wall_dim_y =
new_wall.Representation.Representations[1].Items[0].SweptArea.YDim
        new_wall_is_horizontal =
new_wall.ObjectPlacement.RelativePlacement.RefDirection is None
        # if the direction ratios are not mentioned in a wall, the code would break
trying to look for them in walls that don't have them
        # so we mention them as an exception for walls that do have them
        if new_wall.ObjectPlacement.RelativePlacement.RefDirection is not None:
            new_wall_is_horizontal =
new_wall.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios == (-1.,
0., 0.)
            new_wall_is_vertical =
new_wall.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios in [(0.,
1., 0.), (0., -1., 0.)]
        for new_wall2 in new_walls:
            if new_wall2 != new_wall:

                if
new_wall.ObjectPlacement.RelativePlacement.Location.Coordinates[2] ==
new_wall2.ObjectPlacement.RelativePlacement.Location.Coordinates[2]:  # Check if on
the same floor
                    new_wall2_is_vertical = False
                    new_wall2_is_horizontal = False
                    new_wall2_start, new_wall2_end = extrPoints(new_wall2)
                    new_wall2_dim_y =
new_wall2.Representation.Representations[1].Items[0].SweptArea.YDim
                    new_wall2_is_horizontal =
new_wall2.ObjectPlacement.RelativePlacement.RefDirection is None
```

```
                    if new_wall2.ObjectPlacement.RelativePlacement.RefDirection is
not None:
                        new_wall2_is_horizontal =
new_wall2.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios == (-1.,
0., 0.)
                        new_wall2_is_vertical =
new_wall2.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios in [(0.,
1., 0.), (0., -1., 0.)]

                    # checking for horizontal-horizontal and vertical-vertical
connections
                    if new_wall_is_horizontal and new_wall2_is_horizontal:
                        if euclidean_distance(new_wall_start, new_wall2_start) <=
max(0.55, 2.5*new_wall_dim_y) or euclidean_distance(new_wall_start, new_wall2_end)
<= max(0.55, 2.5*new_wall_dim_y):
                            new_wall_has_hor_connection = True

                    elif new_wall_is_vertical and new_wall2_is_vertical:
                        if euclidean_distance(new_wall_start, new_wall2_start) <=
max(0.55, 2.5*new_wall_dim_y) or euclidean_distance(new_wall_start, new_wall2_end)
<= max(0.55, 2.5*new_wall_dim_y):
                            new_wall_has_ver_connection = True


        for new_wall2 in new_walls:
            if new_wall2 != new_wall:

                if
new_wall.ObjectPlacement.RelativePlacement.Location.Coordinates[2] ==
new_wall2.ObjectPlacement.RelativePlacement.Location.Coordinates[2]:  # Check if on
the same floor
                    new_wall2_is_vertical = False
                    new_wall2_is_horizontal = False
                    new_wall2_start, new_wall2_end = extrPoints(new_wall2)
                    new_wall2_dim_y =
new_wall2.Representation.Representations[1].Items[0].SweptArea.YDim
                    new_wall2_is_horizontal =
new_wall2.ObjectPlacement.RelativePlacement.RefDirection is None
                    if new_wall2.ObjectPlacement.RelativePlacement.RefDirection is
not None:
                        new_wall2_is_horizontal =
new_wall2.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios == (-1.,
0., 0.)
                        new_wall2_is_vertical =
new_wall2.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios in [(0.,
1., 0.), (0., -1., 0.)]
                    if new_wall_is_horizontal:
                        if new_wall_has_hor_connection:
                            # Update start points (Cases 1-4)
                            if new_wall2_is_horizontal and
(euclidean_distance(new_wall_start, new_wall2_start) <= max(0.55,
```

```
2.5*new_wall_dim_y) or euclidean_distance(new_wall_start, new_wall2_end) <=
max(0.55, 2.5*new_wall_dim_y)):
                                if euclidean_distance(new_wall_start,
new_wall2_start) < euclidean_distance(new_wall_start, new_wall2_end):
                                    new_wall_start = new_wall2_start
                                else:
                                    new_wall_start = new_wall2_end
                            else:
                                continue
                        else:
                            #Case 2
                            if new_wall_is_horizontal and new_wall2_is_vertical:
                                if euclidean_distance(new_wall_start,
new_wall2_start) <= max(0.55, 2.5*new_wall_dim_y) or
euclidean_distance(new_wall_start, new_wall2_end) <= max(0.25, 2.5*new_wall_dim_y):
                                    if euclidean_distance(new_wall_start,
new_wall2_start) < euclidean_distance(new_wall_start, new_wall2_end):
                                        new_wall_start = (new_wall2_start[0] - 0.5
* new_wall2_dim_y, new_wall2_start[1], new_wall_start[2])
                                    else:
                                        new_wall_start = (new_wall2_end[0] - 0.5 *
new_wall2_dim_y, new_wall2_end[1], new_wall_start[2])

                    elif new_wall_is_vertical:
                        if new_wall_has_ver_connection:
                            #Case 3
                            if new_wall2_is_vertical and
(euclidean_distance(new_wall_start, new_wall2_start) <= max(0.55,
2.5*new_wall_dim_y) or euclidean_distance(new_wall_start, new_wall2_end) <=
max(0.55, 2.5*new_wall_dim_y)):
                                if euclidean_distance(new_wall_start,
new_wall2_start) < euclidean_distance(new_wall_start, new_wall2_end):
                                    new_wall_start = new_wall2_start
                                else:
                                    new_wall_start = new_wall2_end
                            else:
                                continue
                        else:
                            #Case 4
                            if new_wall_is_vertical and new_wall2_is_horizontal:
                                if euclidean_distance(new_wall_start,
new_wall2_start) <= max(0.55, 2.5*new_wall_dim_y) or
euclidean_distance(new_wall_start, new_wall2_end) <= max(0.55, 2.5*new_wall_dim_y):
                                    if euclidean_distance(new_wall_start,
new_wall2_start) < euclidean_distance(new_wall_start, new_wall2_end):
                                        if
new_wall2.ObjectPlacement.RelativePlacement.RefDirection is None:

                                            new_wall_start = (new_wall2_start[0] +
0.5 * new_wall_dim_y, new_wall2_start[1], new_wall_start[2])
                                        else:
```

```
                                            new_wall_start = (new_wall2_start[0] -
0.5 * new_wall_dim_y, new_wall2_start[1], new_wall_start[2])
                                    else:
                                        if
new_wall2.ObjectPlacement.RelativePlacement.RefDirection is None:
                                            new_wall_start = (new_wall2_end[0] -
0.5 * new_wall_dim_y, new_wall2_start[1], new_wall_start[2])
                                        else:
                                            new_wall_start = (new_wall2_end[0] +
0.5 * new_wall_dim_y, new_wall2_start[1], new_wall_start[2])

            # Now, update the start point of the new wall
            # new_wall.ObjectPlacement.RelativePlacement.Location.Coordinates =
new_wall_start
            new_wall.ObjectPlacement.RelativePlacement.Location.Coordinates =
(new_wall_start[0], new_wall_start[1],
new_wall.ObjectPlacement.RelativePlacement.Location.Coordinates[2])



    # Update end points (Cases 5-8)
    for new_wall in new_walls:
        new_wall_is_horizontal = False
        new_wall_is_vertical = False
        new_wall_start, new_wall_end = extrPoints(new_wall)
        new_wall_dim_y =
new_wall.Representation.Representations[1].Items[0].SweptArea.YDim
        new_wall_length =
new_wall.Representation.Representations[0].Items[0].Points[1].Coordinates[0]
        new_wall_is_horizontal =
new_wall.ObjectPlacement.RelativePlacement.RefDirection is None
        if new_wall.ObjectPlacement.RelativePlacement.RefDirection is not None:
            new_wall_is_horizontal =
new_wall.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios == (-1.,
0., 0.)
            new_wall_is_vertical =
new_wall.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios in [(0.,
1., 0.), (0., -1., 0.)]

        for new_wall2 in model.by_type('IfcWallStandardCase'):
            if
new_wall2.Representation.Representations[1].Items[0].SweptArea.is_a('IfcRectanglePr
ofileDef'):

                if new_wall2 != new_wall:
                    if
new_wall.ObjectPlacement.RelativePlacement.Location.Coordinates[2] ==
new_wall2.ObjectPlacement.RelativePlacement.Location.Coordinates[2]:  # Check if on
the same floor
                        new_wall2_is_horizontal = False
                        new_wall2_is_vertical = False
```

```
                        new_wall2_start, new_wall2_end = extrPoints(new_wall2)
                        new_wall2_dim_y =
new_wall2.Representation.Representations[1].Items[0].SweptArea.YDim
                        new_wall2_is_horizontal =
new_wall2.ObjectPlacement.RelativePlacement.RefDirection is None
                        if new_wall2.ObjectPlacement.RelativePlacement.RefDirection
is not None:
                            new_wall2_is_horizontal =
new_wall2.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios == (-1.,
0., 0.)
                            new_wall2_is_vertical =
new_wall2.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios in [(0.,
1., 0.), (0., -1., 0.)]

                        if new_wall_is_horizontal and new_wall2_is_horizontal:
                            if euclidean_distance(new_wall_end, new_wall2_start) <=
max(0.75, 2.8*new_wall_dim_y) or euclidean_distance(new_wall_end, new_wall2_end) <=
max(0.75, 2.8*new_wall_dim_y):
                                if euclidean_distance(new_wall_end,
new_wall2_start) < euclidean_distance(new_wall_end, new_wall2_end):
                                    new_wall_length = new_wall2_start[0] -
new_wall_start[0]
                                else:
                                    new_wall_length = new_wall2_end[0] -
new_wall_start[0]

                        elif new_wall_is_horizontal and new_wall2_is_vertical:
                            if euclidean_distance(new_wall_end, new_wall2_start) <=
max(0.75, 2.8*new_wall_dim_y) or euclidean_distance(new_wall_end, new_wall2_end) <=
max(0.75, 2.8*new_wall_dim_y):
                                if euclidean_distance(new_wall_end,
new_wall2_start) < euclidean_distance(new_wall_end, new_wall2_end):
                                    new_wall_length = new_wall2_start[0] -
new_wall_start[0] + 0.5 * new_wall2_dim_y
                                else:
                                    new_wall_length = new_wall2_end[0] -
new_wall_start[0] + 0.5 * new_wall2_dim_y

                        elif new_wall_is_vertical and new_wall2_is_vertical:
                            if euclidean_distance(new_wall_end, new_wall2_start) <=
max(0.75, 2.8*new_wall_dim_y) or euclidean_distance(new_wall_end, new_wall2_end) <=
max(0.75, 2.8*new_wall_dim_y):
                                if euclidean_distance(new_wall_end,
new_wall2_start) < euclidean_distance(new_wall_end, new_wall2_end):
                                    new_wall_length = new_wall2_start[1] -
new_wall_start[1]
                                else:
                                    new_wall_length = new_wall2_end[1] -
new_wall_start[1]

                        elif new_wall_is_vertical and new_wall2_is_horizontal:
```

```
                            if euclidean_distance(new_wall_end, new_wall2_start) <=
max(0.75, 2.8*new_wall_dim_y) or euclidean_distance(new_wall_end, new_wall2_end) <=
max(0.75, 2.8*new_wall_dim_y):
                                    new_wall_length = float(new_wall2_start[1] -
new_wall_start[1]) + 0.5 *new_wall2_dim_y

                # Update the new wall's length and position
                new_wall.Representation.Representations[0].Items[0].Points[1].Coord
inates = (new_wall_length, 0.0)
                new_wall.Representation.Representations[1].Items[0].SweptArea.XDim
= float(new_wall_length)
                new_wall.Representation.Representations[1].Items[0].SweptArea.Posit
ion.Location.Coordinates = (float(new_wall_length / 2), 0.0)




    # Repurposing the code to correct connections in points where multiple new
walls connect to each other
    # the process is repeated one more time just to make sure :D
    import math

    def euclidean_distance(point1, point2):
        return math.sqrt(sum((a - b) ** 2 for a, b in zip(point1, point2)))


    for new_wall in new_walls:
        new_wall_is_horizontal = False
        new_wall_is_vertical = False
        new_wall_has_hor_connection = False
        new_wall_has_ver_connection = False
        new_wall_start, new_wall_end = extrPoints(new_wall)
        new_wall_dim_y =
new_wall.Representation.Representations[1].Items[0].SweptArea.YDim
        new_wall_is_horizontal =
new_wall.ObjectPlacement.RelativePlacement.RefDirection is None
        # if the direction ratios are not mentioned in a wall, the code would break
trying to look for them in walls that don't have them
        # so we mention them as an exception for walls that do have them
        if new_wall.ObjectPlacement.RelativePlacement.RefDirection is not None:
            new_wall_is_horizontal =
new_wall.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios == (-1.,
0., 0.)
            new_wall_is_vertical =
new_wall.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios in [(0.,
1., 0.), (0., -1., 0.)]
        for new_wall2 in new_walls:
            if new_wall2 != new_wall:

                if
new_wall.ObjectPlacement.RelativePlacement.Location.Coordinates[2] ==
```

```
new_wall2.ObjectPlacement.RelativePlacement.Location.Coordinates[2]:  # Check if on
the same floor
                    new_wall2_is_vertical = False
                    new_wall2_is_horizontal = False
                    new_wall2_start, new_wall2_end = extrPoints(new_wall2)
                    new_wall2_dim_y =
new_wall2.Representation.Representations[1].Items[0].SweptArea.YDim
                    new_wall2_is_horizontal =
new_wall2.ObjectPlacement.RelativePlacement.RefDirection is None
                    if new_wall2.ObjectPlacement.RelativePlacement.RefDirection is
not None:
                        new_wall2_is_horizontal =
new_wall2.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios == (-1.,
0., 0.)
                        new_wall2_is_vertical =
new_wall2.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios in [(0.,
1., 0.), (0., -1., 0.)]

                    # Update start points (Cases 1-4)
                    if new_wall_is_horizontal and new_wall2_is_horizontal:
                        if euclidean_distance(new_wall_start, new_wall2_start) <=
max(0.55, 2.5*new_wall_dim_y) or euclidean_distance(new_wall_start, new_wall2_end)
<= max(0.55, 2.5*new_wall_dim_y):
                            new_wall_has_hor_connection = True

                    elif new_wall_is_vertical and new_wall2_is_vertical:
                        if euclidean_distance(new_wall_start, new_wall2_start) <=
max(0.55, 2.5*new_wall_dim_y) or euclidean_distance(new_wall_start, new_wall2_end)
<= max(0.55, 2.5*new_wall_dim_y):
                            new_wall_has_ver_connection = True


        for new_wall2 in new_walls:
            if new_wall2 != new_wall:

                if
new_wall.ObjectPlacement.RelativePlacement.Location.Coordinates[2] ==
new_wall2.ObjectPlacement.RelativePlacement.Location.Coordinates[2]:  # Check if on
the same floor
                    new_wall2_is_vertical = False
                    new_wall2_is_horizontal = False
                    new_wall2_start, new_wall2_end = extrPoints(new_wall2)
                    new_wall2_dim_y =
new_wall2.Representation.Representations[1].Items[0].SweptArea.YDim
                    new_wall2_is_horizontal =
new_wall2.ObjectPlacement.RelativePlacement.RefDirection is None
                    if new_wall2.ObjectPlacement.RelativePlacement.RefDirection is
not None:
                        new_wall2_is_horizontal =
new_wall2.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios == (-1.,
0., 0.)
```

```
                        new_wall2_is_vertical =
new_wall2.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios in [(0.,
1., 0.), (0., -1., 0.)]
                    if new_wall_is_horizontal:
                        if new_wall_has_hor_connection:
                            # Update start points (Cases 1-4)
                            if new_wall2_is_horizontal and
(euclidean_distance(new_wall_start, new_wall2_start) <= max(0.55,
2.5*new_wall_dim_y) or euclidean_distance(new_wall_start, new_wall2_end) <=
max(0.55, 2.5*new_wall_dim_y)):
                                if euclidean_distance(new_wall_start,
new_wall2_start) < euclidean_distance(new_wall_start, new_wall2_end):
                                    new_wall_start = new_wall2_start
                                else:
                                    new_wall_start = new_wall2_end
                            else:
                                continue
                        else:
                            #Case 2
                            if new_wall_is_horizontal and new_wall2_is_vertical:
                                if euclidean_distance(new_wall_start,
new_wall2_start) <= max(0.55, 2.5*new_wall_dim_y) or
euclidean_distance(new_wall_start, new_wall2_end) <= max(0.55, 2.5*new_wall_dim_y):
                                    if euclidean_distance(new_wall_start,
new_wall2_start) < euclidean_distance(new_wall_start, new_wall2_end):
                                        new_wall_start = (new_wall2_start[0] - 0.5
* new_wall2_dim_y, new_wall2_start[1], new_wall_start[2])
                                    else:
                                        new_wall_start = (new_wall2_end[0] - 0.5 *
new_wall2_dim_y, new_wall2_end[1], new_wall_start[2])

                    elif new_wall_is_vertical:
                        if new_wall_has_ver_connection:
                            #Case 3
                            if new_wall2_is_vertical and
(euclidean_distance(new_wall_start, new_wall2_start) <= max(0.55,
2.5*new_wall_dim_y) or euclidean_distance(new_wall_start, new_wall2_end) <=
max(0.55, 2.5*new_wall_dim_y)):
                                if euclidean_distance(new_wall_start,
new_wall2_start) < euclidean_distance(new_wall_start, new_wall2_end):
                                    new_wall_start = new_wall2_start
                                else:
                                    new_wall_start = new_wall2_end
                            else:
                                continue
                        else:
                            #Case 4
                            if new_wall_is_vertical and new_wall2_is_horizontal:
                                if euclidean_distance(new_wall_start,
new_wall2_start) <= max(0.55, 2.5*new_wall_dim_y) or
euclidean_distance(new_wall_start, new_wall2_end) <= max(0.55, 2.5*new_wall_dim_y):
```

```
                                    if euclidean_distance(new_wall_start,
new_wall2_start) < euclidean_distance(new_wall_start, new_wall2_end):
                                        if
new_wall2.ObjectPlacement.RelativePlacement.RefDirection is None:

                                            new_wall_start = (new_wall2_start[0] +
0.5 * new_wall_dim_y, new_wall2_start[1], new_wall_start[2])
                                        else:
                                            new_wall_start = (new_wall2_start[0] -
0.5 * new_wall_dim_y, new_wall2_start[1], new_wall_start[2])
                                    else:
                                        if
new_wall2.ObjectPlacement.RelativePlacement.RefDirection is None:
                                            new_wall_start = (new_wall2_end[0] -
0.5 * new_wall_dim_y, new_wall2_start[1], new_wall_start[2])
                                        else:
                                            new_wall_start = (new_wall2_end[0] +
0.5 * new_wall_dim_y, new_wall2_start[1], new_wall_start[2])

            # Now, update the start point of the new wall
            # new_wall.ObjectPlacement.RelativePlacement.Location.Coordinates =
new_wall_start
            new_wall.ObjectPlacement.RelativePlacement.Location.Coordinates =
(new_wall_start[0], new_wall_start[1],
new_wall.ObjectPlacement.RelativePlacement.Location.Coordinates[2])



    # Update end points (Cases 5-8)
    for new_wall in new_walls:
        new_wall_is_horizontal = False
        new_wall_is_vertical = False
        new_wall_start, new_wall_end = extrPoints(new_wall)
        new_wall_dim_y =
new_wall.Representation.Representations[1].Items[0].SweptArea.YDim
        new_wall_length =
new_wall.Representation.Representations[0].Items[0].Points[1].Coordinates[0]
        new_wall_is_horizontal =
new_wall.ObjectPlacement.RelativePlacement.RefDirection is None
        if new_wall.ObjectPlacement.RelativePlacement.RefDirection is not None:
            new_wall_is_horizontal =
new_wall.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios == (-1.,
0., 0.)
            new_wall_is_vertical =
new_wall.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios in [(0.,
1., 0.), (0., -1., 0.)]

        for new_wall2 in model.by_type('IfcWallStandardCase'):
            if
new_wall2.Representation.Representations[1].Items[0].SweptArea.is_a('IfcRectanglePr
ofileDef'):
```

```
                if new_wall2 != new_wall:
                    if
new_wall.ObjectPlacement.RelativePlacement.Location.Coordinates[2] ==
new_wall2.ObjectPlacement.RelativePlacement.Location.Coordinates[2]:  # Check if on
the same floor
                        new_wall2_is_horizontal = False
                        new_wall2_is_vertical = False
                        new_wall2_start, new_wall2_end = extrPoints(new_wall2)
                        new_wall2_dim_y =
new_wall2.Representation.Representations[1].Items[0].SweptArea.YDim
                        new_wall2_is_horizontal =
new_wall2.ObjectPlacement.RelativePlacement.RefDirection is None
                        if new_wall2.ObjectPlacement.RelativePlacement.RefDirection
is not None:
                            new_wall2_is_horizontal =
new_wall2.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios == (-1.,
0., 0.)
                            new_wall2_is_vertical =
new_wall2.ObjectPlacement.RelativePlacement.RefDirection.DirectionRatios in [(0.,
1., 0.), (0., -1., 0.)]

                        if new_wall_is_horizontal and new_wall2_is_horizontal:
                            if euclidean_distance(new_wall_end, new_wall2_start) <=
max(0.75, 2.8*new_wall_dim_y) or euclidean_distance(new_wall_end, new_wall2_end) <=
max(0.75, 2.8*new_wall_dim_y):
                                if euclidean_distance(new_wall_end,
new_wall2_start) < euclidean_distance(new_wall_end, new_wall2_end):
                                    new_wall_length = new_wall2_start[0] -
new_wall_start[0]
                                else:
                                    new_wall_length = new_wall2_end[0] -
new_wall_start[0]

                        elif new_wall_is_horizontal and new_wall2_is_vertical:
                            if euclidean_distance(new_wall_end, new_wall2_start) <=
max(0.75, 2.8*new_wall_dim_y) or euclidean_distance(new_wall_end, new_wall2_end) <=
max(0.75, 2.8*new_wall_dim_y):
                                if euclidean_distance(new_wall_end,
new_wall2_start) < euclidean_distance(new_wall_end, new_wall2_end):
                                    new_wall_length = new_wall2_start[0] -
new_wall_start[0] + 0.5 * new_wall2_dim_y
                                else:
                                    new_wall_length = new_wall2_end[0] -
new_wall_start[0] + 0.5 * new_wall2_dim_y

                        elif new_wall_is_vertical and new_wall2_is_vertical:
                            if euclidean_distance(new_wall_end, new_wall2_start) <=
max(0.75, 2.8*new_wall_dim_y) or euclidean_distance(new_wall_end, new_wall2_end) <=
max(0.75, 2.8*new_wall_dim_y):
```

```
                                if euclidean_distance(new_wall_end,
new_wall2_start) < euclidean_distance(new_wall_end, new_wall2_end):
                                    new_wall_length = new_wall2_start[1] -
new_wall_start[1]
                                else:
                                    new_wall_length = new_wall2_end[1] -
new_wall_start[1]

                        elif new_wall_is_vertical and new_wall2_is_horizontal:
                            if euclidean_distance(new_wall_end, new_wall2_start) <=
max(0.75, 2.8*new_wall_dim_y) or euclidean_distance(new_wall_end, new_wall2_end) <=
max(0.75, 2.8*new_wall_dim_y):
                                new_wall_length = float(new_wall2_start[1] -
new_wall_start[1]) + 0.5 *new_wall2_dim_y

                # Update the new wall's length and position
                new_wall.Representation.Representations[0].Items[0].Points[1].Coord
inates = (new_wall_length, 0.0)
                new_wall.Representation.Representations[1].Items[0].SweptArea.XDim
= float(new_wall_length)
                new_wall.Representation.Representations[1].Items[0].SweptArea.Posit
ion.Location.Coordinates = (float(new_wall_length / 2), 0.0)




    #named as newWall to avoid a possible confusion with new_wall worked on above
    newWalls_matched_to_eachother = []
    for newWall in new_walls:

        # Last step: add connections to other walls into the new wall
                    # Iterate over matched IFC walls
        for ifc_wall in model.by_type("IfcWallStandardCase"):
            if
ifc_wall.Representation.Representations[1].Items[0].SweptArea.is_a('IfcRectanglePro
fileDef'):

                if ifc_wall != newWall:

                    # Extract the start and end points of the new IFC wall
                    new_wall_points = extrPoints(newWall)
                    new_wall_base = new_wall_points[0]
                    new_wall_end = new_wall_points[1]

                    # Extract the start and end points of the previously existing
IFC wall
                    ifc_wall_points = extrPoints(ifc_wall)
                    ifc_wall_base = ifc_wall_points[0]
                    ifc_wall_end = ifc_wall_points[1]
```

```
                    # Calculate the smallest distances between the base and end
points of the new wall and the base and end points of the previously existing IFC
wall
                    distance_base = min(math.dist(new_wall_base, ifc_wall_base),
math.dist(new_wall_base, ifc_wall_end))
                    distance_end = min(math.dist(new_wall_end, ifc_wall_base),
math.dist(new_wall_end, ifc_wall_end))

                    # An existing wall may either be connected to the beginning of
the new wall, to its end, connect to it
                    # along its path (.ATPATH. connection), or unconnected. At path
connections are not dealt with in this
                    # methodology however, because the comparison of walls is based
on point cloud walls and ifc walls having
                    # continuous planes on both sides

                    # If the smallest distance is less than 0.55m, create a new
IfcRelConnectsPathElements relationship
                    # max(0.55, 3.0*ifc_w_dimy)
                    ifc_w_dimy =
ifc_wall.Representation.Representations[1].Items[0].SweptArea.YDim
                    walls_already_connected = False
                    if distance_base < max(0.55, 2.2*ifc_w_dimy):
                        for i in newWalls_matched_to_eachother:
                            if newWall in i and ifc_wall in i:
                                walls_already_connected = True
                        if walls_already_connected is False:
                            newWalls_matched_to_eachother.append([newWall,
ifc_wall])
                            # if starting points are connected
                            if math.dist(new_wall_base, ifc_wall_base) < max(0.55,
2.2*ifc_w_dimy):
                                rel_connects_path_elements =
model.create_entity('IfcRelConnectsPathElements')
                                rel_connects_path_elements.RelatingElement =
ifc_wall
                                rel_connects_path_elements.RelatedElement = newWall
                                rel_connects_path_elements.RelatedConnectionType =
'ATSTART'
                                rel_connects_path_elements.GlobalId =
ifcopenshell.guid.compress(uuid.uuid1().hex)
                                rel_connects_path_elements.Name =
str(f'{newWall.GlobalId} + | + {ifc_wall.GlobalId}')
                                rel_connects_path_elements.OwnerHistory =
owner_history
                                rel_connects_path_elements.RelatingConnectionType =
'ATSTART'
                                rel_connects_path_elements.Description =
'Structural'
```

```
                            #if the starting point of the new wall is connected to
the end of the other wall
                            elif math.dist(new_wall_base, ifc_wall_end) < max(0.55,
3.0*ifc_w_dimy):
                                rel_connects_path_elements =
model.create_entity('IfcRelConnectsPathElements')
                                rel_connects_path_elements.RelatingElement =
ifc_wall
                                rel_connects_path_elements.RelatedElement = newWall
                                rel_connects_path_elements.RelatedConnectionType =
'ATSTART'
                                rel_connects_path_elements.GlobalId =
ifcopenshell.guid.compress(uuid.uuid1().hex)
                                rel_connects_path_elements.Name =
str(f'{newWall.GlobalId} + | + {ifc_wall.GlobalId}')
                                rel_connects_path_elements.OwnerHistory =
owner_history
                                rel_connects_path_elements.RelatingConnectionType =
'ATEND'
                                rel_connects_path_elements.Description =
'Structural'


                    if distance_end < max(0.55, 2.2*ifc_w_dimy):
                        for i in newWalls_matched_to_eachother:
                            if newWall in i and ifc_wall in i:
                                walls_already_connected = True
                        if walls_already_connected is False:
                            newWalls_matched_to_eachother.append([newWall,
ifc_wall])
                            # if the end of the new wall is connected to the start
of the other wall
                            if math.dist(new_wall_end, ifc_wall_base) < max(0.55,
2.2*ifc_w_dimy):
                                rel_connects_path_elements =
model.create_entity('IfcRelConnectsPathElements')
                                rel_connects_path_elements.RelatingElement =
ifc_wall
                                rel_connects_path_elements.RelatedElement = newWall
                                rel_connects_path_elements.RelatedConnectionType =
'ATEND'
                                rel_connects_path_elements.GlobalId =
ifcopenshell.guid.compress(uuid.uuid1().hex)
                                rel_connects_path_elements.Name =
str(f'{newWall.GlobalId} + | + {ifc_wall.GlobalId}')
                                rel_connects_path_elements.OwnerHistory =
owner_history
                                rel_connects_path_elements.RelatingConnectionType =
'ATSTART'
                                rel_connects_path_elements.Description =
'Structural'
```

```
                            # if the end of the new wall is connected to the end of
the other wall
                            elif math.dist(new_wall_end, ifc_wall_end) < max(0.55,
2.2*ifc_w_dimy):
                                rel_connects_path_elements =
model.create_entity('IfcRelConnectsPathElements')
                                rel_connects_path_elements.RelatingElement =
ifc_wall
                                rel_connects_path_elements.RelatedElement = newWall
                                rel_connects_path_elements.RelatedConnectionType =
'ATEND'
                                rel_connects_path_elements.GlobalId =
ifcopenshell.guid.compress(uuid.uuid1().hex)
                                rel_connects_path_elements.Name =
str(f'{newWall.GlobalId} + | + {ifc_wall.GlobalId}')
                                rel_connects_path_elements.OwnerHistory =
owner_history
                                rel_connects_path_elements.RelatingConnectionType =
'ATEND'
                                rel_connects_path_elements.Description =
'Structural'


        else:
            print("No existing wall found.")

    from datetime import datetime
    import ifcopenshell
    current_datetime = datetime.now()

    # Format the date and time as a string in the specified format
    formatted_datetime = current_datetime.strftime("%d%m%y_%H%M")

    # Construct the new filename by appending the formatted date and time
    # If you want to use a specific name or modify it, you can do so here
    new_filename = f"modified_ifc_file_{formatted_datetime}.ifc"

    # Write the modified IFC file with the new filename
    model.write(new_filename)

    # Return the new filename
    return new_filename
```

Appendix VII

ceilingUpdaTor.py checks and updates ceilings

```
# This code is part of the Master Thesis of Jean van der Meer presented to the
Eindhoven University of Technology


import numpy as np
import pandas as pd
import os
import ifcopenshell

# Function to process segmented ceilings from point cloud data and extract the
necessary geometry data
def process_seg_ceilings(files2):
    # the input of the function are the several point cloud files, each one with
one segmented ceiling
    data_dict = {}
    ceiling_dict = {}

    for file in files2:
        data_dict[file] = pd.read_csv(file, sep=' ', header=None, names=['X', 'Y',
'Z', 'R', 'G', 'B'])

    for file, data in data_dict.items():
        # each line is divided into x, y and z values, so each one is stored in
this initial data_dict for each ceiling
        x = data['X'].values
        y = data['Y'].values
        z = data['Z'].values

        z_avg = np.mean(z)  # Compute the average Z value (height), used to find
the cg and other geometry information

        # Compute center of gravity of X and Y coordinates
        x_cg = np.mean(x)
        y_cg = np.mean(y)
        cg = (x_cg, y_cg, z_avg)

        # Compute 8 additional points around the center of gravity, they plus the
cg are used to check if the central area of a
        # point cloud ceiling can be matched to the area of one of the ifc
ceilings, if a majority of those points are within the area of an ifc ceiling
        # The aditional points are basically a square around the cg, or an 8-
pointed star inscribed in a square
        offset = 0.6
        additional_points = [
            (x_cg + offset, y_cg, z_avg),
            (x_cg - offset, y_cg, z_avg),
            (x_cg, y_cg + offset, z_avg),
            (x_cg, y_cg - offset, z_avg),
```

```
            (x_cg + offset, y_cg + offset, z_avg),
            (x_cg - offset, y_cg + offset, z_avg),
            (x_cg - offset, y_cg - offset, z_avg),
            (x_cg + offset, y_cg - offset, z_avg)
        ]

        nine_points = [cg] + additional_points

        # Store the data in ceiling_dict and change the name of each ceiling in the
dictionary from e.g. ceiling1.txt, ceiling2.txt into just celing1, ceiling2
        ceiling_name = os.path.splitext(os.path.basename(file))[0]
        ceiling_dict[ceiling_name] = {
            'z_avg': z_avg,
            'nine_points': nine_points
        }

    return ceiling_dict

# Function to check and update ceilings in the IFC model
def check_and_update_ceilings(model, pc_ceilings):
    for pc_name, pc_data in pc_ceilings.items():
        z_avg = pc_data['z_avg']
        nine_points = pc_data['nine_points']

        # Ceilings usually have their geometry represented either as a rectangle,
for perfectly rectangular rooms, or as an IfcArbitraryClosedProfileDef, for
        # rooms with a more complex layout. A rectangular room is represented by a
rectangle oriented around a center point, that might also have local coordinates
        # that need to be translated into the coordinates of the rest of the model
by a reference direction. Usually IFC (or IFC authoring tools) considers the
        # larger side of an element as its internal main axis, the x axis, but if
in the global coordinates the element has its shortest dimension on the x axis,
        # it will probably have a reference direction of either (0.0, 1.0, 0.0) or
(0.0, -1.0, 0.0) to bring its "wide local x axis" into the global y axis.
        # Ceilings represented by an IfcArbitraryClosedProfileDef can also have
reference axis and follow a similar idea. Ceilings represented by a
        # IfcArbitraryClosedProfileDef have a location given in global coordinates
(except for the z coordinate that is relative to the floor), and then they
        # have a polyline that represents the outline of the ceiling in local
coordinates, that is, each point has x,y coordinates that represent its distance
        # in the internal x and y axis to the location point of the ceiling. Those
coordinates might have to be transformed, with internal
        # x and y values being added or removed from the location coordinate of the
ceiling, to then find the global coordinates of the polyline points, as
        # will be shown later in the code
        for ceiling in model.by_type('IfcCovering'):
            if
ceiling.Representation.Representations[0].Items[0].SweptArea.is_a('IfcRectangleProf
ileDef'):
                # Handle IfcRectangleProfileDef, here x_dim and y_dim are the
internal coordinate dimensions of the rectangle profile
```

```
                x_dim =
ceiling.Representation.Representations[0].Items[0].SweptArea.XDim
                y_dim =
ceiling.Representation.Representations[0].Items[0].SweptArea.YDim
                # here are the coordinates of the point that locates semi-globally
(except for z) the ceiling
                base_x =
ceiling.Representation.Representations[0].Items[0].Position.Location.Coordinates[0]
                base_y =
ceiling.Representation.Representations[0].Items[0].Position.Location.Coordinates[1]
                # here is the height of the floor the ceiling is located, the
global height of the ceiling is that of the floor (floor_z) plus ceiling_z
                floor_z =
ceiling.ObjectPlacement.PlacementRelTo.RelativePlacement.Location.Coordinates[2]
                ceiling_z =
ceiling.Representation.Representations[0].Items[0].Position.Location.Coordinates[2]
                ref_direction =
ceiling.Representation.Representations[0].Items[0].Position.RefDirection

                if ref_direction: #ceilings aligned the global coordinates, usually
ceilings whose larger dimension is in the x axis, have no specific RefDirection
                    direction_ratios = ref_direction.DirectionRatios
                    if direction_ratios == (0.0, 1.0, 0.0) or direction_ratios ==
(0.0, -1.0, 0.0):
                        # as explained previously, if the internal coordinates are
orthogonal to the global coordinates we invert x and y dimensions
                        x_dim, y_dim = y_dim, x_dim

                # Compute the bounds of the ifc ceiling volume
                x_min = base_x - x_dim / 2
                x_max = base_x + x_dim / 2
                y_min = base_y - y_dim / 2
                y_max = base_y + y_dim / 2
                z_min = floor_z + ceiling_z - 0.5 # a threshold is used (here 0.5m)
to look for a match with ceilings globally 0.5 meters above or 0.5 meters below the
ifc ceiling
                z_max = floor_z + ceiling_z + 0.5

                # Check if at least 6 of the 9 points are within the volume
                match_count = sum(1 for point in nine_points if x_min <= point[0]
<= x_max and y_min <= point[1] <= y_max and z_min <= point[2] <= z_max)

                if match_count >= 6:
                    new_ceiling_z = z_avg - floor_z # setting the height of the IFC
ceiling to that of the matched point cloud ceiling, and converting the global
height to local height
                    ceiling.Representation.Representations[0].Items[0].Position.Loc
ation.Coordinates =
(float(ceiling.Representation.Representations[0].Items[0].Position.Location.Coordin
ates[0]),
```

```
float(ceiling.Representation.Representations[0].Items[0].Position.Location.Coordina
tes[1]), float(new_ceiling_z))
                    #
ceiling.Representation.Representations[0].Items[0].Position.Location.Coordinates[2]
= new_ceiling_z' -> this does not work, an entire tuple of coordinates needs to be
assigned
                    print('rectangular celiling updated')


            # now is the script for when a ceiling is defined by a polyline
            elif
ceiling.Representation.Representations[0].Items[0].SweptArea.is_a('IfcArbitraryClos
edProfileDef'):

                floor_z =
ceiling.ObjectPlacement.PlacementRelTo.RelativePlacement.Location.Coordinates[2]
                ceiling_z =
ceiling.Representation.Representations[0].Items[0].Position.Location.Coordinates[2]
                z_min = floor_z + ceiling_z - 0.5
                z_max = floor_z + ceiling_z + 0.5

                base_point =
ceiling.Representation.Representations[0].Items[0].Position.Location.Coordinates
                ref_direction =
ceiling.Representation.Representations[0].Items[0].Position.RefDirection

                # here is where the points that outline the polyline are given
                points =
ceiling.Representation.Representations[0].Items[0].SweptArea.OuterCurve.Points
                polygon_points = []

                # here the points given in internal coordinates are translated to
global coordinates if the reference direction is orthogonal to the global axes
                if ref_direction is None or ref_direction.DirectionRatios != (0.0,
1.0, 0.0):
                    # No RefDirection or different direction ratios; use base point
directly, x is added to x, y to y
                    polygon_points = [(base_point[0] + p.Coordinates[0],
base_point[1] + p.Coordinates[1]) for p in points]
                else:
                    # RefDirection with DirectionRatios (0.0, 1.0, 0.0); transform
coordinates, because of the rotarion direction from (1.0, 0.0, 0.0) to (0.0, 1.0,
0.0)
                    # the y local distances are reduced from the x global
distances, and the x local distances are added to the y global distances
                    polygon_points = [(base_point[0] - p.Coordinates[1],
base_point[1] + p.Coordinates[0]) for p in points]

                from shapely.geometry import Polygon, Point
                # a 3d polygon is made to see if the 9 points extracted from a
point cloud ceiling fit within the projection of the IFC ceiling
```

```
                polygon = Polygon(polygon_points)

                # Check if at least 6 of the 9 points are within the volume
                match_count = sum(1 for point in nine_points if
polygon.contains(Point(point[:2])) and z_min <= point[2] <= z_max)

                if match_count >= 6:
                    new_ceiling_z = z_avg - floor_z

                    ceiling.Representation.Representations[0].Items[0].Position.Loc
ation.Coordinates =
(float(ceiling.Representation.Representations[0].Items[0].Position.Location.Coordin
ates[0]),
float(ceiling.Representation.Representations[0].Items[0].Position.Location.Coordina
tes[1]), float(new_ceiling_z))

                    print('polygonal ceiling updated')
    # Save the modified IFC file with the date and time of the changes
    from datetime import datetime

    current_datetime = datetime.now()
    formatted_datetime = current_datetime.strftime("%d%m%y_%H%M")
    new_filename = f"modified_ifc_file_{formatted_datetime}.ifc"
    model.write(new_filename)

    return new_filename
```

Appendix VIII

columnUpdaTor.py checks and updates columns

```
# This code is part of the Master Thesis of Jean van der Meer presented to the
Eindhoven University of Technology
import ifcopenshell
import numpy as np
import pandas as pd
import os

def process_seg_columns(files2):
    # the segmented point clouds of columns are loaded and geometric information is
extracted from them, assuming a manhattan world scenario with orthogonal planes
    column_dict = {}

    for file in files2:
        # Load the point cloud data
        data = pd.read_csv(file, sep=' ', header=None, names=['X', 'Y', 'Z', 'R',
'G', 'B'])

        # Extract X, Y, Z coordinates
        x = data['X'].values
        y = data['Y'].values
        z = data['Z'].values

        # Compute min and max values for X, Y, and Z
        x_min = np.min(x)
        x_max = np.max(x)
        y_min = np.min(y)
        y_max = np.max(y)
        z_min = np.min(z)
        z_max = np.max(z)

        # Define the four points, the four vertices of a square column, e.g. xlyh:
x lowest point, y highest point, xhyh: x highest point, y highest point etc
        xlyh = (x_min, y_max, z_min)
        xhyh = (x_max, y_max, z_min)
        xlyl = (x_min, y_min, z_min)
        xhyl = (x_max, y_min, z_min)

        # Calculate center of gravity (cg)
        cg_x = (x_max + x_min) / 2
        cg_y = (y_max + y_min) / 2
        cg = (cg_x, cg_y, z_min) #the base point is considered as the base of the
column from which it is "extruded", thus z_min

        # Calculate column height
        column_height = z_max - z_min

        # Calculate profile base and height
```

```
        # However the code is not developed here, the profile base and profile
height can be easily used to compare the profiles of columns being checked,
        # and if the column in IFC has dimensions closer to a different profile
found at the model, the column type can be updated, by finding which
        # column type has the closest profile to that of the point cloud column
        profile_base = x_max - x_min
        profile_height = y_max - y_min

        # Snippet to remove the file extension
        column_name = os.path.splitext(os.path.basename(file))[0]

        # Store the data in the column dictionary
        column_dict[column_name] = {
            'xlyh': xlyh,
            'xhyh': xhyh,
            'xlyl': xlyl,
            'xhyl': xhyl,
            'cg': cg,
            'column_height': column_height,
            'profile_base': profile_base,
            'profile_height': profile_height
        }

    return column_dict



import ifcopenshell
from datetime import datetime
# This function also used in several operations, with walls, finds the global start
and end point of a wall in a standardized manner.
# It is used here to help locate the space a wall occupies and areas very close to
it, to identify columns that could be hidden inside walls
def extrPoints(wall):
    local_placement = wall.ObjectPlacement
    if local_placement:
        location = local_placement.RelativePlacement.Location.Coordinates

        # this segment is not relevant here but works regardless, sometimes the z
can be extracted from location[2] when a new wall is still being generated from
point cloud data
        if wall.ContainedInStructure[0].RelatingStructure.Elevation:
            z_coord = wall.ContainedInStructure[0].RelatingStructure.Elevation
        else:
            z_coord = location[2]

        # Get the length of the wall, to help to find its end point that is
normally only implicit in IFC
        lengthW =
wall.Representation.Representations[0].Items[0].Points[1].Coordinates[0]
```

```
        # Determine the end coordinate based on direction
        if local_placement.RelativePlacement.RefDirection is not None:
            axis2 = local_placement.RelativePlacement.RefDirection.DirectionRatios
            axis1 = local_placement.RelativePlacement.Axis.DirectionRatios

            if axis1 == (0, 0, 1) and axis2 == (-1, 0, 0):
                end_coordinate = ((location[0] - lengthW), location[1],
location[2])
            elif axis1 == (0, 0, 1) and axis2 == (0, 1, 0):
                end_coordinate = (location[0], (location[1] + lengthW),
location[2])
            elif axis1 == (0, 0, 1) and axis2 == (0, -1, 0):
                end_coordinate = (location[0], (location[1] - lengthW),
location[2])
        else:
            end_coordinate = (location[0] + lengthW, location[1], location[2])

        return (location[0], location[1], z_coord), (end_coordinate[0],
end_coordinate[1], z_coord)

    return None

def check_and_update_columns(model, pc_columns):
    import ifcopenshell.api
    ifc_columns_close_to_walls = []
    ifc_columns_not_close_to_walls = []
    # here "emb" means embedded, do columns that are found inside a wall in the ifc
model or in the point cloud data, whose detection might be hindered
    ifc_emb_columns_no_match = []
    matched_emb_pc_columns = []

    walls = model.by_type('IfcWallStandardCase')
    columns = model.by_type('IfcColumn')

    for column in columns:
        # Some columns depending on how they are modeled have their location at
column.Representation.Representations[0].Items[0].MappingSource.MappedRepresentatio
n.Items[0].Position.Location.Coordinates
        # instead of at
column.ObjectPlacement.RelativePlacement.Location.Coordinates, where it usually is,
so we need to check for both cases
        # When the column's location is at
column.ObjectPlacement.RelativePlacement.Location.Coordinates, the other location,
column.Representation.Representations[0].Items[0].MappingSource.MappedRepresentatio
n.Items[0].Position.Location.Coordinates
        # is always set as (0.0, 0.0, 0.0), so we can use that as a test to know
where the location is stored. The other scenario, where the location is stored at
        # the mapping source usually happens when every column in the file, or many
of them, have their own column type and all geometric infornmation is then
        # stored at the column type instead of the column instance, even when many
columns have the same profile. This can happen depending on how columns are
```

```
        # modeled in a BIM authoring tool
        if
column.Representation.Representations[0].Items[0].MappingSource.MappedRepresentatio
n.Items[0].Position.Location.Coordinates == (0.0, 0.0, 0.0):
            column_position =
column.ObjectPlacement.RelativePlacement.Location.Coordinates
            column_x, column_y, column_z = column_position
        else:
            column_position =
column.Representation.Representations[0].Items[0].MappingSource.MappedRepresentatio
n.Items[0].Position.Location.Coordinates
            column_x, column_y, column_z = column_position

        column_is_close_to_any_wall = False  # Flag to track if the column is close
to any wall, all ifc columns are checked for all point cloud columns, and if they
are once close to a wall they are labeled as so, to make sure there are no
duplicates

        for wall in walls:
            start_point, end_point = extrPoints(wall)
            if not start_point or not end_point:
                continue

            # Determine if the wall is horizontal or vertical
            local_placement = wall.ObjectPlacement
            if local_placement.RelativePlacement.RefDirection is None or
(local_placement.RelativePlacement.Axis.DirectionRatios == (0, 0, 1) and
local_placement.RelativePlacement.RefDirection.DirectionRatios == (-1, 0, 0)):
                is_horizontal = True
            elif (local_placement.RelativePlacement.Axis.DirectionRatios == (0, 0,
1) and local_placement.RelativePlacement.RefDirection.DirectionRatios in [(0, 1,
0), (0, -1, 0)]):
                is_vertical = True
            else:
                is_horizontal = False
                is_vertical = False

            if is_horizontal:
                # Check if the column is within the range of a horizontal wall
                y_min, y_max = sorted([start_point[1], end_point[1]])  # y min and
y max are the same y coordinate in a horizontal wall
                x_min, x_max = sorted([start_point[0], end_point[0]])

                # some thresholds are used for the y value (that remains the same
in vertical walls) and the x values that the center of a column can be to be
considered embedded in that wall
                # some tolerance is also given for the z value of the based of the
wall and column
                if y_min - 0.35 <= column_y <= y_max + 0.35 and x_min - 0.35 <=
column_x <= x_max + 0.35 and abs(column_z - start_point[2]) <= 0.25:
                    column_is_close_to_any_wall = True
```

```
                    break  # No need to check further walls if a match is found

            elif is_vertical:
                # Check if the column is within the range of a vertical wall
                x_min, x_max = sorted([start_point[0], end_point[0]]) # x min and x
max are the same y coordinate in a horizontal wall
                y_min, y_max = sorted([start_point[1], end_point[1]])

                # some thresholds are used for the x value (that remains the same
in horizontal walls) and the y values that the center of a column can be to be
considered embedded in that wall
                # some tolerance is also given for the z value of the based of the
wall and column
                if x_min - 0.35 <= column_x <= x_max + 0.35 and y_min - 0.35 <=
column_y <= y_max + 0.35 and abs(column_z - start_point[2]) <= 0.25:
                    column_is_close_to_any_wall = True
                    break  # No need to check further walls if a match is found

        if column_is_close_to_any_wall:
            ifc_columns_close_to_walls.append(column)
        else:
            ifc_columns_not_close_to_walls.append(column)

    # Matching (or try to) IFC columns close to walls with point cloud columns
    # create a copy of point cloud columns to then remove all columns that are
close to a wall and matched to an embedded IFC column, to know how many point cloud
columns are left
    remaining_pc_columns = pc_columns.copy()

    for ifc_column in ifc_columns_close_to_walls:
        if
ifc_column.Representation.Representations[0].Items[0].MappingSource.MappedRepresent
ation.Items[0].Position.Location.Coordinates == (0.0, 0.0, 0.0):
            column_position =
ifc_column.ObjectPlacement.RelativePlacement.Location.Coordinates
            column_x, column_y, column_z = column_position
        else:
            column_position =
ifc_column.Representation.Representations[0].Items[0].MappingSource.MappedRepresent
ation.Items[0].Position.Location.Coordinates
            column_x, column_y, column_z = column_position
        # find the elevation of the floor an IFC column is located, to help
comparing the local z value of the IFC column to the global z value of the point
cloud column
        elevation = ifc_column.ContainedInStructure[0].RelatingStructure.Elevation

        matched = False
        for pc_column_name, pc_column_data in list(remaining_pc_columns.items()):
            cg_x, cg_y, cg_z = pc_column_data['cg']
```

```
            # the elevation of the floor an IFC column is in is reduced from the
elevation of the point cloud column it is being compared to, to see if their
position is comparable
            cg_z_transformed = cg_z - elevation

            distance = np.sqrt((column_x - cg_x)**2 + (column_y - cg_y)**2 +
(column_z - cg_z_transformed)**2)
            # Threshold for embedded ifc-pcd column matching
            if distance <= 1.1:
                matched_emb_pc_columns.append(pc_column_name)
                remaining_pc_columns.pop(pc_column_name)
                matched = True
                print(f'Column {pc_column_name} got matched to an IFC column
embedded in a wall!')
                break

        if not matched:
            ifc_emb_columns_no_match.append(ifc_column.GlobalId)

    num_ifc_emb_columns_no_match = len(ifc_emb_columns_no_match)
    if num_ifc_emb_columns_no_match > 0:
        print(f'There are {num_ifc_emb_columns_no_match} IFC columns embedded in
walls that couldn not be checked against point cloud data.')

    num_pc_columns_remaining = len(remaining_pc_columns) # point cloud columns that
are not close to a wall, or at least not matched to ifc columns embedded in walls
    num_ifc_columns_not_close = len(ifc_columns_not_close_to_walls)
    message2 = ''
    if num_pc_columns_remaining < num_ifc_columns_not_close:
        print(f'There were {num_pc_columns_remaining} columns found in the point
cloud data away from walls, while the IFC as-designed file had more columns
({num_ifc_columns_not_close} IFC columns not embedded in walls).')
        message2 = f'There were {num_pc_columns_remaining} columns found in the
point cloud data away from walls, while the IFC as-designed file had more columns
({num_ifc_columns_not_close} IFC columns not embedded in walls). This means the
designed project had more columns not embedded in walls, so you are advised to
check the point cloud and see if the threshold should be adapted or a manual
intervention is needed'




    # Last step: Match remaining point cloud columns with IFC columns not close to
walls
    matched_ifc_columns = []
    unmatched_ifc_columns = ifc_columns_not_close_to_walls.copy()

    for pc_column_name, pc_column_data in list(remaining_pc_columns.items()):
        cg_x, cg_y, cg_z = pc_column_data['cg']
        matched = False
```

```
        for ifc_column in ifc_columns_not_close_to_walls:
            # here again, both positions where the location of a column might be,
depending on whether it is defined by the column type/mappingSource or
            # column instance, have to be handled accordingly
            if
ifc_column.Representation.Representations[0].Items[0].MappingSource.MappedRepresent
ation.Items[0].Position.Location.Coordinates == (0.0, 0.0, 0.0):
                column_position =
ifc_column.ObjectPlacement.RelativePlacement.Location.Coordinates
                column_x, column_y, column_z = column_position
            else:
                column_position =
ifc_column.Representation.Representations[0].Items[0].MappingSource.MappedRepresent
ation.Items[0].Position.Location.Coordinates
                column_x, column_y, column_z = column_position
            elevation =
ifc_column.ContainedInStructure[0].RelatingStructure.Elevation

            cg_z_transformed = cg_z - elevation

            distance = np.sqrt((column_x - cg_x)**2 + (column_y - cg_y)**2 +
(column_z - cg_z_transformed)**2)

            ######## THRESHOLD: ########
            ############################
            if distance <= 1.2:
                # Update matched IFC column position with point cloud data
                # Here again, depending on where that column was keeping its
location we need to add the new location to that place too, as other data
pertaining
                # the geometrical representation of the column are also in that
"location" (either the ObjectPlacement or the MappingSource via a column type).
                # Furthermore, because new columns that need to be created are
copied based on existing column types at the model and have their new location
                # assigned afterwards, the assignment of that location also needs
to respect how their reference column structured its geometry
                if
ifc_column.Representation.Representations[0].Items[0].MappingSource.MappedRepresent
ation.Items[0].Position.Location.Coordinates != (0.0, 0.0, 0.0):
                    ifc_column.Representation.Representations[0].Items[0].MappingSo
urce.MappedRepresentation.Items[0].Position.Location.Coordinates = (float(cg_x),
float(cg_y), float(column_z))
                else:
                    ifc_column.ObjectPlacement.RelativePlacement.Location.Coordinat
es = (float(cg_x), float(cg_y), float(column_z))
                matched_ifc_columns.append(ifc_column)
                unmatched_ifc_columns.remove(ifc_column)
                remaining_pc_columns.pop(pc_column_name)
                matched = True
                print(f'Column {pc_column_name} got matched to an IFC column not
embedded in a wall!')
```

```
                break

        if not matched:
            # If no match, create a new column in IFC, as mentioned above, the new
column will receive its locating point following the schema of the model column it
copies semantics from,
            # i.e., either ObjectPlacement or MappingSource of the representation.
The elevation of the floor is reduced from the global z value of the point cloud,
to give the new column a
            # proper local position relative to the floor it is in
            possible_columns = [col for col in ifc_columns_not_close_to_walls if
abs(col.ContainedInStructure[0].RelatingStructure.Elevation - cg_z) <= 0.4]
            if possible_columns:
                existing_column = possible_columns[0]
                new_column = ifcopenshell.util.element.copy_deep(model,
existing_column, exclude=None)
                if
new_column.Representation.Representations[0].Items[0].MappingSource.MappedRepresent
ation.Items[0].Position.Location.Coordinates != (0.0, 0.0, 0.0):
                    new_column.Representation.Representations[0].Items[0].MappingSo
urce.MappedRepresentation.Items[0].Position.Location.Coordinates = (float(cg_x),
float(cg_y), float(cg_z -
existing_column.ContainedInStructure[0].RelatingStructure.Elevation))
                else:
                    new_column.ObjectPlacement.RelativePlacement.Location.Coordinat
es = (float(cg_x), float(cg_y), float(cg_z -
existing_column.ContainedInStructure[0].RelatingStructure.Elevation))
                print(f'New IFC column created for unmatched point cloud column
{pc_column_name}.')
    num_unmatched_free_columns = len(unmatched_ifc_columns)
    # Remove unmatched IFC columns
    for column_not_matched in unmatched_ifc_columns:
        colGuid = column_not_matched.GlobalId
        ifcopenshell.api.run("root.remove_product", model,
product=column_not_matched)
        print(f'IFC column {colGuid} removed as it was not matched to any point
cloud column.')
    # Save the modified IFC file
    current_datetime = datetime.now()
    formatted_datetime = current_datetime.strftime("%d%m%y_%H%M")
    new_filename = f"modified_ifc_file_{formatted_datetime}.ifc"
    model.write(new_filename)

    return {
        'new_filename': new_filename,
        'ifc_emb_columns_no_match': ifc_emb_columns_no_match,
        'num_ifc_emb_columns_no_match': num_ifc_emb_columns_no_match,
        'message' : message2,
        'num_unmatched_free_ifc_columns': num_unmatched_free_columns
    }
```

Appendix IX

pcdSimplifier2.py removes unnecessary data from point clouds making them use less space

```
# This code is part of the Master Thesis of Jean van der Meer presented to the
Eindhoven University of Technology
# simple file to update one or several point cloud files at once, some of them
might have to many fields of data that either
# make the files too heavy or might cause the code to malfunction, so this code
only keeps the first three values of the point data,
# that is, the x, y and z values, that are the ones used by the tool

# the code is made for point cloud files in ASCII characters with values separated
by spaces where each line represents a point

import tkinter as tk
from tkinter import filedialog

def process_file(file_path):
    with open(file_path, 'r') as file:
        lines = file.readlines()

    processed_lines = []
    for line in lines:
        values = line.strip().split()
        if len(values) >= 3:
            processed_lines.append(' '.join(values[:3]))

    with open(file_path, 'w') as file:
        file.write('\n'.join(processed_lines))

def open_files_dialog():
    root = tk.Tk()
    root.withdraw()  # Hide the root window
    file_paths = filedialog.askopenfilenames(
        title="Select Text Files",
        filetypes=(("Text Files", "*.txt"), ("All Files", "*.*"))
    )
    if file_paths:
        for file_path in file_paths:
            process_file(file_path)
            print(f"Processed the file: {file_path}")

if __name__ == "__main__":
    open_files_dialog()
```