HouseScan: Building-scale interior 3D reconstruction with KinectFusion

Master's Thesis – Imperial College London

Niklas Hambüchen

Abstract

Recent advances in reconstruction systems like *KinectFusion* allow us to quickly take 3D models of scenes in high quality using low-cost commodity RGB-D cameras. Unfortunately, creating models of whole buildings is difficult because simple geometry prevents camera tracking from traversing doors, and *drift* accumulated from small tracking errors degrades the quality of the model. While existing systems use *loop closure* to correct drift, they do not exploit architectural knowledge when *loop closure* is not applicable.

In this thesis I present a semi-automatic system for performing multi-room, multi-storey interior 3D reconstruction that uses the simple geometry prevalent in buildings to its advantage. Individual rooms are scanned using a *KinectFusion* implementation and cuboids are fit to them. They are combined to a high-resolution, coloured mesh model using a globally consistent room pose optimisation algorithm, in which a constraint graph over the possible positions of each room is reformulated as an inconsistent system of linear equations and then solved with minimal error. The presented system can be applied by non-technical users and aims to bring building-scale reconstruction to a wider audience.

To demonstrate the effectiveness of the system, a house with 23 rooms on 3 levels is scanned and the accuracy of the generated model is evaluated using ground truth obtained with a laser rangefinder. The results show that the reconstruction error of the final model is low, that room pose optimisation does not introduce significant error on top of the error in the individual rooms, and that the achieved accuracy is close to that of conventional 1D consumer measuring tools.

June 2014

Acknowledgements

I would like to thank my supervisor, Andrew Davison, for impressing me with *KinectFusion* a few years ago, and for his guidance throughout my project. Equally I would like to thank my second supervisor, Bernhard Kainz, for his feedback and for coming up with great resources when things did not work well. My respect goes to researchers that make code and data available. With joy I owe my friends: Patrick Chilton and Nandor Licker for technical contributions, Silvia Dobrota for helping me with experiments, Razvan Marinescu for teaching me 3D animation, Wookie Park and Grahame Harris for being dedicated proof-readers, George Papamakarios for his wisdom and Mihaela Rosca for her inspiring rigour. A special mention goes to my personal tutor, Ian Hodkinson, for giving great advice when needed, and for being a prototype of academic awesomeness throughout my four years at Imperial. Finally, I would like to thank my parents for helping me with this project, and for their unconditional support that brought me to this point.

Contents

List of Figures

1 Introduction

3D models will become the photographs of the 21st century.

In recent years, 3D reconstruction has made great advances. Off-the-shelf desktop 3D scanners accompany the rise of 3D printers, and recent milestones in Computer Vision research like *KinectFusion* [1] make it possible to create 3D models in *real-time* by simply moving a commodity RGB-D camera through a scene. This takes little time and the results are of high quality.

A new problem is 3D scanning *at scale*.

Models of whole buildings would greatly benefit interior design, robotics, and other applications where detailed indoor maps are needed. When searching for a new home, we currently have to resort to photos that only give a rough idea of what a property looks like. An interior 3D model would solve this.

Reconstruction at building-scale is difficult because current camera tracking techniques tend to fail in areas with few geometrical features, which appear during door traversals. In addition, camera *drift* accumulated from small tracking errors distorts the geometry of the model. Current systems [2] employ *loop closure* to correct drift, but do not exploit architectural knowledge when *loop closure* is not applicable, such as when separately scanned parts of a building need to be combined.

This thesis presents *HouseScan*, a semi-automatic system to perform multi-room, multi-storey interior 3D reconstruction using commodity RGB-D cameras.

HouseScan implements recent advances in Computer Vision research into a tool that allows non-technical users to create accurate and visually appealing models of buildings with ease. Individual rooms are scanned using *KinFu* [3], an open-source implementation of the original *KinectFusion* paper, and combined into a full model in a semi-automatic fashion.

While one might try to align room models *by hand and eye* using 3D editing software like *Meshlab* [4], this is laborious work, and prone to inaccuracies when dozens of pieces have to be arranged in agreement with each other. It is further complicated by the difficulty of performing precise three-dimensional transformations with common 2D computer input systems, such as keyboard and mouse.

In contrast, creating a large-scale 3D model with *HouseScan* is convenient and fast: The correct alignment of an entire house from individual scans of 23 rooms on 3 levels (see section 3.6) can be achieved in under two hours in a semi-automatic work flow with immediate visual feedback.

HouseScan performs room-pose optimisation based on wall constraints entered by the user, resulting in a combined, globally consistent model whose accuracy surpasses manual assembly.

Further, the experiments described in section 5 demonstrate that the *system error* across multiple rooms in the composite model is lower than the average error of the individual rooms from which it is built. This suggests that the global optimisation employed for room alignment can partially compensate for noise in the *KinFu* based reconstruction.

Interactivity in *HouseScan* is threefold. All the user has to do is

1. perform a free-hand 3D reconstruction of individual rooms using an improved version of *KinFu*, trying to capture as many 3D features in the room as they desire, while making sure a sufficient amount of floor, wall and ceiling parts are captured,
2. choose the 8 corner points of each room aided by automatic suggestions obtained from plane detection, and

3. define the building topology by creating wall connections between rooms.

HouseScan will then optimise the room positions according to the constraints set by the user, and export the composite mesh model in the full original resolution.

Based on the excellent model quality that a *KinectFusion* reconstruction offers, *HouseScan* can be a useful tool for house owners and interior designers by providing a high-resolution three dimensional floor plan. It can also be used to create large-scale models for robotics research, planning, physics simulations or forensics. Lastly, it may ease the work of game designers when creating realistic environments, and enables anyone to capture a precise 3D snapshot of their home at low cost.

1.1 Objectives

The main goals I set out to achieve in this project are to

1. exploit the simple geometry that is common to buildings, making use of prominent walls, presence of a ground plane, abundance of right angles,
2. develop an algorithm to align multiple rooms based on architectural constraints, minimising global error,
3. implement a graphical tool to perform this algorithm interactively,
4. evaluate the error in models created this way and compare it with the error of single-room scans.

My stretch goals are to

5. extend existing *KinectFusion* tools by adding colours to the mesh extraction algorithm,
6. improve *KinectFusion*'s robustness against plain geometry by using architectural information in its tracking, and
7. render a service to the Computer Vision community by creating an extensive RGB-D data set for building-scale reconstruction.

Let us begin.

2 Background and related work

2.1 Floor plan creation

A floor plan is an architectural drawing created by an architect or interior designer that captures the precise locations and dimensions of all rooms in a building.

Common tools used for creating floor plans are tape measures, folding rulers, ultrasonic sensors, and more recently laser rangefinders (see section 2.2.2: Hardware).

Floor plans are often expensive because of the time needed for a professional to visit a property, measure all relevant distances, and then create precise drawings of all floors in orthographic top-down and side perspectives, while calculating eventually missing values from the others. Since walls are not perfectly straight, inconsistencies have to be accounted for. The price for a professionally created floor plan for a multi-level building often exceeds 1000 USD.

2.1.1 Floor plans from depth and conventional cameras

The advent of consumer-grade RGB-D cameras (see section 2.2.2) has inspired methods to create floor plans automatically.

Taylor and Cowley [5] proposed a system to create floor plans with Manhattan structure from single RGB-D images, in which an image segmentation step is followed by solving a labelling problem using dynamic programming. It is implemented on the CPU and does not run in real-time. It also does not try to combine multiple images to create a complete floor plan.

Another system is Walk&Sketch [6], in which an RGB-D video is recorded and processed offline. The authors combine RGB features and downsampled depth images to estimate a camera trajectory. They then perform segmentation to extract polylines, and merge them into a single 2D map using a Manhattan assumption. While this work has extended range compared to [5], its frame processing time of multiple seconds per frame is far above real-time.

Work that predates the availability of consumer-grade depth cameras uses conventional RGB cameras to create floor plans.

In [7] Furukawa et al. construct floor plans from unsorted collections of photos. They use multi-view stereo to generate axis-aligned depth maps that are integrated into a binary 3D voxel grid. As a result, they can reconstruct three-dimensional floor plans with high detail. Like the work mentioned above they uses a Manhattan assumption, which gives their 3D reconstructions the looks of *box worlds*.

In more recent work, Cabral and Furukawa [8] use RGB panorama images to reconstruct piecewise planar floor plans, onto which textures from the original images are mapped. This allows a free-viewpoint visualisation that is almost photorealistic.

Notably, none of the above methods enable real-time feedback during the creation of room models.

2.2 3D Reconstruction

3D *reconstruction* is the concept of building a three-dimensional model of a scene. In contrast to *Computer-aided design*, in which a human designer draws the model manually, in 3D reconstruction a *sensor* is used to directly capture the scene from the environment.

Multiple types of sensors can be used for this purpose; an overview is given in the Hardware section (2.2.2).

3D *rendering* can be understood the opposite of reconstruction. Here, a 3D model is projected into a 2D image to be shown on a screen, displayed as a snap shot from a certain direction. Rendering benefits from the fact that all information needed to produce the output is readily present in the model.

The scene impressions measured by a sensor however are usually partial. Combining multiple of these measurements into a single model is called *registration*, and is often the most difficult part in a 3D reconstruction system.

A good reconstruction system can create a model that accurately reflects the shapes, dimensions and relative positions of objects in the scene. In many cases it can also correctly reflect colours and surface details.

This section describes basic reconstruction concepts, devices and techniques. Practical instances of these can be found in the following sections, in particular in *KinectFusion* (2.3) as a full recognition system.

2.2.1 Models for 3D scenes

There are two types of models frequently used for representing 3D scenes in computers.

Polygon mesh models consist of a set of points, called *vertices*, hovering in the 3D space. Combinations of these points can span surface polygons (usually triangles), from which all objects in the scene are composed. Mesh models are *sparse*, which means that they only store information that contributes to the shape of objects; free space in the scene need not be explicitly stored.

Volumetric models split the represented scene space into small units of volume called *voxels*[1]. Each voxel stores information, such as whether it is part of an object (inside or at its surface), the colour of the represented space, or composing material. Voxels can also contain meta-information. For example, SDF[2] volumes store at each voxel the distance to the closest surface. Volumetric models are often *dense*, meaning that any bit of volume information is explicitly stored.

The complexity and required storage space of a polygon mesh model grow with the number of vertices, but are independent from the physical size of the represented scene. In a volumetric model, the size of the individual voxels puts a lower limit on how finely objects can be represented; the required storage space grows as this volume resolution is increased.

It is possible to compute either type of model from the other, with the need for *interpolation* when values from one model cannot be directly expressed in the other one[3].

The choice of model depends on the application and the available computing resources. We will see examples of how volumetric models are very amenable to being updated by the measurements of depth sensors, and thus are a good intermediate representation for 3D reconstruction.

2.2.2 Hardware

This section compares a range of typical sensors used for capturing the world in three dimensions.

[1] This is equivalent to surfaces of two dimensional of images being split uniformly into *pixels*.
[2] SDF stands for *Surface Distance Function*.
[3] For example, a vertex could fall on the boundary between two voxels.

- **Ultrasonic sensors**

  Replicating the perception system of dolphins and bats, an ultrasonic sensor emits a sound wave to be reflected by an obstacle in front of it, and receives the reflected signal while timing how long it travelled. Knowing how fast sound propagates in the surrounding medium (such as air), the distance to the obstacle can be calculated.

  Ultrasonic sensors are widely available at low prices under 10 USD.

  A disadvantage of ultrasonic sensors is that they typically measure the distance to a single point, and thus have to be moved in two degrees of freedom to scan over a surface. The speed of sound limits how fast these sensors can operate, since they have to wait for the impulse to return before it can take the next measurement.

  A commodity ultrasonic sensor is displayed in Figure 4.1.

- **Laser rangefinders**

  A laser rangefinder projects a laser beam forward to be reflected from the obstacle and received by its camera.

  While being conceptually similar to ultrasonic sensors, making use of the speed of light allows laser rangefinders to operate faster. In addition, they project a line, recognising the difference to multiple points at once, which is why they need to move in only one degree of freedom to scan over a surface.

  Rotating laser rangefinders with a range up to around 5 m are often used in robotics, but are expensive in the thousand USD range. Handheld 1D laser rangefinders for distance measurements are available starting from about 70 USD, one of which is shown in Figure 4.1.

- **Projective Range Cameras**

  A projective range camera shines a grid or point cloud of light into the scene. It uses a camera to inspect how the pattern was projected on the obstacle surfaces, and calculates their geometry from it.

  Since the emitted pattern is already two dimensional, no degree of freedom is needed to scan a surface and full surface distance information can be extracted in every frame that the camera produces – usually 30 frames per second.

  In order to hide the light pattern from human eyes, often infrared light is used (made visible in Figure 2.2).

  Projective range cameras have found their way into the gaming industry and are available for under 100 USD, with the *Kinect* by Microsoft being available for their *Xbox* game console, and the Asus *Xtion* as a standalone device (see Figure 2.1). Both cameras use a sensor component developed by *PrimeSense*. The visible range of these cameras lies between 50 - 400 cm. They are also equipped with an additional, conventional RGB video camera, and are thus called *RGB-D* cameras (RGB with depth).

  These cameras make the distance information available to programmers as a *depth image*, which is similar to a grey scale image of a normal video camera, except that each pixel intensity represents the distance from the camera to the object shown at that pixel.

  Consumer-grade RGB-D cameras are reasonably accurate, but often feature significant noise in depth data at high frequency.

  Common RGB-D cameras can be accessed using the *OpenNI* framework [9] originally developed by *PrimeSense*.

Figure 2.1: Left: A Microsoft Kinect RGB-D camera, with IR projector, RGB camera, IR camera (from left to right). Right: An Asus Xtion Live RGB-D camera.

- **Time of Flight Cameras**

  Time of Flight Cameras (ToF) are essentially two-dimensional range scanners, illuminating the whole scene with a single light pulse. They combine the advantages of laser scanners (high precision) and projective range cameras (full depth capture per frame).

  Similar to RGB-D cameras, ToF cameras are entering the gaming market at the time of writing, and consumer models are becoming available for around 150 USD.

- **Normal video cameras**

  It is also possible to do 3D reconstruction with conventional video cameras that do not measure depth directly.

  Photometric stereo makes use of two or more cameras, where corresponding points on each camera are searched for. The actual distance to the point can then be *triangulated* given that the distance between the cameras is known.

  An alternative approach that only uses a single camera is *Simultaneous Localisation and Mapping with a Single Camera* (MonoSLAM [10]), where the location of the camera is continuously estimated from what it sees, and distances to recognised objects are triangulated from their position before and after camera movements.

  Intuitively, 3D reconstruction is much more difficult using a normal camera than when using a sensor that provides depth information.

2.2.3 Technique: SLAM

In robotics, *mapping* describes the process of creating a map of the robot's environment. *Localisation* means to find the position of the robot in this map.

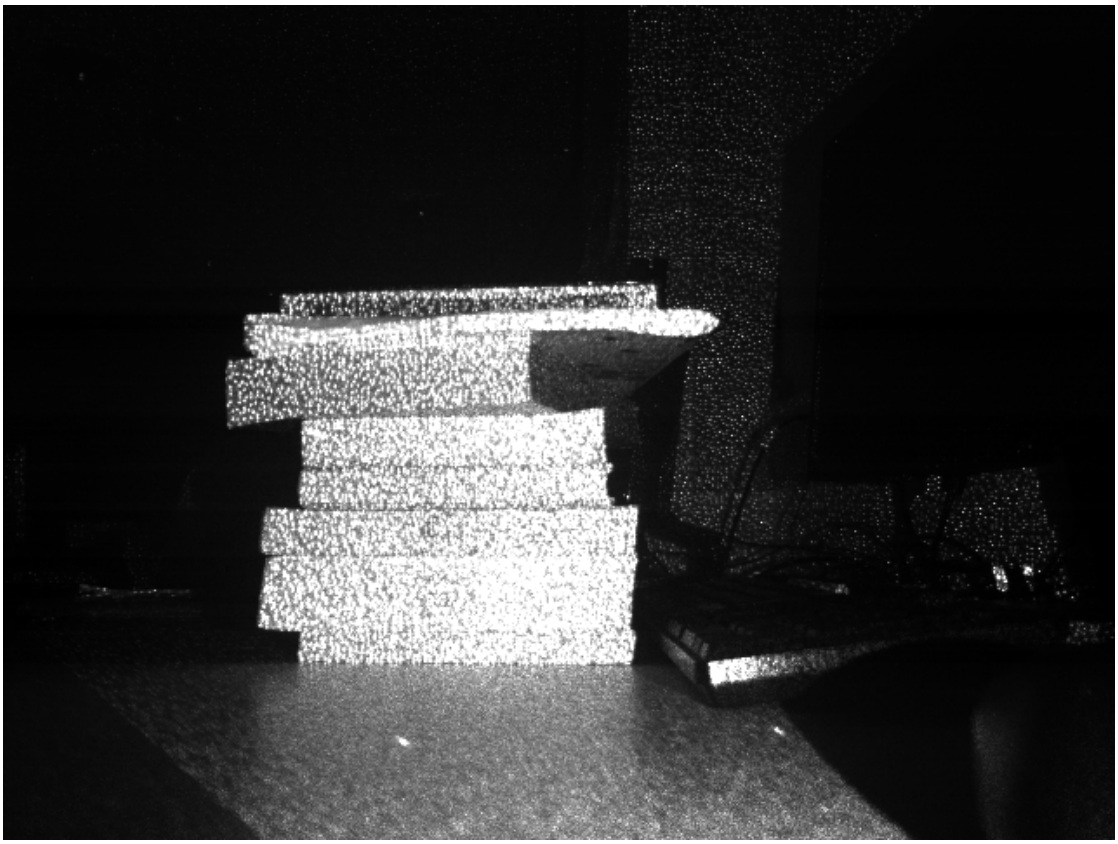

Figure 2.2: Infrared points emitted by a Kinect, made visible with its IR camera. Courtesy of Patrick Chilton.

When reconstructing a scene using one of the depth sensors described above, it is important to know the position of the camera to be able to correctly update the map being built.

If the position of the camera is not known and the camera image is the only source of information, the only way to find the position is to estimate it from comparing what the camera sees with the map constructed so far.

This creates a dependency circle:

- to find the position, we need to have a working map;
- to update the map, we need to know the camera position.

This problem is called *Simultaneous Localisation and Mapping*, and keeps making robots struggle.

Even when no robot is involved and a scene shall be reconstructed from a camera held and moved by a human, SLAM needs to be performed in order to liberate the human from telling the reconstruction system where the camera is. In this case, the localisation is called *camera tracking*.

Section 2.3 describes how *KinectFusion* solves the SLAM problem, and in particular how it tracks the camera using the ICP algorithm (2.4.1).

2.2.4 Technique: Loop closure

When reconstructing a scene and building a map using a sensor, the position of the sensor is always uncertain:

- Moving a robot with wheels of known circumference for a known time gives a good estimate of the position after the movement, but slippery ground, uneven surfaces and wind can distort the estimate.
- Fast movements of a camera performing SLAM can make tracking inaccurate (for an example, see the end of "The ICP algorithm" (2.4.1)).

This uncertainty introduces a *location error* that leads to points of the scene being stored with an offset from their actual location, and *accumulates* over the duration of the scene recognition (the errors add up). Especially frequent are direction-biased drifts and overestimation of angle movements.

When completing the registration of a circular scene (a *loop*, e.g. around a building), we will encounter some objects twice.

Due to the accumulated positional error, these objects seem to appear in two different places of the map, leading to a knowledge contradiction. It may even be the case that objects from the beginning and end of the loop *overlap*, and are incorrectly merged into each other.

Naturally, we seek to identify these situations in which we encounter a view we have already registered before. At that moment, the error we have accumulated so far will become obvious and measurable; we can use it to correct the mapping of all points we have registered so far from the beginning until the end of the loop. This error correction step is called *loop closure*, and often a *pose-graph relaxation* algorithm [11] is used to adjust past measurements.

A loop closure algorithm in the context of scene recognition for robotics is described in [12]. A video demonstrating loop closure on a street network based on photos from a driving car can be found on [13].

2.3 *KinectFusion*

KinectFusion [1] is a 3D reconstruction system that can create models from a scene in real time. It uses SLAM with a Kinect depth camera and its core innovation was to make extensive use of data-parallel computing on graphics processors, which allows it to build a *dense* volumetric model at the frame rate of the depth camera (30 FPS).

Earlier reconstructions were not dense. As Henry *et al.* [14] describe:

> "Considering that each frame from the RGB-D camera gives us roughly 250,000 points, it is necessary to create a more concise representation of the map. One option is to downsample the clouds. However, it is more appealing to incorporate all the information from each frame into a concise representation for visualisation."

Henry *et al.* then introduce *surfels*, 2D coloured patches oriented in 3D space, to avoid the high computational costs of a dense representation.

KinectFusion however does work directly on the full dense point clouds received by the depth camera. This allows the *KinectFusion* algorithms to be significantly simplified, reducing it to a pipeline of four stages (based on [15], see also Figure 2.3):

1. **Depth map conversion**. The depth image of the RGB-D camera is converted to a set of vertices with attached normals in the camera coordinate space.
2. **Camera tracking**. The ICP algorithm is used to compute a 6 degrees-of-freedom camera transformation that aligns the points of the most recent frame with the synthetic depth map from step 4.
3. **Volumetric integration**. Based on the calculated camera pose, a uniform voxel grid is updated. Each voxel contains a *Truncated Signed Distance Function* (TSDF) that indicates the distance to the closest surface. For each frame, all voxels are traversed, and updated with a running TSDF average based on the depth value of the camera pixel to which the voxel projects given the current camera pose.
4. **Raycasting**. The voxel volume is traversed; zero-crossings in the TSDF values describe the implied surface. Normal vectors are computed as the gradient of the TSDF at the zero-crossings, and the result is rendered to a shaded grey-scale image. In addition, the raycaster creates a *synthetic depth map* from the voxel grid. It exhibits low noise and is used as the correspondence for the ICP algorithm from step 2.

The barrier of computational cost for keeping a dense uniform voxel grid is overcome leveraging the increased computational power provided by GPUs. The *KinectFusion* authors chose variants of the core algorithms that can operate per-pixel / per-voxel, making them amenable to the data-parallelism provided by modern GPUs. In fact, all four stages in the pipeline are implemented for the GPU, which provides sufficient performance to not only work on a *dense* model, but also to update it in *real time*; [14] did prognosticate, but not feature this. It allows immediate feedback to a user scanning a scene.

However, [14] did implement *loop closure* and global pose optimisation which *KinectFusion* lacked; reasons for this are described in the section Problems with KinectFusion (2.3.2).

Unfortunately, the source code of the original KinectFusion implementation, as written in cooperation with Microsoft, has not been released. However, independent open-source implementations have been developed based on the concepts described in the original paper [1]; they are described in the next section.

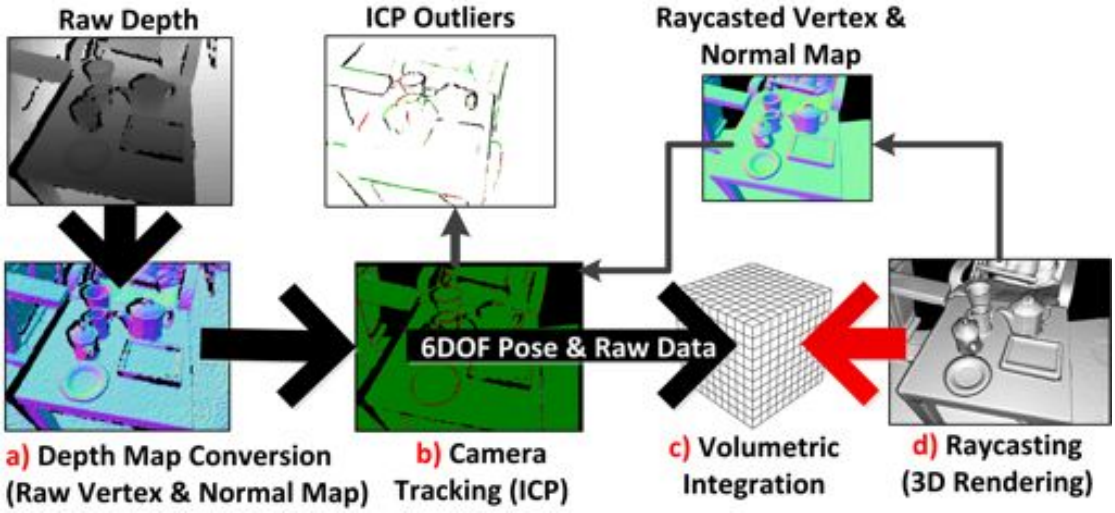

Figure 2.3: KinectFusion pipeline. The top left shows the view of the depth camera, the bottom right the reconstructed 3D scene. Source: [15]

2.3.1 Implementations: *KinFu* and *kfusion*

This section discusses two open-source implementations of *KinectFusion*. They are the only ones available to the public as of writing.

2.3.1.1 *KinFu*

KinFu was the first available open-source implementation of *KinectFusion*. It was developed as part of the *Point Cloud Library* (PCL [16]) in 2011, a few months after publication of the original paper.

In addition to the core features described in the paper, it features a point cloud extraction functionality that converts the complete TSDF voxel grid into a coloured point cloud with attached normals, in order to be used by other parts of PCL or externally. It also containes a GPU implementation of the *Marching Cubes* algorithm [17] that computes a triangle mesh instead of a point cloud. While this mesh was originally uncoloured, I extended it with a colour extraction functionality as described in section 3.2.

The *Point Cloud Library* is a collective effort, with some of the code contributed as one-off contributions and lacking thorough review and active maintenance. When I discovered *KinFu*, some features did not work or crashed the program, and parts of the code, such as the *Marching Cubes* implementation, are cryptic and poorly documented. I fixed some of these issues as a side effect of this thesis. Still, *KinFu* is the most feature-complete *KinectFusion* implementation available, and many related projects are built upon it.

2.3.1.2 *kfusion*

kfusion [18] is an alternative *KinectFusion* implementation that is especially attractive due to its minimalism.

While *KinFu* makes extensive use of functionality available in the 100000 lines of PCL code, *kfusion* is almost self-contained within 2000 lines of code.

kfusion does not feature point cloud or triangle mesh extraction mechanisms, but its camera tracking implementation works more reliably than *KinFu*'s (the reason for this is discussed in the next section).

2.3.2 Problems with *KinectFusion*

There are a few shortcomings of *KinectFusion* that prevent it from being used by itself for reconstructing multi-room and building-scale environments, some of which have been addressed (see next section):

- **Bounded reconstruction volume**

  KinectFusion operates in a fixed voxel grid. While its real-world dimensions can be changed upfront, once chosen the whole area to be scanned must fit within this cube; the pre-determined scan volume cannot be dynamically extended. The available working memory of current graphics cards limits the resolution to 512^3 voxels, making a detailed multi-room scan infeasible.

  This issue has been addressed by the *KinFu Large Scale* and *Kintinuous* implementations discussed in the next section.

- **Track loss**

  KinectFusion uses the ICP algorithm (see 2.4.1) to track the position of the camera. ICP assumes that the camera pose changes very slightly between two frames, and that the scene is sufficiently featureful to be able to converge. On rapid movements or when confronted with plain walls, IPC fails and tracking is lost. This happens especially when walking through doors, which is essential for doing multi-room scans.

  KinFu tries to detect track loss by checking if the determinant of the covariance matrix described in section 2.4.2.4 is smaller than a chosen small threshold.

  However, it seems that this is not sufficient to detect track loss reliably, leading *KinFu* into assuming a wrong position of the camera, and consequently corrupting the model constructed so far. By contrast, *kfusion* compares the disagreement of the reconstructed model and the camera depth image, which detects track loss in almost all cases. It also allows to *resume* the scan after a track loss in most situations, making it much easier to scan scenes in which ICP fails often.

2.3.3 Extensions of *KinectFusion*

There are a number of projects which improve upon the original KinectFusion paper.

KinFu Large Scale [19] extends *KinFu* by implementing a movable reconstruction volume. When the camera position exceeds a threshold distance from the centre of the current scan cube, the volume is *shifted*, centring a new cube at the current camera position. The old cube is transferred from GPU memory to main memory (and thus can automatically be paged out to disk by the Operating System), allowing *KinFu Large Scale* to scan unbounded environments.

Notably, the voxel grid resolution is still fixed and does not automatically adjust (e.g. when coming very close to an object to perform a detail scan); the shifting implementation however allows the cube's 512^3 voxels to inhabit a sufficiently small real world volume (e.g. 1 m) and creates scans with high detail density. Yet I want to emphasise that the voxel resolution is uniform across the whole scan – this is different from increasing the resolution as the camera gets closer to scanned surfaces.

KinFu Large Scale is now part of the PCL library. It comes with an offline tool to convert the recorded volumetric models to a mesh representation, and colourise it projecting RGB pictures (taken at regular intervals) into the mesh.

A shortcoming of *KinFu Large Scale* is a noticeable drop in frame rate since "additional operations are taking place in the frame processing loop as a result of the large scale implementation" [20]. This also

increases the problem of losing track in scenes with poor geometry (such as transitions between rooms) in comparison to *KinFu*, so scanning multiple rooms with doors in between is still problematic. Yet it still runs in real time on modern hardware.

A similar moving volume implementation is given in [21], with the main difference being that the scanning volume can also be rotated (*KinFu Large Scale* only translates it). The authors argue that "this can be useful to control its orientation, e.g. to maximise overlap of the camera frustum or to align the volume with task-relevant directions". According to the authors, this implementation has been proposed to be integrated into PCL, but at the time of writing the code does not seem to be publicly available.

Another implementation created during the same time is *Kintinuous* [2]. In addition to shifting the recognition cube, it computes a triangular mesh from the volumetric model online, and adds *loop closure* detection and *pose-graph optimisation*. The authors report being able to accurately reconstruct continuous areas across a two story apartment (demo video at [22]). While they apparently pledged at *CVPR2012* to make the source code available in the future [23], it is not available at the time of writing (two years later) for conducting comparisons with other techniques.

Finallly, Steinbrücker et al. [24] recently developed a real-time system that uses an octree representation for extending the reconstruction range. Similar to *Kintinuous*, it performs *loop closure* and *pose-graph optimisation*. In contrast to all implementations mentioned above, their work uses a probabilistic framework for position estimation. It also contains a detailed evaluation of its trajectory error on publicly available benchmark sequences, demonstrating it to be substantially lower than that of *KinFu*.
While their previous, CPU based, non-realtime work [25] is available as open-source software, it is unclear at the time of writing whether the code for this new GPU based system is available; I am waiting for a response from the authors.

2.4 Mesh alignment

A critical task in 3D reconstruction is aligning two meshes.

First, it is needed when two partially overlapping scans shall be combined into one. In this case, aligning the parts present in both scans will "glue" the two meshes together in the correct orientation.

Second, it is useful for camera tracking. When moving the depth camera, two slightly misaligned surfaces are recorded, and from aligning them we can calculate how far the camera moved and how far it rotated.

The following section discusses an often used algorithm for mesh alignment, which is a core component of KinectFusion and the systems derived from it.

2.4.1 The Iterative Closest Point (ICP) algorithm

ICP (*Iterative Closest Point*, although [26] observes that "Iterative *Corresponding* Point" is probably a more appropriate name) is an algorithm for aligning overlapping three-dimensional models with a *rigid* transformation. The *points* to be aligned are usually vertices in a mesh, but can also be taken from voxels from a dense point cloud.

There are many variants of ICP; a thorough comparison has been performed by Rusinkiewicz and Levoy in 2001 [26], especially in regard to their convergence performance.

They identified and discussed six core stages of the algorithms. Quoting:

1) **Selection** of some set of points in one or both meshes.
2) **Matching** these points to samples in the other mesh.
3) **Weighting** the corresponding pairs appropriately.
4) **Rejecting** certain pairs based on looking at each pair individually or considering the entire set of pairs.
5) **Assigning** an error metric based on the point pairs.
6) **Minimising** the error metric.

Rusinkiewicz and Levoy do not recommend matching (2) the *closest* point:

> "In the presence of noise and outliers, the closest-point matching algorithm potentially generates large numbers of incorrect pairings when the meshes are still relatively far from each other.""

The ICP algorithm as part of *KinectFusion* instantiates the above stages with the following choices:

1) As the source mesh, all points from the current depth image of the Kinect are selected; the destination mesh with normals is calculated from the part of the synthetic model that lies in the frustum of the last camera pose.
2) The *projective data association* method is used, since it is "less sensitive to the presence of noise" [26] and is well-suited to a GPU implementation. This means associating points that project to the same camera pixel.
3) All correspondences carry equal weight.
4) Outlier rejection by testing if "the Euclidean distance and angle [..] are within a threshold" [1].
5) A *point-to-plane*, *least squares* error metric.
6) A linear approximation as described in [27][4] for optimising the error metric, made possible by assuming the camera is only moved slightly between each frame.

One of the most important choices for the performance of the ICP algorithm is the choice of the error metric (5). A *point-to-point* euclidean distance is probably the most intuitive choice, but [26] demonstrated (in their Figure 15) that a *point-to-plane* error metric makes ICP converge significantly faster, while also reducing the chance of getting stuck in local minima. This however comes with the trade-off that certain geometries, like parallel planes with holes, cannot be aligned correctly, since the *point-to-plane* error will not decrease when the planes are moved in parallel against each other. This is the main reason why *KinectFusion* loses track on scenes with few geometric features.

For some scenes, this problem can be mitigated by selecting point samples that constrain unstable transformations [28], but scenes with very little geometric complexity remain problematic.

2.4.2 ICP: Deriving the minimisation step

To make the reader familiar with the mathematics involved in the ICP algorithm, this section describes the final **minimisation** step in three different ICP settings[5].
One of them is described in detail to highlight core concepts, while the other two are only discussed briefly to point out the differences.

[4] Rusinkiewicz and Levoy already hinted at this idea in [26], but [27] describes it in detail.
[5] Where the setting is defined by the choices we made for stages 1-5.

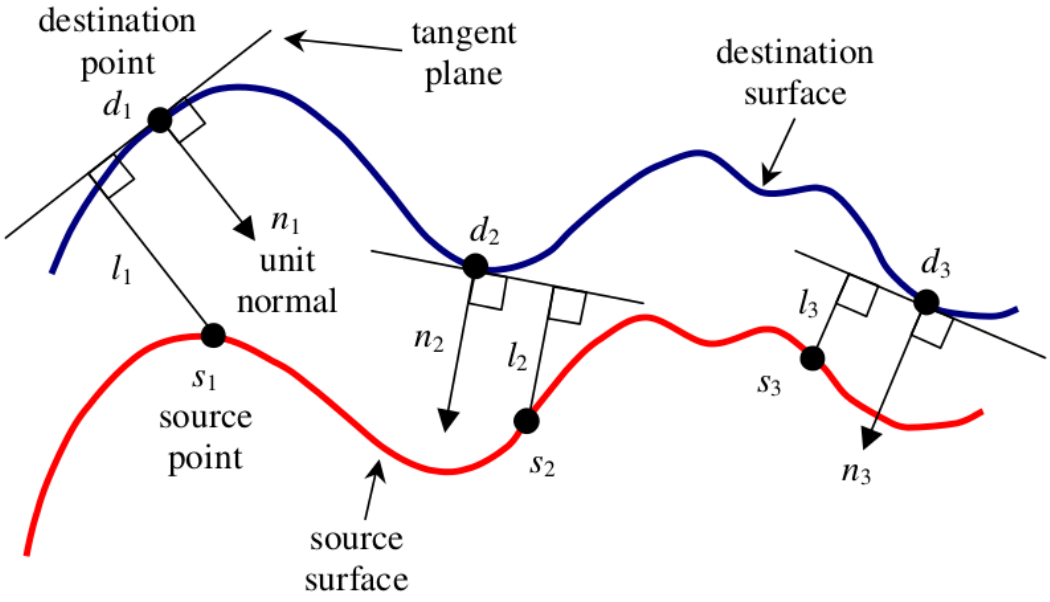

Figure 2.4: Point-to-plane error between two surfaces. Source: [27]

2.4.2.1 Point-to-Point ICP In 1987, the original *point-to-point* ICP algorithm was described by Arun, Huang and Blostein [29].

They noticed that the actual optimisation was only dependent on the rotation, and that the translation could be recovered as mentioned in the next section.

For two sets of points, let their elements s_i and d_i (*source* and *destination*) be in correspondence.

We seek to find the rotation matrix R that minimises:

$$\sum_{i=1}^{N} ||d_i - Rs_i||^2$$

Their algorithm is as follows:

1. First calculate the means c_s and c_d (c for "centre").
   Subtract the means from the data, obtaining $d'_i = d_i - c_d$ and $s'_i = s_i - c_s$.
2. Calculate the 3×3 scatter matrix[6]
   $$H = \sum_{i=1}^{N} s_i d_i^T$$
   where $s_i d_i^T$ is the outer product of the two point vectors.
3. Find the Singular Value Decomposition (SVD) of H,
   $$H = U \Lambda V^T$$

[6] A scatter matrix is a covariance matrix that is not scaled down by the number of samples (N).

4. Calculate
$$X = VU^T$$
5. If $det(X) = +1$, then $R = X$ is the solution.
   If $det(X) = -1$, then the alignment fails (they demonstrate that this is a rare case).

Note how the solution consists of the left and right matrix from the SVD only, with the middle Λ matrix left out.
This reflects the idea that we are seeking a *rigid* rotation, not permitting any scaling, and aligns well with the common intuitive interpretation of the SVD that the three matrices in the decomposition form a sequence of a rotation, a scaling, and another rotation ([30], see [31] for a graphical animation).

2.4.2.2 The small angle assumption

Derivation based on [27].

In the above, we calculate a full 3D rotation matrix R of form

$$\begin{pmatrix} \cos\gamma\cos\beta & -\sin g\cos\alpha + \cos\gamma\sin\beta\sin\alpha & \sin\gamma\sin\alpha + \cos\gamma\sin\beta\cos\alpha \\ \sin\gamma\cos\beta & \cos\gamma\cos\alpha + \sin\gamma\sin\beta\sin\alpha & -\cos\gamma\sin\alpha + \sin\gamma\sin\beta\cos\alpha \\ -\sin\beta & \cos\beta\sin\alpha & \cos\beta\cos\alpha \end{pmatrix}$$

as the result of a closed-form solution of a minimisation problem; α, β and γ are the three angles of rotations around the coordinate axes.

Sometimes, we wish to work with a *linearised* rotation instead, namely when

- it is faster to compute,
- a closed form of the non-linearised rotation is not available, or because
- the linearised solution extends more easily to a globally consistent ICP for a set of *n* point clouds, instead of only two [32].

A common instance of rotation linearisation is the *small angle assumption*. For small angles, we can approximate that $sin\, x \approx x$ and $cos\, x \approx 1$. Substituting this into R, we get a simplified, *linearised* rotation

$$\begin{pmatrix} 1 & \alpha\beta - \gamma & \alpha\gamma + \beta \\ \gamma & \alpha\beta\gamma + 1 & \beta\gamma - \alpha \\ -\beta & \alpha & 1 \end{pmatrix}$$

By further assuming that the product of two or more small angles is negligibly close to zero, we obtain the final result

$$\begin{pmatrix} 1 & \gamma & \beta \\ \gamma & 1 & -\alpha \\ -\beta & \alpha & 1 \end{pmatrix}$$

which is useful for the following two sections.

Aside. It is useful to visualise what the small angle assumption does geometrically (see Figure 2.5): For the 2D small-angle rotation matrix $\begin{pmatrix} 1 & -\alpha \\ \alpha & 1 \end{pmatrix}$ we see that for angles up to $90°$, instead of along a unitary circle, unit axis points move along the sides of a unitary square (which is a circle in the maximum (L_∞) norm.

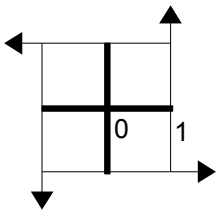

Figure 2.5: A visualisation of a rotation under the small angle assumption.

2.4.2.3 Linearised Point-to-Point ICP This derivation is based on [32] and augments it by explaining the details.

Let

$$E(R,t) = \frac{1}{N}\sum_{i}^{N} \|s_i - (Rd_i + t)\|^2$$

be our cost function to minimise, where s_i and d_i are the points in correspondence, and $[R|t]$ is the desired transformation as a homogeneous matrix.

As [32] discusses between equations (4) and (5), it is sufficient to calculate R and then recover $t = c_s - Rc_d$.

Calculating R happens on mean-subtracted data: $s'_i = s_i - c_s$ and $d'_i = d_i - c_d$. Our new cost function is:

$$E(R) = \frac{1}{N}\sum_{i}^{N} \|s'_i - Rd'_i\|^2$$

With the small angle assumption, the rotation matrix has form

$$R = \begin{pmatrix} 1 & -\theta_z & \theta_y \\ \theta_z & 1 & -\theta_x \\ -\theta_y & \theta_x & 1 \end{pmatrix}$$

Pulling out the diagonal, we get

$$
\begin{aligned}
s'_i - Rd'_i &= s'_i - \left[\boldsymbol{I} + \begin{pmatrix} 0 & -\theta_z & \theta_y \\ \theta_z & 0 & -\theta_x \\ -\theta_y & \theta_x & 0 \end{pmatrix} \right] d'_i \\
&= s'_i - d'_i - \begin{pmatrix} 0 & -\theta_z & \theta_y \\ \theta_z & 0 & -\theta_x \\ -\theta_y & \theta_x & 0 \end{pmatrix} d'_i
\end{aligned}
$$

Since we desire to find $\theta_x, \theta_y, \theta_z$, we combine them in a vector we will solve for:

$$
\begin{aligned}
s'_i - Rd'_i &= s'_i - d'_i - \begin{pmatrix} 0 & -\theta_z & \theta_y \\ \theta_z & 0 & -\theta_x \\ -\theta_y & \theta_x & 0 \end{pmatrix} \begin{pmatrix} d'_{i,x} \\ d'_{i,y} \\ d'_{i,z} \end{pmatrix} \\
&= s'_i - d'_i - \underbrace{\begin{pmatrix} 0 & d'_{i,z} & -d'_{i,y} \\ -d'_{i,z} & 0 & d'_{i,x} \\ d'_{i,y} & -d'_{i,x} & 0 \end{pmatrix}}_{D_i} \underbrace{\begin{pmatrix} \theta_x \\ \theta_y \\ \theta_z \end{pmatrix}}_{\theta} \\
&= s'_i - d'_i - D_i\theta
\end{aligned}
$$

Substituting this into the cost function and minimising:

$$
\begin{aligned}
& \underset{R}{\operatorname{argmin}}\, E(R) \\
= \; & \underset{\theta}{\operatorname{argmin}}\, \tfrac{1}{N} \textstyle\sum_i^N \|s'_i - d'_i - D_i\theta\|^2 \\
= \; & \underset{\theta}{\operatorname{argmin}}\, \textstyle\sum_i^N ((s'_i - d'_i) - D_i\theta)^T ((s'_i - d'_i) - D_i\theta) \\
= \; & \underset{\theta}{\operatorname{argmin}}\, \textstyle\sum_i^N \underbrace{(s'_i - d'_i)^T (s'_i - d'_i)}_{\text{constant}} - \underbrace{\underbrace{(s'_i - d'_i)^T D_i\theta}_{} - \underbrace{(D_i\theta)^T (s'_i - d'_i)}_{}}_{\text{identical scalars}} + (D_i\theta)^T D_i\theta \\
= \; & \underset{\theta}{\operatorname{argmin}}\, \textstyle\sum_i^N -2(s'_i - d'_i)^T D_i\theta + \theta^T D_i^T D_i\theta
\end{aligned}
$$

Taking derivatives and setting equal to 0 to find the minimum (we use denominator layout for matrix calculus):

$$
\begin{aligned}
0 &= \tfrac{d}{d\theta} \left[\textstyle\sum_i^N -2(s'_i - d'_i)^T D_i\theta + \theta^T D_i^T D_i\theta \right] \\
&= \textstyle\sum_i^N -2[(s'_i - d'_i)^T D_i]^T + 2D_i^T D_i\theta \\
&= \textstyle\sum_i^N -2D_i^T (s'_i - d'_i) + 2D_i^T D_i\theta
\end{aligned}
$$

Rearranging

$$
\begin{aligned}
& \textstyle\sum_i^N D_i^T D_i\theta &&= \textstyle\sum_i^N D_i^T (s'_i - d'_i) \\
\Leftrightarrow \; & \left(\textstyle\sum_i^N D_i^T D_i \right) \theta &&= \textstyle\sum_i^N D_i^T (s'_i - d'_i)
\end{aligned}
$$

we obtain an equation of form $Ax = b$, where $x = \theta$, and can thus solve for θ using a standard least-squares solver.

2.4.2.4 Linearised Point-to-Plane ICP

This is what *KinectFusion* uses, based on [27].

Following up on the *small angle assumption* from above, we seek a projection matrix

$$M = \begin{pmatrix} 1 & \gamma & \beta & t_x \\ \gamma & 1 & -\alpha & t_y \\ -\beta & \alpha & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

that moves and rotates the points s_i such that the distance between the moved points and the tangential planes defined by the surface normals n_i is minimal:

$$\arg\min_M \sum_i^N ((Ms_i - d_i) \cdot n_i)^2$$

Here, $\cdot$ is the scalar product, and n_i is the normal vector on d_i (see Figure 2.4).
Surface normals are readily available from *KinectFusion* depth images.

In [27], Low shows in detail how to reformulate this problem such that the unknowns $\alpha, \beta, \gamma, t_x, t_y, t_z$ are gathered in a vector x, such that

$$\arg\min_M \sum_i^N ((Ms_i - d_i) \cdot n_i)^2 = \arg\min_x |Ax - b|^2$$

where A is an $N \times 6$ matrix containing linear combinations between n_i and s_i and b is an N-vector built from linear combinations of the s_i, d_i and n_i.

This standard linear least-squares problem has a well-known solution:

$$A^T Ax = A^T b$$

Instead of getting x directly from the pseudo-inverse A^+ (obtained by left-multiplying with the inverse[7] of $A^T A$ to obtain $A^+ = (A^T A)^{-1} A^T$, or like Low does it, using the SVD of A) with the result $x = A^+ b$, implementations like *KinFu* prefer to compute $C = A^T A$ and $D = A^T b$ on the GPU using a log-parallel reduction, and then solve the constant size (C has dim. 6×6) linear equation system

$$Cx = D$$

using a standard least-squares solver on the CPU.

2.5 Plane detection

When analysing point clouds, it is often desired to extract planes from these point clouds. This section discusses three approaches.

Section 3.3 discusses which method was chosen for *HouseScan*.

[7] Which only exists if A has linearly independent columns. That means that no data point is a linear combination of the others. This is usually not true when A is obtained from arbitrary point clouds, so the mentioned alternatives are preferred.

2.5.1 Region Growing

A plane detection method specifically fitted to range sensors has been described by Poppinga *et al.* [33].

It performs *region growing* starting from random points sampled from the point cloud by repeatedly adding nearest neighbours to a set of potential plane point candidates, as long as a plane can be fitted on this set with a mean square error less than a fixed threshold.

The plane fitting algorithm follows a standard *Principal Components Analysis* (PCA) technique based on eigenanalysis. The covariance matrix of all points is computed, and its eigenvector with lowest corresponding eigenvalue is the normal vector to the optimal plane.

The key contribution of [33] is to reformulate the algorithm to work incrementally, allowing it to quickly recompute the optimal planes as more points are added to the cloud. In addition, it copes well with images from noisy sensors like the *Kinect*, and speeds up the search for nearest neighbours by making use of the neighbourhood relation in range images. This makes it suitable for detecting planes in *real time* during 3D reconstruction.

2.5.2 Hough Transform

A common method in Computer Vision to detect any kind of parametric object is the *Hough Transform*. It defines a quantised space of counters (the *voting space*) over the parameters of the object type to recognise (e.g. m and c in the object equation $y = m \cdot x + c$ when detecting lines). Then each sample point (e.g. pixel/voxel) *votes* for possible parameters by increasing the counter for each combination of parameters that satisfies the object equation given the sample.

Example: In line detection, the sample point $(5, 3)$ in an image would vote for the line $m(c) = \frac{y-c}{x}$ in the *m*-*c*-parameter space.

After all sample points have voted, the *buckets* with the most votes in the quantised parameter space describe the most probable objects.

A problem with the Hough Transform as explained so far is the unboundedness of the voting space (also called the *accumulator*), making it inefficient to represent or access in a computer. A common way of addressing this for 2D images to transform the voting space, representing lines as an angle ϕ with offset from the origin d, which allows bounding the voting space between $(-1, 1)$ and $(0, 180°)$.

For the example of finding lines in 2D given n points, with a voting space raster resolution of $n_\phi \times n_d$ we have to perform up to $n \cdot n_\phi \cdot n_d$ vote count increases and need $n_\phi \cdot n_d$ space.

In the general case, to recognise parametric objects with n samples having p parameters with raster resolution of r per parameter, we need

$$n \cdot r^p$$

update operation and r^p space.

Planes in 3D point clouds have $p = 3$ parameters, so a lot of memory is needed to represent the voting space. Using a depth camera with $640 \cdot 480$ depth measurements per frame with 30 frames per second, up to $9216000 \cdot r^3$ updates have to be performed each second. This computational cost makes using the Hough transform for detecting 3D planes in point clouds difficult.

A recent review [35] revives the idea of using the Hough Transform for plane detection, evaluating different variants of the algorithm. In particular, it finds that the *Randomised Hough Transform* can exceed other techniques (including the Region Growing method discussed above) in terms of runtime performance while maintaining high detection rates.

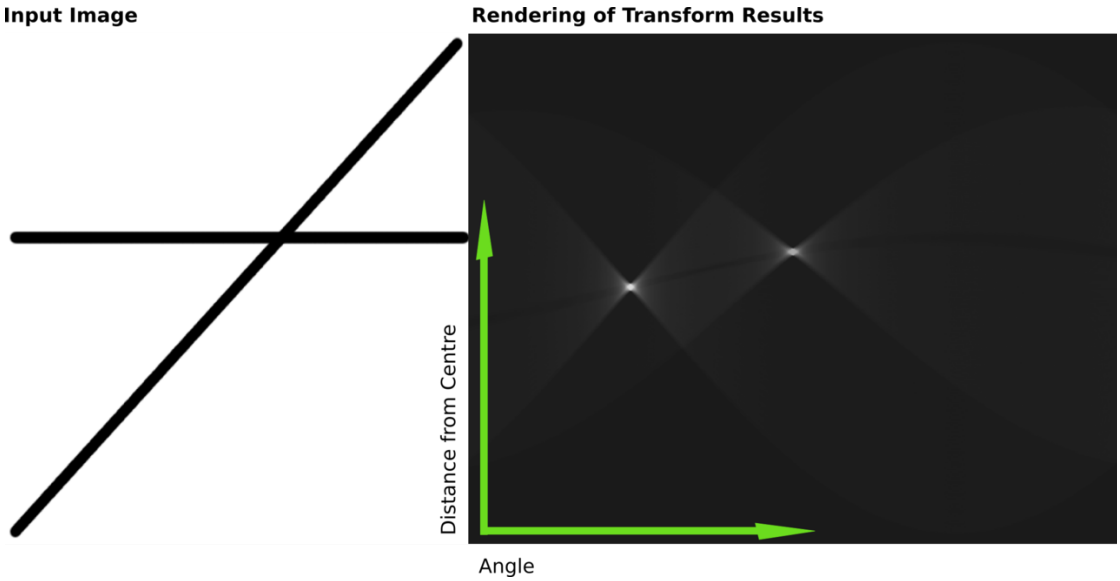

Figure 2.6: The Hough transform with a sine field accumulator. Two lines and their equivalents in the voting space are shown. Source: [34]

2.5.3 RANSAC

RANSAC is short for *Random Sample Consensus*, and a method to fit arbitrarily parametrised mathematical models to noisy data, while being robust to outliers.

It works by randomly selecting just enough samples from the noisy data to estimate parameters the sought model, and then checking how much of the remaining data agrees with these parameters. This is repeated a fixed number of times, after which the parameters with the strongest agreement are chosen.

For finding planes in point clouds, it works like this:

1. Randomly select 3 points from the cloud.
2. Calculate the parameters of the plane that goes through them.
3. Check how much the remaining points agree with this plane by counting how many of them are closer to it than a chosen threshold.
4. Repeat steps 1-3 a fixed number of times, keeping track of those plane parameters that have the most voters.
5. Optionally, check once again which points voted for the winning plane, and then re-estimate the plane parameters from *all of them* instead of only the three that were chosen randomly.
   This yields parameters that explain the final plane better, and is usually done using Principal Component Analysis (PCA): The covariance matrix of the mean-subtracted voting points is eigen-analysed, and the eigenvector corresponding to the smallest eigenvalue is the normal vector of the optimal plane[8].
6. Remove all points assigned to the plane from the cloud, and continue with step 1 until as many planes as desired are found.

[8]This is because the normal vector is the direction of smallest variance in the point cloud.

3 HouseScan: Concept and implementation

This section describes the ideas behind and implementation of *HouseScan*, a system for semi-automatic building-scale interior 3D reconstruction.

With *HouseScan*, a user scans multiple rooms independently with a low-cost handheld RGB-D camera (2.2.2) using an implementation of *KinectFusion* (2.3), and interactively combines them to a full 3D model.

3.1 Overview

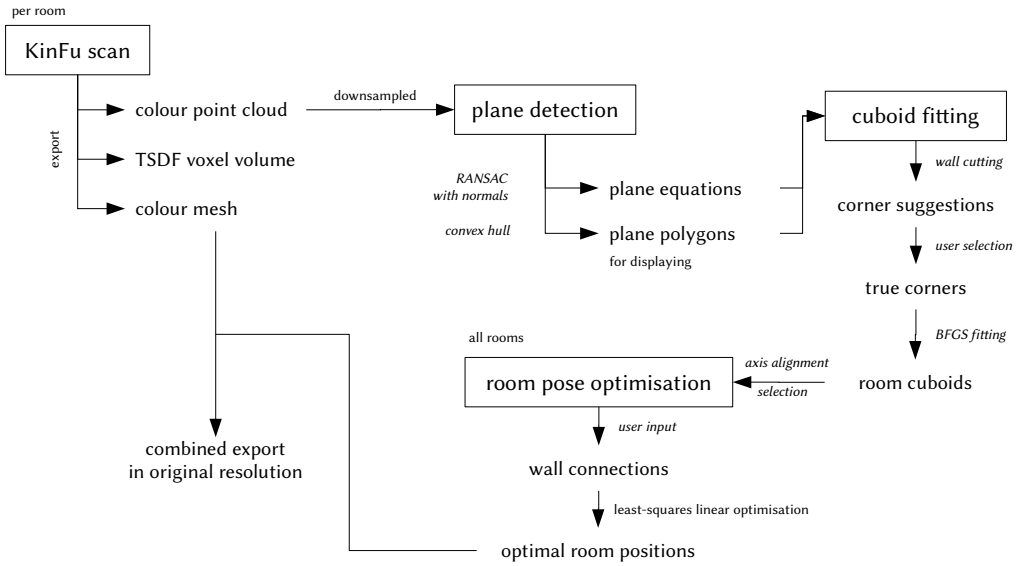

Figure 3.1: Work flow for creating a building reconstruction with *HouseScan*.

HouseScan consists of the following components:

- An improved version of *KinFu* (section 2.3.1.1) to reconstruct and export individual rooms.
- A command line program for plane detection from point clouds with normals.
- A graphical tool for fitting cuboids to room clouds, defining wall constraints and interactive global room pose optimisation.

A future version of *HouseScan* will unify these components in a single user interface. Presently they remain separate programs to be invoked after each other, allowing a simple interchange with other tools fulfilling the corresponding task; of particular interest is the replacement of *KinFu* by an alternative *KinectFusion* implementation, such as *kfusion* (see section 2.3.1.2).

The method for performing a 3D reconstruction (see also Figure 3.1) of a building is as follows:

For each room, the user performs a ***KinFu* reconstruction**, saving created **point cloud** with per-vertex normals and colours, as well as the generated coloured **triangle mesh**. The cubic scanning volume has to be chosen large enough to contain all features of interest, and ideally large parts of floor, ceiling and walls to make their automatic detection in a later step as robust as possible.
Next, a *RANSAC* **plane detection** algorithm is run on a downsampled version of the point cloud to detect the n most prominent planes, where n is typically chosen between 6 and 15, and increased by the user in case only small parts of a wall are present in the 3D scan of a room. All points belonging to a plane are projected onto it and a convex hull is calculated to form a sensible *plane boundary*, which together with the *plane equation* is used for interactive **cuboid fitting**.
In this step, the plane boundaries are displayed as semi-transparent polygons in a 3D application, together with the point cloud model of the room. The user can navigate in the 3D environment to inspect which planes represent the walls of the room, and then picks the 8 corner points that define the *room cuboid* – the cuboid on which the principal walls[9] of the room lie. The choice of corner points is guided by suggestions that are created by intersecting the detected planes. Since the corner points are generated from noisy data, a perfect cuboid is fit to them using the *BFGS* algorithm for unconstrained nonlinear optimisation.
Subsequently all room cuboids are aligned to the coordinate axes, allowing the user to choose one of 4 possible rotations confined to 90° steps; the bottom plane is automatically guessed from the orientation of the camera when the scan started.
After all rooms have been prepared this way, the user connects walls of different rooms for **room pose optimisation**. Each connection describes either that the wall planes are *opposite* of each other (with the two planes being the two corresponding sides of the wall between the rooms), or that they belong to the *same* side of the same wall (facing into identical directions).
These room connections form a *constraint graph* over the possible correct positions of the rooms. The constraint graph is transformed into an *inconsistent system of linear equations*, which is then solved with *least-squares* error, yielding the final optimal position for each room.
The user can either construct the full *constraint graph* at once or work incrementally, performing the optimisation after each or multiple steps to obtain direct feed-back on what effect the created constraints have; the point clouds of the rooms move in the 3D user interface when the optimisation is applied. At all times the constraint graph is displayed visually as a network of lines between rooms, and graphically distinguishes between the two different types of wall constraints.
Once the user is content with the results, the transformation that brings each room into the desired pose is applied to the full resolution coloured mesh models saved from the *KinectFusion* scan, thus creating the final **exported model**.

The following sections describe the steps of scanning single rooms, plane detection, room fitting and room pose optimisation in detail.

For an example of how *HouseScan* works in practice, see chapter 3.6.

3.2 Data acquisition: Scanning single rooms

The foundation of *HouseScan* is a *KinectFusion* implementation; the choice for this project is *KinFu* (2.3.1.1).

KinFu performs 6-DoF SLAM (2.2.3) on depth images while updating a static, fixed size voxel cube to reconstruct the scanned scene. It was the first open-source implementation of *KinectFusion* and is part of the *Pointclouds* library, implemented in C++ and CUDA.

[9] The principal walls are those that shall be used in defining wall constraints between rooms. In most cases, they are the outermost walls of a room. For exceptions, see the examples of attic floors in section 3.6.2.

Alternatives considered were *kfusion* (2.3.1.2) and *KinFu Large Scale*.

kfusion, whilst featuring more robust tracking[10] and being a much smaller and easier to work with, had to be discarded because it can only do reconstruction and does not feature any extraction, neither for point clouds nor for meshes (see sections ?? and ?? for how they are implemented in *KinFu*); both are essential for *HouseScan*. While it is certainly possible and desirable to add them to *kfusion*, it could hardly be done in the time allocated to this thesis.

KinFu Large Scale (2.3.3), being an extension of *KinFu*, does have the required features, and could theoretically deliver superior model quality since its voxel cube can move and consequently contain a smaller region of the real world, recording more voxels per m^3. Unfortunately, its operation is quite unreliable: When the cube is moved, scanning is interrupted for up to multiple seconds (seemingly depending on how many times the cube moved already), thus often leading to track loss.

Kintinuous [2] (see also 2.3.3) could not be considered, since its authors did not make their implementation available. Whilst the videos published about it leave little question that *Kintinuous* can deliver improvements over *KinFu*, its unavailability renders it useless for other researchers.

To obtain good results with either *KinectFusion* implementation, it has to be told the correct *calibration intrinsics* of the used camera. This means setting the correct *focal distance* and *image centre*. Especially the focal distance affects the calculations of distances, and so has an immediate effect on the error of the produced model.

RGB-D cameras based on the *Primesense* chip contain factory-calibrated values for these parameters for the *depth* camera in their ROM, which can be queried using *OpenNI*. *KinFu* originally had hardcoded depth camera intrinsics, which I first changed to read the correct values off the camera[11]. However, I later discovered that when Depth-to-RGB registration[12] is enabled, only the RGB camera intrinsics matter, for which no ROM values are available. Thus I performed a standard RGB calibration with a checkerboard paper and OpenCV. Code for this calibration[13] is available at http://gist.github.com/nh2/f50147b. The results found by calibration replace *KinFu*'s `KINFU_DEFAULT_RGB_FOCAL_*` values.

After the camera has been calibrated, a *KinFu* reconstruction of each room is performed. Since a fixed size, static voxel volume is used, the user has to choose the appropriate cube edge length in meters. This can be independently per room, allowing a better resolution for smaller rooms; *HouseScan* does not require a uniform voxel size across rooms. The size restriction is also a simple way to ensure that parts of adjacent rooms are not included in the current room model, which otherwise might later result in unaesthetic mesh overlaps in the combined model.

By default the user begins a scan in the middle of a room, and the starting position is taken to be in the centre of the scanning volume. For cases in which this is inconvenient, I have added a `--start-at-side` flag to *KinFu*, that sets the starting position to a side surface of the scanning cube.

The user then moves the camera through the room, capturing all as many 3D features as possible or desired. The reconstructed scene is shown in real-time on the screen, giving immediate feedback on what has been scanned so far. For *HouseScan*'s plane detection in a later step, it is recommended that the user tries to capture sufficient parts of the ground, ceiling, and walls of the room.

[10] For mostly unknown reasons[^kfusionunknownreasons]. I discovered why *kfusion* can *detect* track loss much better than *KinFu* as described in section 2.3.1.1, but that doesn't explain why it loses track easier in the first place.

[11] The code for this was inspired by its equivalent in the RGBDemo program: http://github.com/rgbdemo/nestk/blob/b87caaa918/ntk/camera/openni2_grabber.cpp#L568

[12] *Registration*: A camera firmware setting that scales and aligns the depth image to overlay with the RGB image. It is necessary because the depth and RGB sensors in RGB-D cameras do not usually have completely identical fields of view.

[13] Generously provided by Patrick Chilton.

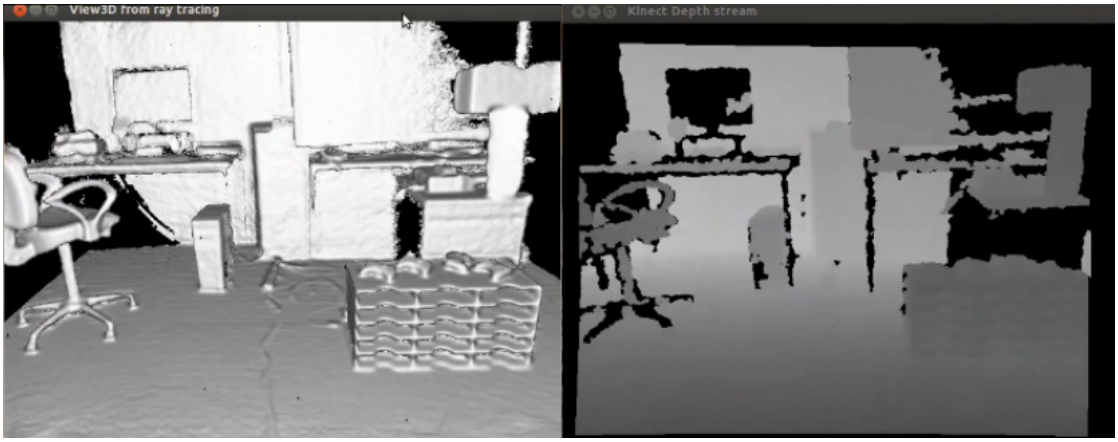

Figure 3.2: A *KinFu* reconstruction of a room, showing the raycast model (left) and camera depth image (right). Source: [36]

When performing a scan, it is important to remember that the RGB camera is only used for giving colours to the 3D model. Tracking in *KinFu* works solely on the depth images, and is thus prone to failure when pointing the camera at scenes containing few 3D features (this problem is discussed in section 2.3.2). Consequently, a room part whose geometry may seem obvious to a human can cause problems in the scan since the current form tracking cannot benefit from colour and shading information. *KinFu* displays the current depth image to help the user detect this. It happens primarily when the camera can only see a single wall or an edge between two walls, cases in which the ICP algorithm (2.4.1) fails and a position jumping or "sliding effect" can be observed, usually resulting in a corruption of the created model. In some situations the system can recover from this: When the user observes sliding and the camera is moved back the path it came into the problematic position, ICP usually fails "the inverse way", and scanning can be continued via other paths without major corruptions. One can also make use of *KinectFusion*'s self-healing properties, in that it will adjust the model to match the camera depth image over a time window of configurable length. In this case, the user can observe how new objects gradually build up in front of the corrupted scene, and after some time the view will be consistent. Still, it is possible that the old, now invalid geometry is simply hidden from view, persist as "phantom features", and will later complicate wall detection and cuboid fitting.

Generally, it is advised to try and avoid these problems from the beginning to save time and be confident that the there are no leftover corruptions. It follows a list of *scanning tips* that have worked well in most situations encountered when performing scans for this thesis:

- **Start in the middle** of the room, facing at the wall with the least interesting geometry. When the scan is completed after a 360° turn, there will be a "vertical cut"-like inconsistency because minimal tracking error accumulate and *KinFu* does not perform loop closure (2.2.4). Having this cut in the wall with least interesting geometry is less visually disturbing to the human eye than having it tear apart a detailed object. This means that we usually start facing a wall with a door in it.
- Perform a **dry run** before the actual scan to estimate how big the scanning cube should be and to explore which parts of the room are especially prone to track loss. Check if moving more slowly makes tracking robust enough in these areas, or if you should avoid them completely.
- As a general scanning strategy, carefully **tilt the camera upwards and downwards** while slowly

turning left (or right) around your own axis. This seems to give superior results than, say, rotating around the room twice, capturing lower parts in the first turn and higher parts in the second.
- Avoid **danger zones**. Single walls and edges are especially problematic – always make sure to have a wall lamp or parts of a table visible in the depth image when pointing at them. Corners work well, since the 3 planes around them constrain all degrees of freedom in the ICP algorithm.
- Avoid scanning objects from **large distances** if you care about their quality in the output mesh. The further you are away from an object, the less pixels will be allocated to it in the depth and colour images, making the result look coarse. In addition, depth values farther than 6 m reported by the depth camera are not very accurate.
- Do not forget to include **floor and ceiling** in the scan. To not lose track on the ceiling plane, try to connect from high cupboards, lamps or door frames.
- **Incisions** between tiles can be a boon for tracking in bathrooms and kitchens.
- Scanning **under tables*, chairs etc. is usually unproblematic due to the detailed geometry.
- A **helper** carrying the scanning device can make sure you always have an eye on the reconstructed scene and the depth image.
- 3D **video glasses** or a **heads-up display** can replace a helper.

Equipped with these tips, it should be possible to record most rooms within 5 minutes.

At the end of the scan, the user saves

- the coloured point cloud extracted from the voxel volume (see ??)
- the coloured triangle mesh extracted from the voxel volume (see ??)
- the voxel volume itself and the colour volume

to disk using the respective *KinFu* shortcuts.

KinFu could originally only export an uncoloured mesh. I extended the *CUDA* code in its GPU implementation of the *Marching Cubes* algorithm to include per-vertex colours, which, like the triangle vertices, have to be correctly interpolated from the voxel grid.

Having per-vertex colours instead of one colour for each triangle allows for the mesh to be displayed with colour interpolation similar to *Gouraud Shading* [37], giving smooth colours instead of visible triangles.
Note that for this to be displayed correctly, shading and lighting must be disabled in the viewing program – otherwise triangles will reappear. This is expected because the triangles are already correctly lit, using not realistic lighting but *real* lighting!

When creating models of people instead of rooms[14], I noticed that there might be a bug in *KinFu*'s colour volume updating code: When the scan finished facing a person frontally, the sides of the head carried white stripe-like miscolouring artifacts. The uncoloured geometry in those places remained correct. These artifacts did not appear in room models.

> **A note about camera settings.** I discovered that Primesense based cameras, when using either OpenNI 1 or 2, feature vertical stripes artifacts in the depth image, independent from the camera orientation.
> While they are unnoticeable when overlaying complex geometry, their effect is strong

[14] As a form of creative break activity.

when the camera is pointed at planes, potentially worsening already weak tracking robustness that *KinFu* has on plain geometry.
It turns out that they are caused by the `GMCMode` setting in the OpenNI configuration. Setting `GMCMode=0` removes the stripes and might improve the scan quality. It is not known as of now what other effects this setting has. The issue is discussed at [38].
All scans in this thesis were performed with the default `GMCMode=1`.

I added a range of further improvements for single-room scanning:

- I made **recordings reproducible** by adding a *simultaneous recording* functionality to both *KinFu* and *kfusion*, allowing them to record the raw depth and RGB video frames while performing the *KinectFusion* implementation. Since this was implemented efficiently as an OpenNI *Recorder*, it incurred no noticeable performance penalty on the frame rate.

  KinFu already had a facility to run a scan from a recorded *.oni* file instead of a camera, and I added the same functionality to *kfusion*. This way one can perform a room scan in either of the programs and then compare the quality of its reconstruction with the other one.

  In addition, *KinFu* features a *triggered* mode for such replays, in which each recorded camera frame is used. By contrast, in normal replays or live reconstructions frames are skipped when they arrive faster than *KinFu* can process them. *Triggered* mode makes reconstruction completely deterministic.

  It may be useful to note that at the time of writing, *KinFu* uses OpenNI 1 while *kfusion* uses OpenNI 2. The *.oni* files recorded with these versions are compatible with each other, but *only* when colour compression is disabled (using `XN_CODEC_NULL`).

- *KinFu* could already export the current TSDF volume, but not make any further use of it. I added a feature to allow **loading saved TSDF and colour volumes**, which allowed testing different point and triangle extractions on exactly the same voxel grid.

- This also allowed me to find the source of a problem in *KinFu*'s *Marching Cubes* implementation. Being an **invalid memory access** in the GPU code, it regularly crashed the program. Unfortunately, it only manifested when computing meshes from large scanned areas, making it time consuming to reproduce. Having added the video replaying feature did not help much for this, since the problem would only appear after replaying past the 10 minute mark to obtain enough filled voxels to trigger the crash. Because the `cuda-memcheck` tool slows down *KinFu*'s execution by approximately factor 30, repeatedly obtaining detailed information with it was impossible. The above volume voxel loading feature made it possible to save a volume on which triangle extraction would crash, and skip all reconstruction work for the execution with `cuda-memcheck`. This revealed that the size of the buffer for holding the extracted triangles was hardcoded, despite the number of potentially generated triangle being unbounded.

- I added OpenNI support to *kfusion* [39], thus enabling it to run with a variety of RGB-D cameras, including my *Asus Xtion Pro*. Before it only worked with a Kinect.

After each room has been scanned, the user proceeds into fitting cuboids to them.

3.3 Plane detection

After a point cloud of each room has been obtained, we need to detect planes in order to facilitate finding the room corners.

Section 2.5 presented algorithms for this task: Region Growing, Hough Transform and RANSAC.

HouseScan uses RANSAC because it is the simplest of the three and already works well enough.

Unlike the Hough Transform, it does not work in a transformed parameter space, which makes it easier to visualise and debug.

An unmodified Region Growing approach is not always practical for wall detection because walls are often occluded by other objects in the room. As a result, what we would like to detect as one wall is split into two separate regions which the Nearest Neighbour Search usually employed in Region Growing does connect, and the wall is detected as two independent planes that would have to be merged in a post-processing step. This is even more apparent for a wall that contains an architectural interruption like a flue or an outset shower (see section 3.6.2 for examples).

RANSAC's main drawback against the other alternatives is that, being a randomised algorithm, it is not guaranteed to find the most prominent planes, nor is it deterministic.

In an initial version of *HouseScan* I used the most basic version of RANSAC, working on points that only had (X,Y,Z) coordinates. While the results were useful, they contained some stray planes that were contributed by objects in the room. As an example, the frame of a bed together with the lower side of a chair could create enough voting consensus to have a plane detected through them.

I escaped these problems by switching to a *RANSAC with normals*, which punishes the voting weight of points whose surface normal does not align with the plane normal. In PCL's `SampleConsensusModelFromNormals`, the weight of normal agreement is set with the `normalDistanceWeight` parameter, whose default value *0.1* has given good results for *HouseScan*.

Before performing the actual RANSAC plane detection, the room point cloud is downsampled using a uniform voxel grid. Each point from the original point cloud falls into one of the voxels, and the mean of all points in the voxel is used to create a new point in the downsampled point cloud. A leaf size of 10 cm is the default for *HouseScan*, as it both makes the RANSAC step fast, reduces noise, and prunes the abundance of points sufficiently so that each remaining point is meaningful when displayed in the cuboid fitting user interface.

As described in section 2.5.3, RANSAC provides us with the n most prominent planes. Values of n between 6 and 15 are useful for *HouseScan*. Since walls tend to be occluded, finding the top 6 planes is often not enough and faces of cupboards or tables will have more voting points. Increasing the planes recognised with RANSAC does not come with a penalty, since *HouseScan*'s graphical user interface allows for an easy removal of those that are not the true plane walls.

For each recognised plane, the points that voted for it are projected onto it and a convex hull is computed, called the *plane boundary*. This is used to display clickable 3D polygons for the planes in the next step.

The provided C++ command-line program written using PCL saves the downsampled cloud, the plane equations and plane boundaries of the room to disk, ready to be used for *room fitting*.

3.4 Room fitting

Room fitting and *room pose optimisation* take place in a 3D GUI written in Haskell.

The user imports each room to be processed from its containing directory, which automatically loads and displays the downsampled coloured point cloud and detected planes.

The objective is now to find the 8 corners of the room, so that a perfect cuboid can be fit to it.

The faces of the cuboid shall be called the *principal walls* of the room. These are the walls that the user can connect to other rooms in *room pose optimisation*, and are usually the walls that an architect would draw into a floor plan.

It is important to notice that the cuboid corners need not be present in the point cloud, for example when they were not included in the scan or occluded by furniture.

Sometimes they do not even exist in the real room, like for an attic room under a slanted roof. It will have *virtual corners* outside of the room that would complete it to a cuboid if the roof was not slanted.

The idea is that these corners can be accurately recovered by cutting the RANSAC-detected principal walls.

The user can therefore select 3 planes and cut them to obtain a corner, which is immediately displayed.

To reduce the amount of clicking, *HouseScan* cuts all available planes with each other to display suggestions, which the user can pick with a single click. This will naturally create corners that are not good suggestions, like far-off points when two of the planes are almost parallel (like floor and ceiling). To not clutter the user interface these far-off suggestions, they are filtered to exclude points whose distance to the room cloud mean is greater than 1.2 times maximum distance of any cloud point from that mean.

A natural question is why we do not simply pick 6 wall planes, which automatically define 8 corners. The reason is that because *KinFu* does not perform loop closure, it cannot create self-consistent models. When a scan is started at one half of a wall and finished at its other half after a 360° turn, the two wall segments reconstructed will not perfectly align with each other, and angles up to 30° can appear in practice.
This leaves us with two planes which should, in reality, be the same wall. Picking either of them as a *principal wall* will create corner points far off the actually visible positions of the corners. By defining the cuboid corners instead of its faces, we eschew this problem, and our two inconsistent plane segments can each define two corners with their adjacent walls.
While a loop closure detection followed by a model correction would be a better solution to this problem, it is out of scope for this project, and to my knowledge no research has been published that implements loop closure inside of *KinectFusion*'s voxel grid. Kintinuous [2] does perform loop closure optimisation, but it works on a mesh streamed out of the moving scan volume and does not correct the contents of the voxel grid itself. It appears to me that a loop closure correction inside the scanning volume is a difficult, unsolved problem, and that the voxel grid's continuous, rigid density makes it particularly inaccessible to global corrections.

Once 8 room corners have been defined and the user is content with their positions, a perfect cuboid is fit to them.

This is done using the Broyden-Fletcher-Goldfarb-Shanno algorithm (BFGS, [40], p. 55) for multi-dimensional, unconstrained, nonlinear optimisation. It is an iterative algorithm that, given a cost function $f : \mathbb{R}^n \mapsto \mathbb{R}$ of n parameters and an initial guess, tries to find a global optimum using a hill-climbing procedure inspired by Newton's method. The implementation is provided by the `multimin` function of the GSL library [41], and used from Haskell via the `hmatrix` package [42].

The cost function of choice is the square distance of all 8 corners to their closest corner on the estimated cuboid:

$$f(C_{\mathrm{xyz}}, a, b, c, Q) = \sum_{i=1}^{8} \| \mathrm{ce}_i - c_i \|^2$$

$$\text{where} \qquad \text{ce}_i = \arg\min_{e \in E} \|e - c_i\| \qquad \text{and}$$

$$E = \{\ C_{\text{xyz}} + R_Q v \mid v \in C_\perp \} \quad \text{with} \quad C_\perp = \left\{ \begin{array}{lll} \left(-\frac{a}{2}, & -\frac{b}{2}, & -\frac{c}{2}\right)^T \\ \left(-\frac{a}{2}, & -\frac{b}{2}, & +\frac{c}{2}\right)^T \\ \left(-\frac{a}{2}, & +\frac{b}{2}, & -\frac{c}{2}\right)^T \\ \left(-\frac{a}{2}, & +\frac{b}{2}, & +\frac{c}{2}\right)^T \\ \left(+\frac{a}{2}, & -\frac{b}{2}, & -\frac{c}{2}\right)^T \\ \left(+\frac{a}{2}, & -\frac{b}{2}, & +\frac{c}{2}\right)^T \\ \left(+\frac{a}{2}, & +\frac{b}{2}, & -\frac{c}{2}\right)^T \\ \left(+\frac{a}{2}, & +\frac{b}{2}, & +\frac{c}{2}\right)^T \end{array} \right\}$$

Here, C_{xyz} is the centre of the estimated cuboid, Q is a unit quaternion that describes the rotation of the estimated cuboid, and a, b and c are the three side lengths of the cuboid. The c_i are the corners chosen by the user, ce_i is the closest estimated corner for each user corner, E is the set of corners of the rotated estimated cuboid, R_Q is a rotation matrix that expresses the same rotation as Q, and $C_\perp$ are the corners of an unrotated, axis-aligned cuboid centred at the origin with side lengths a, b and c.

So we create a cuboid at C_{xyz} that can rotate around, its centre and change its side lengths. We choose the square distance from each user-defined corner to the respective closest corner of the cuboid as the cost function f, a function of 10 floating-point parameters. We then let BFGS minimise it for us, yielding the optimal centre, rotation and side lengths of the cuboid.

The initial guess for the parameters is:

- The mean of the room point cloud for C_{xyz}.
- $(0.1, 0.1, 0.1, 0.1)^T$ for the rotation quaternion Q.
- A guess for the cuboid dimensions a, b and c as follows:

  Let f be the first of the 8 corners supplied by the user.
  Compute the distances of all other corners to f.

  Pick a and b as the two smallest distances, and let d be the largest. Pick c as $\sqrt{d^2 - a^2 - b^2}$.

  The corresponding Haskell code:

```haskell
guessDims :: [Vec3] -> (Double, Double, Double)
guessDims p = (a, b, c)
  where
    f:rest = p
    [a,b,_,_,_,_,_,d] = sort $ map (distance f) rest
    c = sqrt (d*d - a*a - b*b)
```

To justify the use of BFGS, it should be noted that our cost function is indeed unconstrained, not linear, and smooth except for when a jump in the nearest corner association happens.
Like with all hill climbing methods running on non-convex functions, it is possible to get stuck in a

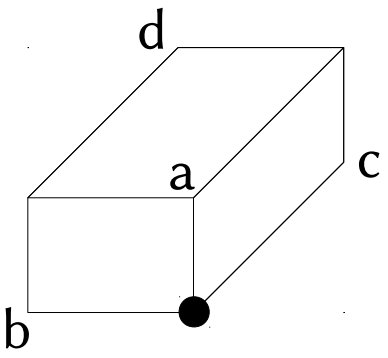

Figure 3.3: Cuboid dimension guessing. a and b are the distances to the two closest corners, d is the distances to the farthest corner.

local extremum. Luckily, the initial guess we can make is quite accurate, and in practice fitting cuboids this way works very well.[15]

The nearest-corner-lookup is an admission that we cannot assign each user selected corner with a cuboid corner in an obvious fashion.
It may be interesting to note that if we knew the correct corner assignment, fitting the cuboid would reduce to a point-to-point ICP where we keep the middle matrix of the SVD to allow scaling the cuboid (see section 2.4.2).

Before settling on the described form of optimisation, I tried a number of variants.
My initial version was a cost function of 9 parameters, expressing the position and orientation of the cuboid as a 4×4 homogeneous rotation/translation matrix with 6 degrees of freedom (3 angles + 3 offsets). I switched to the Quaternion to make the rotation smoother, as is often done in Computer Graphics when trying to linearly interpolate rotations.
I also originally expressed the cuboid orientation as a rotation around the origin, which unsurprisingly magnified the effect of even the smallest changes in rotations if the cuboid was far away from the origin. Switching to a rotation around the centre made it "slide into place" more naturally, and improved convergence.
Since most BFGS implementations make the iterative steps, the *solution path*, available to the programmer, their effects on cuboids can be nicely visualised, and I made extensive use of this while working on cuboid fitting.

In my experiments it seemed that it is possible to achieve slightly better results by first fixing C_{xyz} to the mean of the point cloud, running BFGS to convergence (thus optimising rotation and side lengths for that fixed centre), and then doing another BFGS run with all parameters free. The difference was small though, and more investigation is necessary to decide if this really is a worthwhile improvement.

It initially surprised me that I could not find a closed form solution for fitting a cuboid to 8 corner points.
I discovered various papers on how to enclose a point cloud with a sensible cuboid, but none that

[15] The *true* motivation for doing this was of course to get some experience on how to solve problems using generic numerical optimisation algorithms. Throwing the BFGS sledgehammer at 8 corner points is an act of compelling brutality. That said, I have not encountered a better method so far.

interprets points as unordered, noisy corner points.
By now, I conjecture that a closed form solution is only possible if the point correspondences are known.

With each room being fit to a cuboid, we can proceed to the main contribution: Arranging them to a full model.

3.5 Globally consistent room pose optimisation

Equipped with cuboidal rooms, we now set out to assemble them into a combined model.

To do this, *HouseScan* allows the user to enter information about the relative positioning of the rooms in the form of *wall constraints*.

They are informed by architecture, such as the user knowing that certain rooms are adjacent or on top of each other. As an example, the information that "this wall from room *A* and that wall from room *B* are opposite sides of a wall in the building" constrains the positions of room *A* and *B*, removing one degree of freedom. No matter where they end up in the final model, two of their walls must snap together.
If we can provide enough constraints, each room will have lost any freedom of movement, and the position of each room in the final model is determined.

A system like this allows entering contradictory constraints. Consider models of two adjacent, cubic rooms of 5 m length each, with a wall between them that was measured to be 20 cm thick. The distance between their outer walls should consequently be 10.20 m. If however we measure the actual distance to be 10.25 m, our two measurements are conflicting and we have to find a compromise. In this case, your set of constraints is said to be *inconsistent*.

A commonly employed compromise is the *least-squares* solution, in which the deviations of all constraints V from their actual, measured values are squared and summed up:

$$err = \sum_{v \in V} ||v - v_{\text{compromise}}||^2$$

We have to find a set of compromises such that err is minimised, thus achieving *global consistency*.

HouseScan does not restrict the constraints to be about rooms that are next to each other. If for example a room at the side of a building has the same outer wall as a room on top of it four levels higher up, a constraint across multiple levels can be added.

Currently, two types of constraints are supported:

- *"Opposite"* constraint: Two wall planes are opposite of each other, with the two planes being the corresponding sides of the wall between rooms. This constraint also carries with it a given distance between the two rooms, the *wall thickness*.
- *"Same"* constraint: Two planes belong to the same side of the same wall, facing in the same direction, with the plane-to-plane distance being 0.

These two types of constraints are enough[16] to create models of most buildings, thanks to the simple geometry that they possess and which *HouseScan* exploits in various places.
While this works well in most cases, it fails with non-standard architecture such as round rooms, rooms

[16] We will soon see that `Same` is actually a special case of `Opposite`, but it is helpful to distinguish them at least in the user interface.

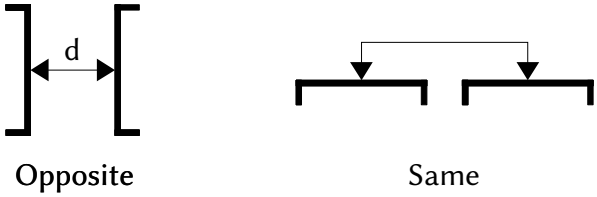

Figure 3.4: Illustration of `Opposite` (left) and `Same` (right) constraints. `Opposite` constraints have a *wall thickness* attached.

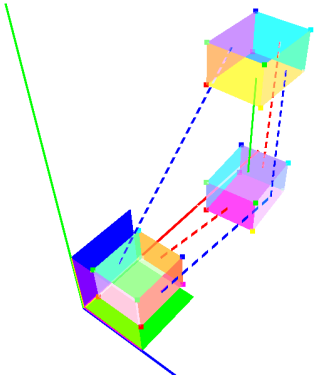

Figure 3.5: Wall constraints between rooms. The colour indicates the axis on which the constraint is active. The stroke style distinguishes the constraint type: solid for 'Same', dashed for 'Opposite'.

without well defined boundaries, rooms too big to contain in a single 3D model, caves, and buildings designed by Gaudí, since an underlying rectangular floor plan is assumed.

Since we have restricted the rooms to be cuboids that have to live in a *Manhattan* world, the user next chooses one of the four possible orientations for each room. The floor plane is guessed automatically, since *KinFu* scans, while allowing a completely unrestricted camera movement, are usually started with the camera being held upwards.

In *HouseScan*'s 3D user interface, wall constraints are created by selecting two walls and creating either a *Same* constraint or an *Opposite* constraint with accompanying thickness. A constraint is active on one of the three coordinate axes. We can thus express it in Haskell as a 4-tuple:

```
type WallConstraint = (Axis, ConstraintType, Wall, Wall)

data Axis = X | Y | Z

data ConstraintType = Same | Opposite { thickness :: Double }
```

All constraints are displayed in the 3D environment as lines between room planes, with the colour indicating the axis on which they are active and the stroke style distinguishing the constraint type.

The entered constraints form an undirected *constraint graph*, where each node is a room and each edge is a constraint on walls of the connected rooms.

Our task is now to perform *room pose optimisation*, calculating the optimal translation (the orientations were already chosen above) for each room from this constraint graph. In case of inconsistencies, we desire a least-squares solution.

This is done by reformulating the constraint graph as an inconsistent (or *overconstrained*) system of linear equations, and solving it using a standard least-squares linear solver.

Beforehand, it is useful to realise that constraints on one axis can only influence the position of rooms *along that axis*, and have no effect on the other axes. For example, knowing that the gap between two rooms on the *x*-axis is 20 cm may cause a translation of the rooms along the *x*-axis, but does not give any information about the *y* and *z* axes[17]. Therefore, the translation optimisation can be done *independently* for all axes, and in arbitrary order.

We now seek to construct one equation system *per axis*.

The first step is to transform the constraints on room walls into constraints on the rooms themselves. It is convenient to use the room centres[18] for this, though any fixed point in a room would do.

Let $c_{A,x}$ and $c_{B,x}$ be the components of axis x of the centres of two rooms A and B. Let w_A be a wall of A that is orthogonal to axis x and placed at $w_{A,x}$ on that axis, and let w_B be a similar wall of B placed at $w_{B,x}$.

Each constraint (x, `Opposite t`, w_A, w_B) expresses a desire that the distance between $c_{A,x}$ and $c_{B,x}$ be

$$d_{AB} = s + \operatorname{sgn}(t) \cdot t \quad \text{with} \quad s = (w_{A,x} - c_{A,x}) + (c_{B,x} - w_{B,x})$$

where sgn is the sign function (1 or -1 depending on the sign of s).

For a constraint (x, `Same`, w_A, w_B), t is simply `0`.

We now accommodate all these desires into a single equation system of the form $Ax = b$.
The system has one row in A and one corresponding entry in b for each constraint.
The vector for which we solve, x, contains the unknown centres $\{c_{A,x},\ c_{B,x},\ \dots\}$, one for each room.
A selects which two centres (i, j) are involved in the constraint. The entry in b is set to the desired distance d_{ij} of the constraint.

The result is an equation system like:

$$\begin{pmatrix} 1 & -1 & 0 & \cdots \\ 1 & 0 & -1 & \cdots \\ 0 & 1 & -1 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} c_{A,x} \\ c_{B,x} \\ c_{C,x} \\ \vdots \end{pmatrix} = \begin{pmatrix} d_{AB} \\ d_{AC} \\ d_{BC} \\ \vdots \end{pmatrix}$$

We can now solve this system using a standard linear least-squares solver.

For *HouseScan* this is `linearSolveLS` from the `hmatrix` package, which calls out to *LAPACK*'s `dgels` routine.

There are two improvements we have to make before this technique works in practice.

[17] This would not be true if we were not in the axis-aligned *Manhattan* world.

[18] With *room centre* I mean the centre of the room cuboid. This is different from the *room point cloud mean* referred to in the cuboid fitting in section 3.4.

First, we have only set relative constraints, but expect the solver to come up with absolute values for the room centres. This will not work, because the equation system is underdetermined: For any value of $c_{A,x}$, we can find a least-squares solutions for the other centres.
To make the solution unique, we have to fix one room to an absolute location in the world space. We do so by setting $c_{A,x} = 0$, dropping it from the x vector and removing the first column from the A matrix. We can see in the remaining system from our example,

$$\begin{pmatrix} -1 & 0 & \cdots \\ 0 & -1 & \cdots \\ 1 & -1 & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} c_{B,x} \\ c_{C,x} \\ \vdots \end{pmatrix} = \begin{pmatrix} d_{AB} \\ d_{AC} \\ d_{BC} \\ \vdots \end{pmatrix}$$

that this immediately forces a value upon $c_{A,x} = 0$, and because each wall constraint can only refer to two walls, transitively leave no centre a degree of freedom.

Since all computed centres are now relative to room A, we should add $c_{A,x}$ to the solutions of the equation system to obtain the new centre of each room in the world space.

The above works except for when the transitive chain starting at w_A does not pervade the whole equation system, which brings us to the second improvement.

If we have 4 rooms that are connected in independent pairs (say [A-B], [C-D]), then the created equation system will look like this:

$$\begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} c_{B,x} \\ c_{C,x} \\ c_{D,x} \end{pmatrix} = \begin{pmatrix} d_{AB} \\ d_{CD} \end{pmatrix}$$

This equation system is *rank-deficient*, leaving a degree of freedom for the pair (C, D).
Our least-squares solver is allowed choose an arbitrary offset for this pair, and in practice it picks values of immense distance from the origin, making the pair vanish from the screen.

How should we deal with this? We could switch our solver to `linearSolveSVD` (`dgelss` in *LAPACK*), and since the SVD is *rank-revealing*, it can detect free values and set them to 0.

This however will place all "free" rooms at the same location, which is undesirable since it makes it difficult for the user to interact with them and keep overview.

A better solution is to perform a *connected component analysis* on the constraint graph and solve the equation system for each graph component independently.
That way, none of the systems can be rank-deficient.

The room pose optimisation needs by no means to be performed on the final constraint graph. Instead, the user can run it *interactively*, after the addition of every consecutively added constraint if so desired, to obtain immediate visual feedback on what effect the constraint has.

> **Note.** It may seem restrictive that *HouseScan*'s pose optimisation only works on translations and uses a *Manhattan world* assumption, and indeed it is. While rectangular geometry is found in most houses and the separation between axes simplifies the work flow and allows the user to work on them independently, aligning planes in arbitrary direction to allow scanning more exotic buildings would be a useful extensions. I discuss this as Future Work (chapter 6), and propose a technique to do it.

We have now calculated a globally consistent, optimal position for each room.

From the time when they were loaded into the program until this step, *HouseScan* maintains a 4×4 homogeneous projection matrix for each room, in which all performed rotations and translations are recorded.

The user can now export this projection matrix to move each full resolution 3D mesh in the *.ply* files saved from the *KinFu* scan to its final position using the `plyxform` tool, and display the final combined model in a 3D viewer of their choice.

```
plyxform -f projection-matrix.xf original.ply > final.ply
```

Programs like *Meshlab* [4] also feature a *ruler* tool, with which distances in the complete model can easily be taken, turning it into the 3D equivalent of a floor plan.

To get an idea of how aligning rooms works in practice, there are multiple screen shots provided in section 3.6.1: Composing rooms. To skip to the finished model consider section 3.6.3 instead.

3.6 A full walk through

This section demonstrates how *HouseScan*'s abilities by combining rooms from the data set introduced in chapter 4 into a full mesh model of a house with 23 rooms.

Chapter 5 contains experiments to measure how accurate *HouseScan*'s reconstruction on this data set is.

3.6.1 Composing rooms

The user begins with a loaded room with recognised planes (Figure 3.6).

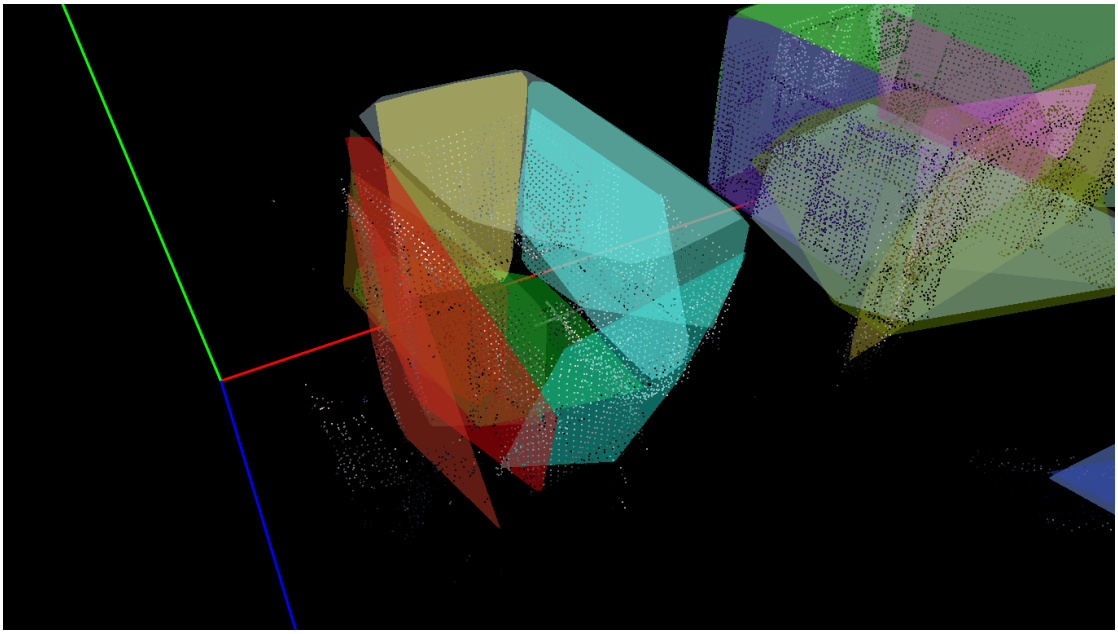

Figure 3.6: Loaded room with detected planes.

Some recognised planes do not belong to walls, but to cupboards and other flat surfaces. The user selects and removes them by pressing `Delete` (Figure 3.7).

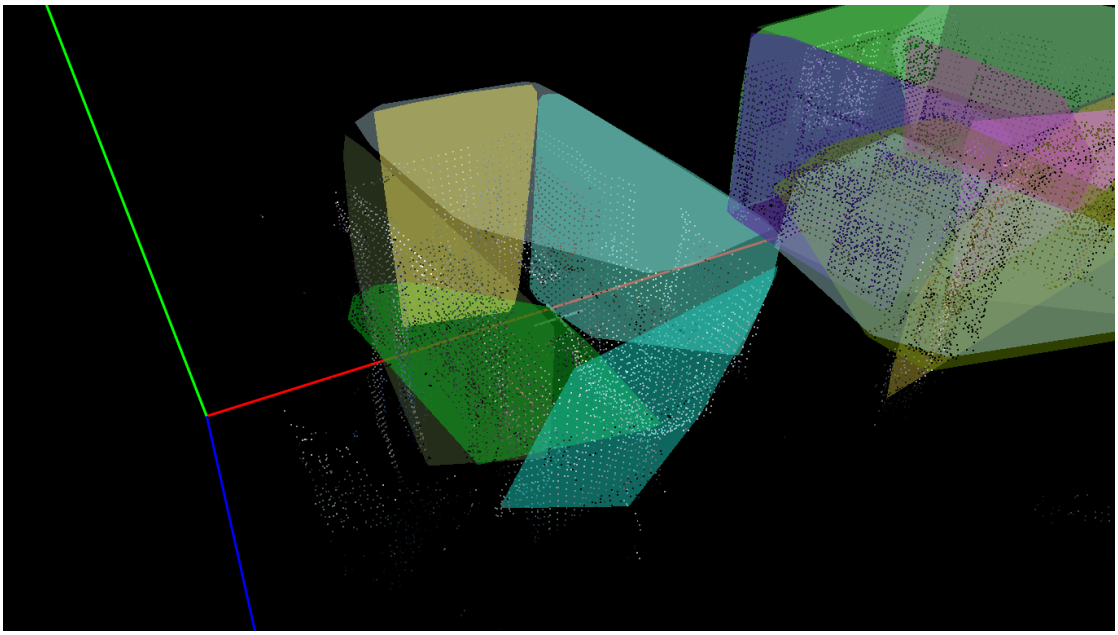

Figure 3.7: Loaded room with non-wall planes removed.

Next, the user presses `g` to display corner suggestions from cutting the available planes, and picks the best 8 corners by clicking on them.
The selected corners are displayed as coloured points (Figure 3.8).

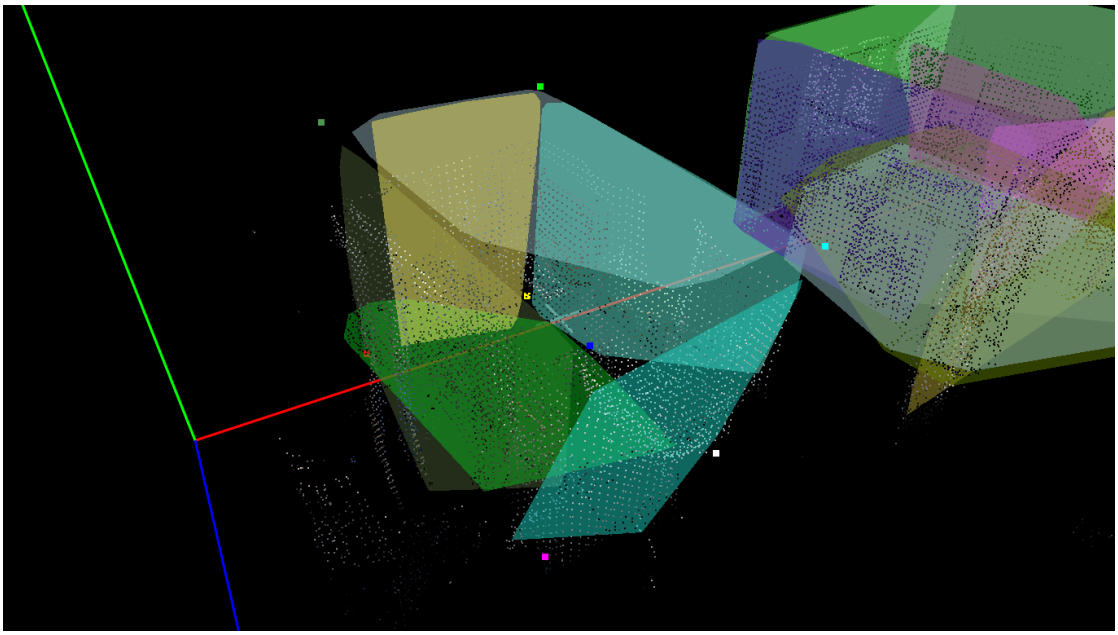

Figure 3.8: Room with selected corners.

Pressing `f` fits a cuboid to the corners using the algorithm described in section 3.4. The result is shown in Figure 3.9.

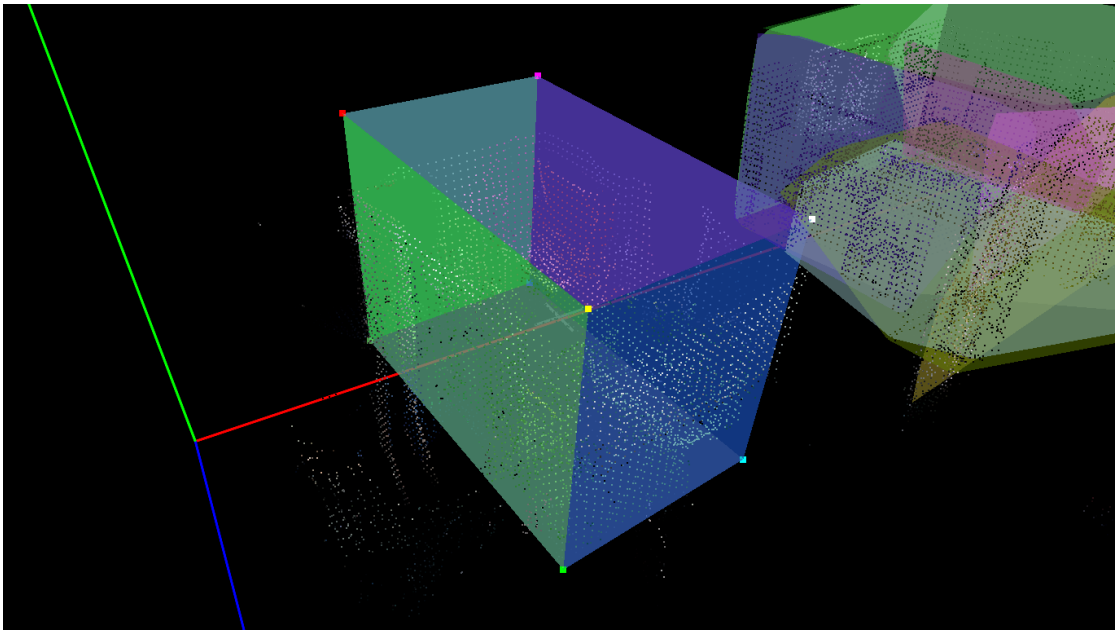

Figure 3.9: A cuboid is fit to the room.

The cuboid walls now need to be aligned to the coordinate axes (Figure 3.10). Pressing `a` cycles through the four possible rotations, while the bottom plane is automatically guessed from the orientation of the camera when the scan started.

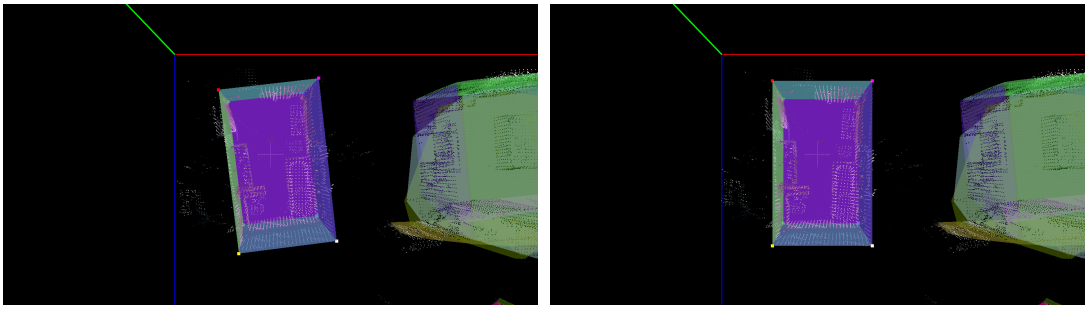

Figure 3.10: Rotating a room, snapping to the axes. Initial orientation (left), final orientation (right).

Cuboid fitting is applied to all rooms in the building (Figure 3.11).

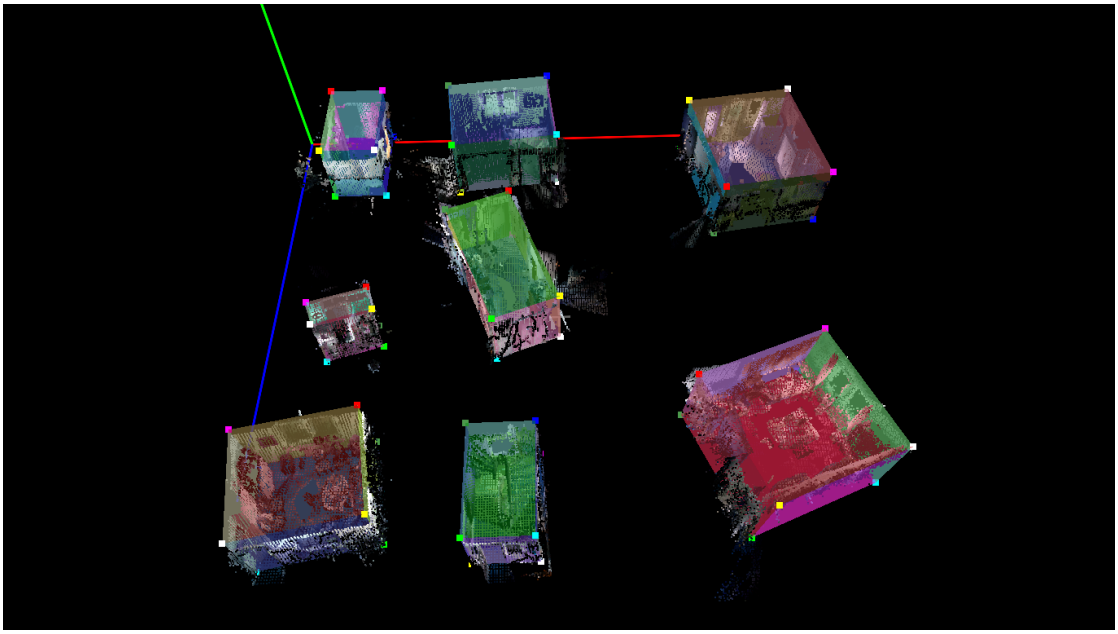

Figure 3.11: Overview showing fit cuboids of one building level.

Now it is time to define wall constraints for global room pose optimisation as described in section 3.5. Walls are connected using the `Opposite` and `Same` constraints, creating a visible *constraint graph* over the rooms (Figure 3.12).
In this graph, the colour of the constraint edge indicates the axis on which the constraint is active, and the stroke style distinguishes the constraint types (solid for `Opposite`, dashed for `Same`).

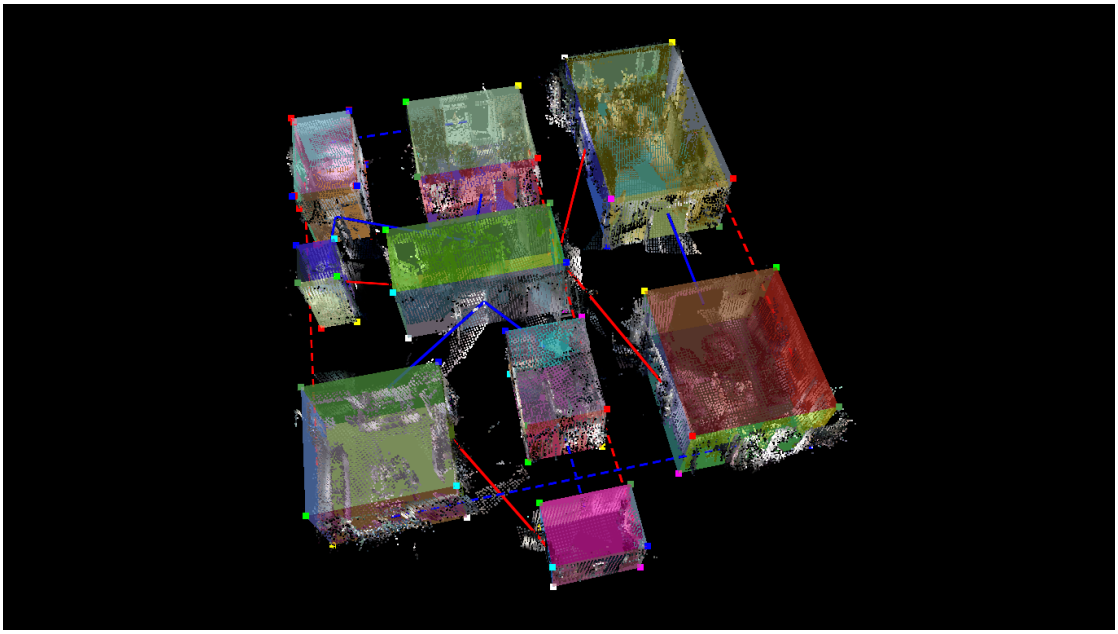

Figure 3.12: Rooms connected by wall constraints.

Pressing the o key performs the optimisation, and the rooms snap together (see Figure 3.13).

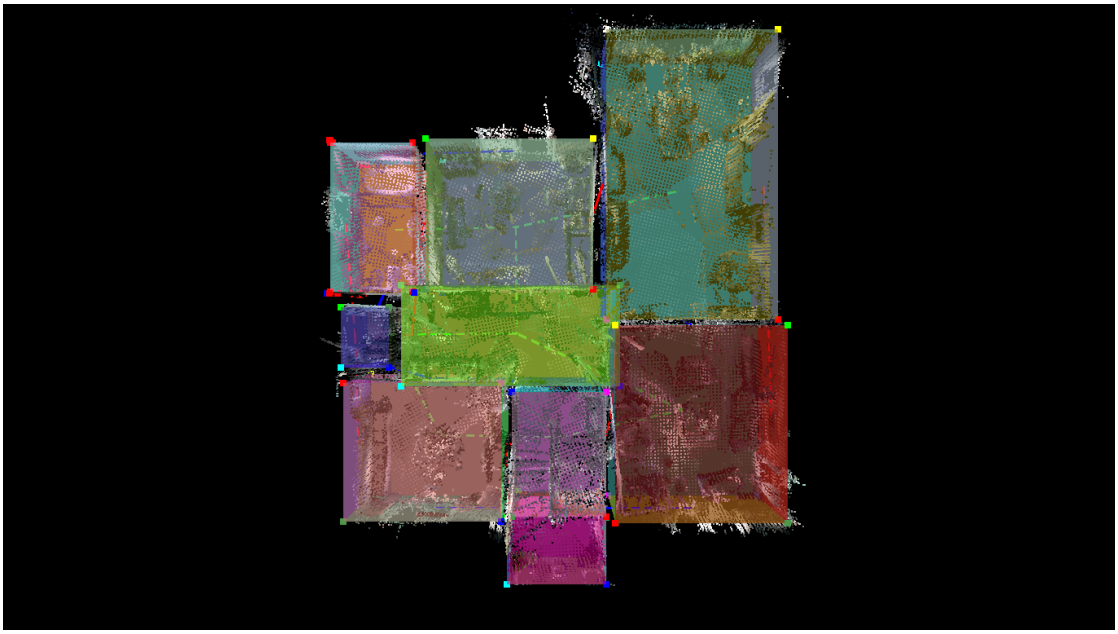

Figure 3.13: Globally consistent room pose optimisation pulls the rooms together.

At any point, planes can be disabled to inspect the quality of the alignment (Figure 3.14). The ceiling

points are hidden automatically to allow an unobstructed view into the rooms.

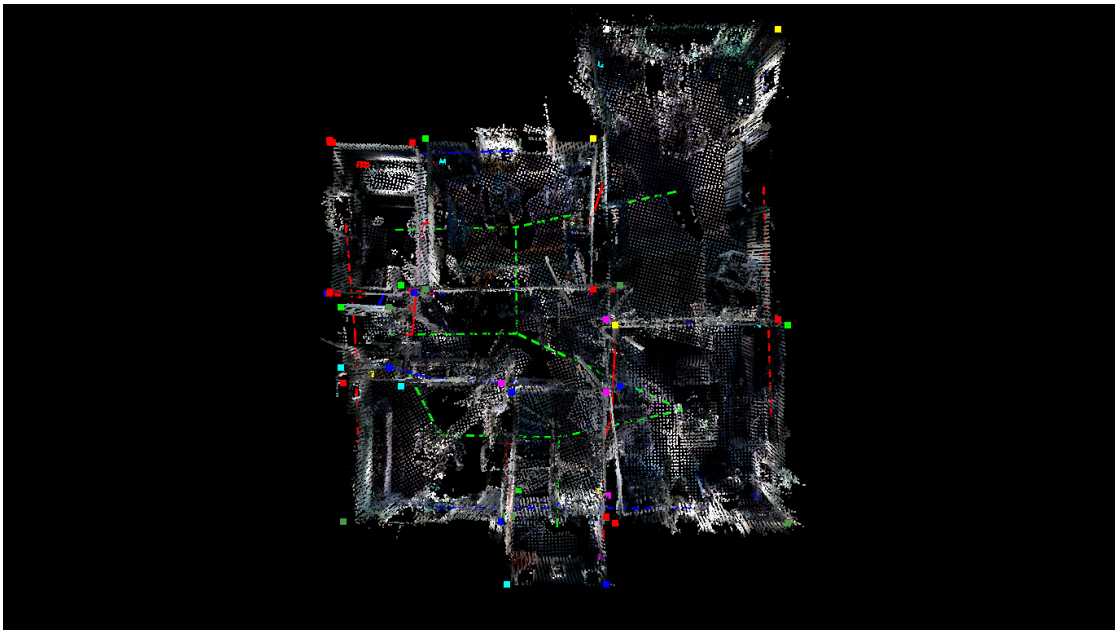

Figure 3.14: Disabling planes reveals the quality of the alignment.

A recommended way to work with multi-storey buildings is to align them level by level. Afterwards, a vertical constraint will place the levels on top of each other (Figure 3.15), and connecting the outer walls will yield the correct multi-level alignment (Figure 3.16).

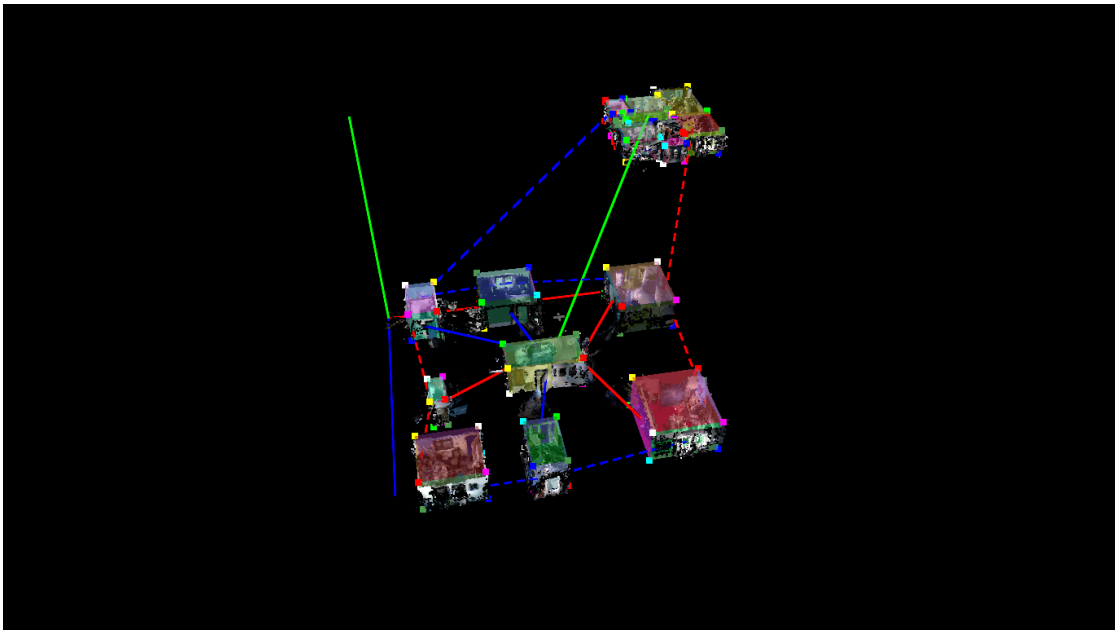

Figure 3.15: Two levels are being connected by a vertical ‘Opposite‘ constraint (green) and two outer wall ‘Same‘ constraints (blue, red).

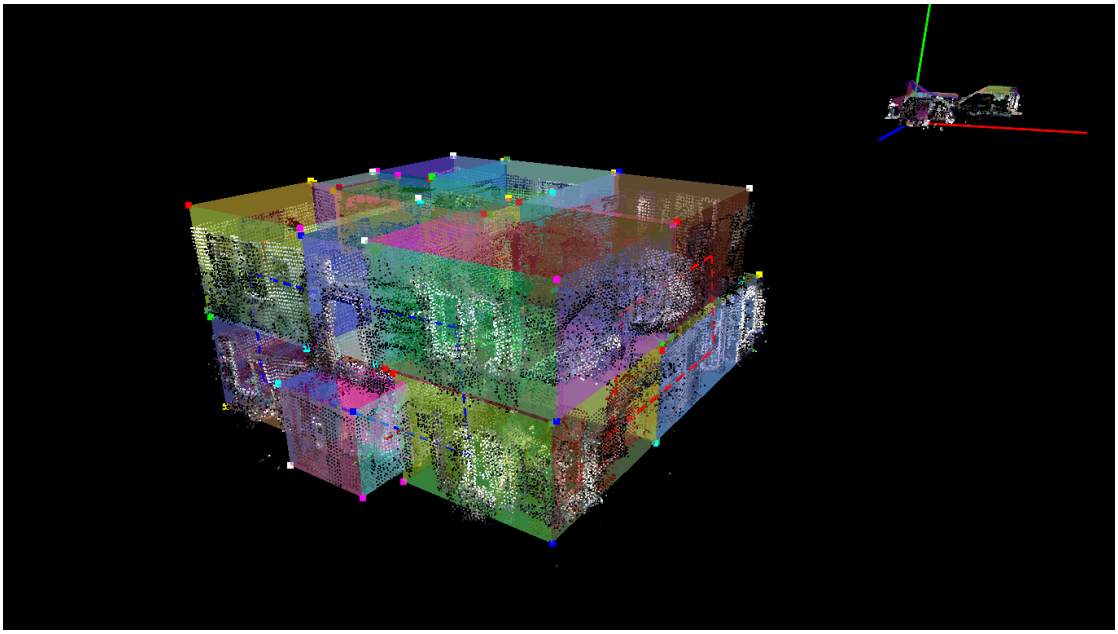

Figure 3.16: Two levels of a house are correctly aligned, while a third level is being prepared in the background.

The final arrangement of all 23 rooms in the completed house is shown in Figure 3.17. It demonstrates

how even the small offset between the outer walls of the middle and the slightly smaller topmost floor is correctly recovered. This was achieved by setting cross-storey constraints only where they appear in reality: in the staircase, instead of connecting all outer walls, which would yield an incorrect model. By adding only those constraints that are clearly correct, the user both saves time and achieves the best result.
Lastly, Figure 3.18 highlights the cuboids and the 84 constraints involved.

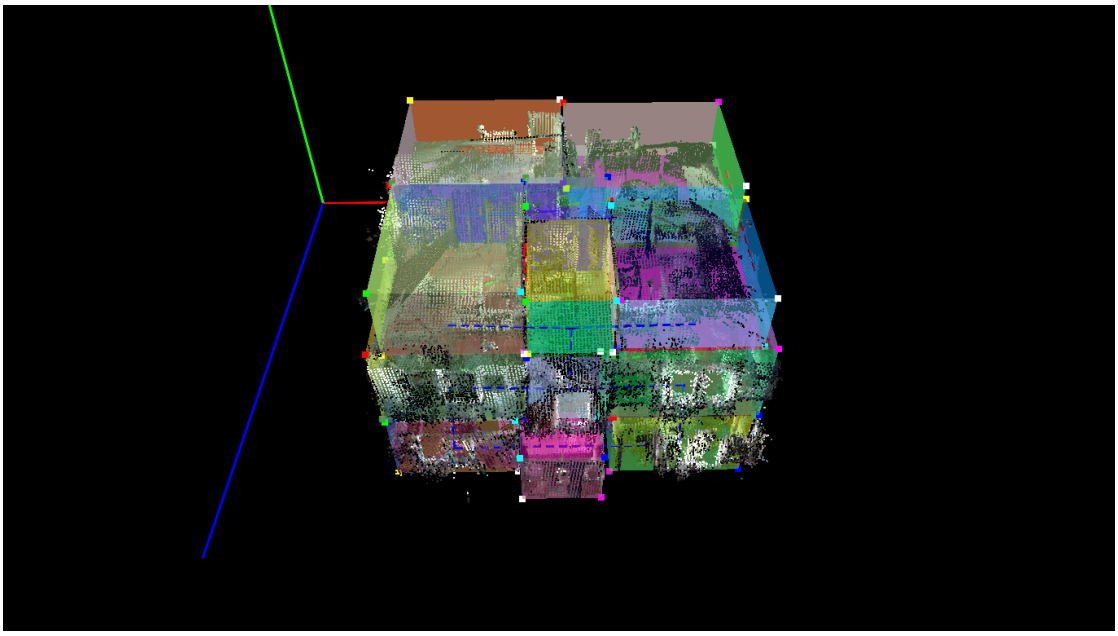

Figure 3.17: Final assembly containing all 23 rooms.

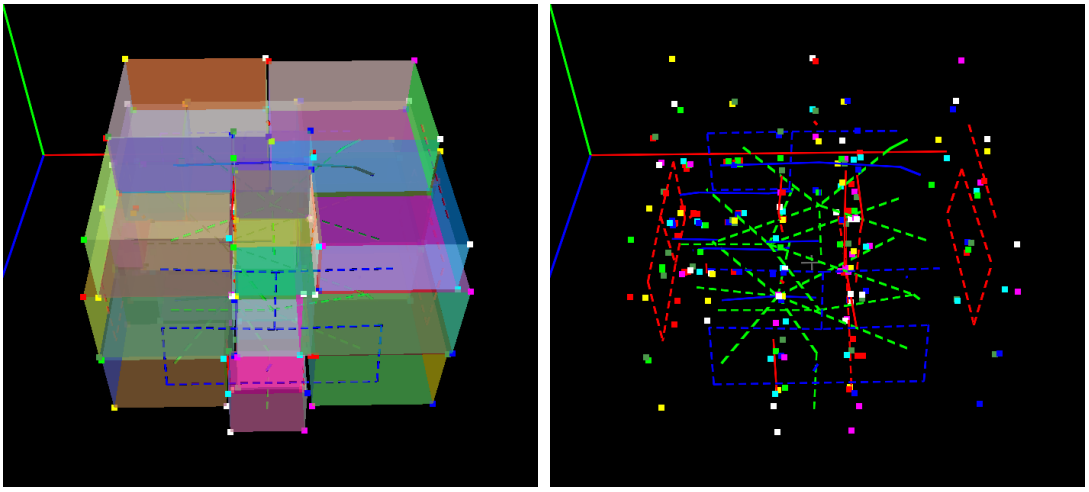

Figure 3.18: The final cuboids and 84 wall constraints are highlighted.

3.6.2 Exceptional cases

HouseScan also allows working with rooms that are not obvious cuboids.

Figure 3.19 shows an attic room with a slanted roof (left) which lacks a 6^{th} side plane on the side where the slant is low. By rotating the view (right), we can see the points through which the true outer wall should run (red circle). This plane has not been recognised within the chosen user threshold because many other planes in this room are more prominent. To solve the issue, the user can choose to increase the number of planes recognised, or manually create the correct plane by selecting 3 that define it.

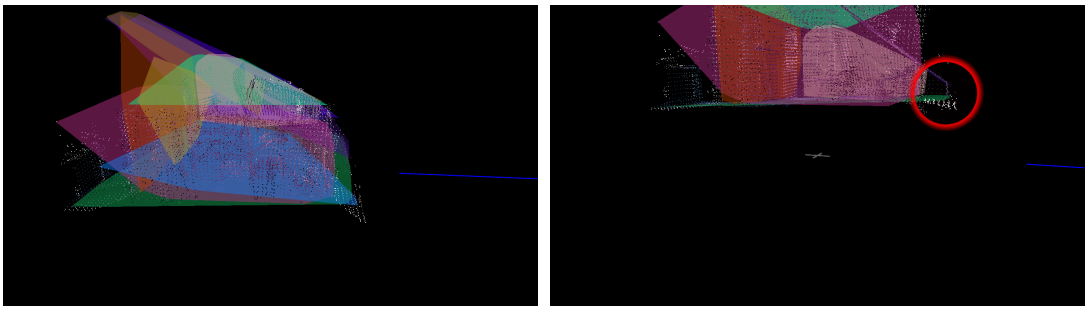

Figure 3.19: An attic room. The slanted roof is clearly visible and has been recognised as a prominent plane (left). Small wall section has not been recognised (right, circled).

Another exceptional case is when the outermost planes of a room are not the true walls. This happens when a room contains an outset that protrudes into another room, like the small section of the kitchen in Figure 3.20 (annotated with an arrow on the right).

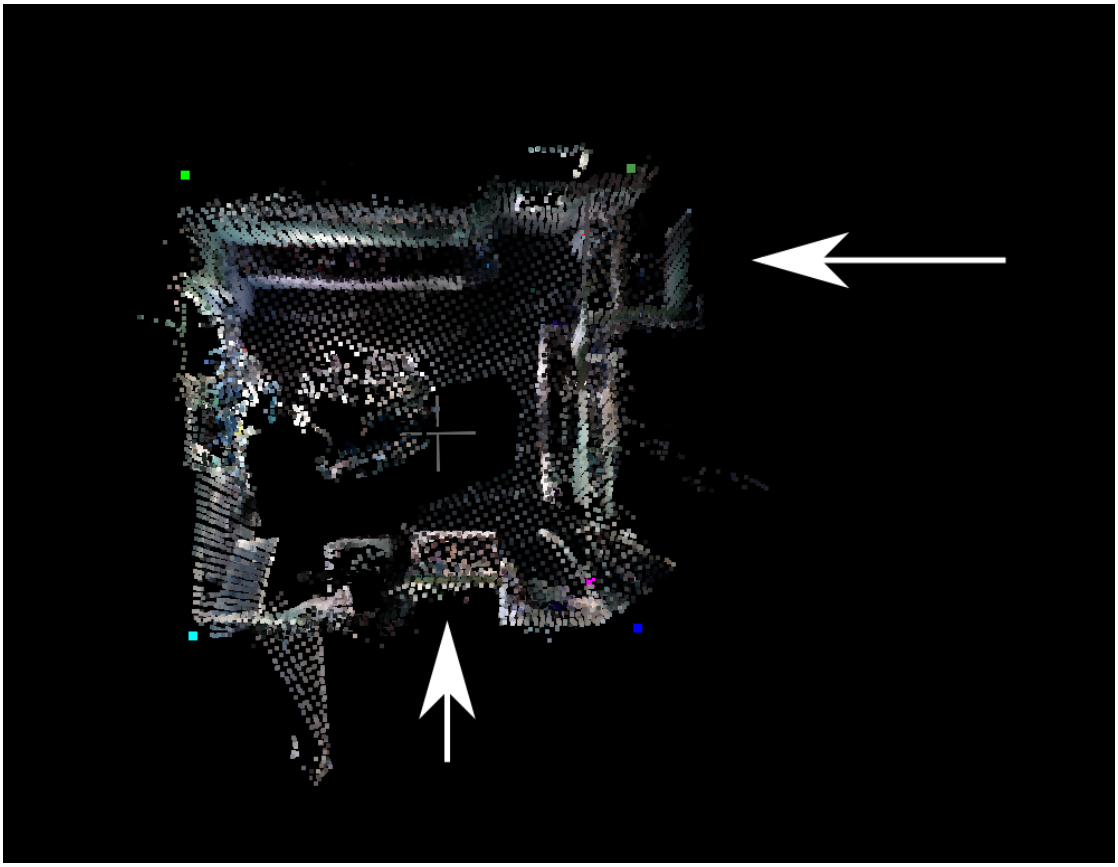

Figure 3.20: High-angle shot of a kitchen demonstrating a protrusion and a flue.

The flue in the kitchen of Figure 3.20 (lower arrow) also shows how a wall can be completely interrupted in the middle, splitting it into two non-contiguous planes. As discussed in section 3.3, the RANSAC approach chosen for *HouseScan*'s plane detection correctly identifies the two sections as a single plane, while a Region Growing approach would not.

3.6.3 Inspecting the model in full resolution

The work is done, the house is reconstructed! Using the `plyxform` tool, the final position for each room is swiftly applied to the original mesh models in full resolution. The combined model consists of 44 million coloured triangles.

Figures 3.23, 3.21 and 3.22 display the result from three different perspective. The outer walls were partially removed to allow an unobstructed view.

This is the time when the user leans back and enjoys.

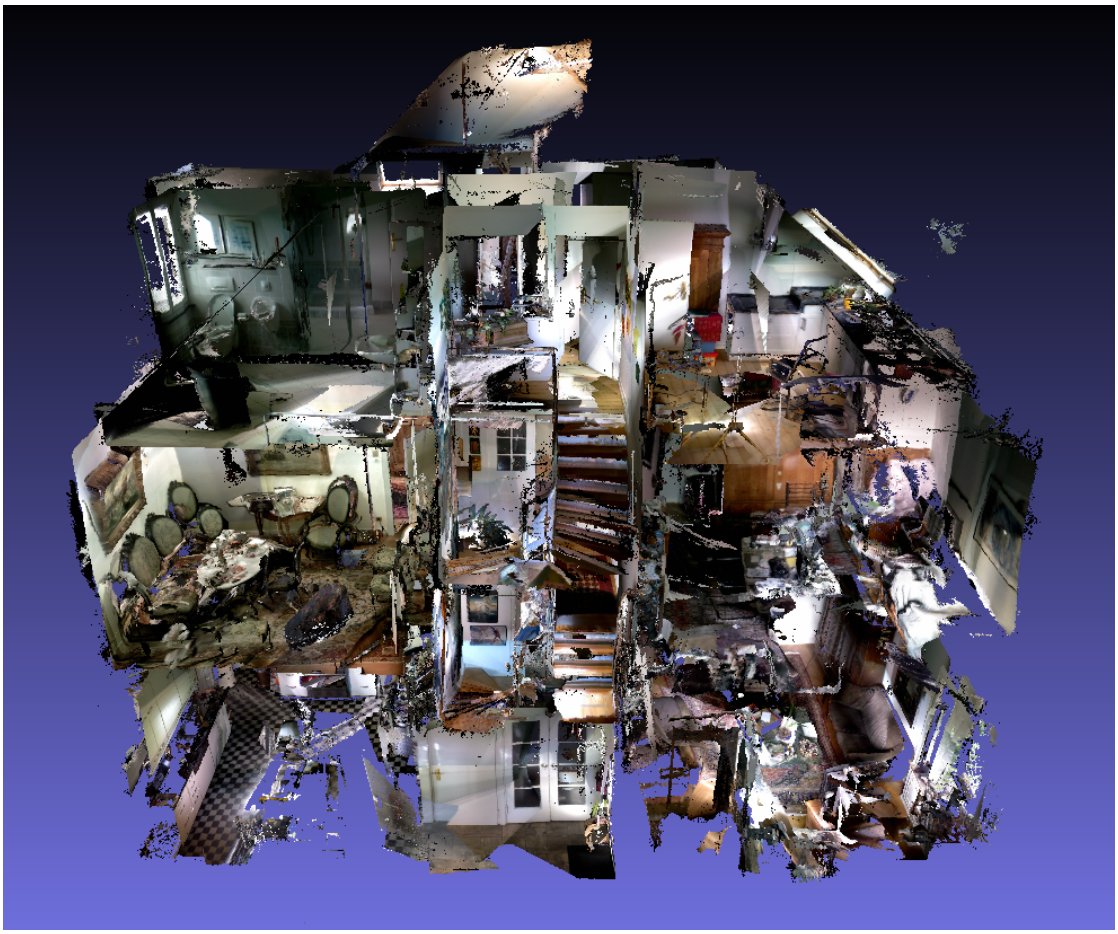

Figure 3.21: Front view of the the reconstructed house in full resolution. The final model consists of 44 million triangles with per-vertex colours. Backface culling was enabled to allow an unobstructed view into the rooms. Shading and lighting are disabled in the viewing program – *real lighting* is at work.

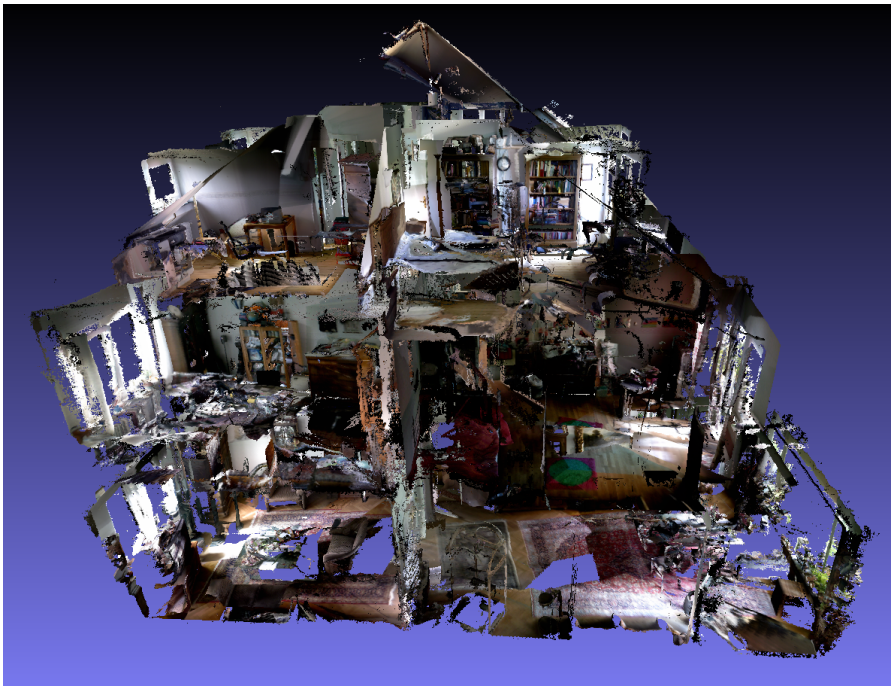

Figure 3.22: View from the right side of the final model. Each room was reconstructed from 512^3 voxels, preserving details up to high zoom levels.

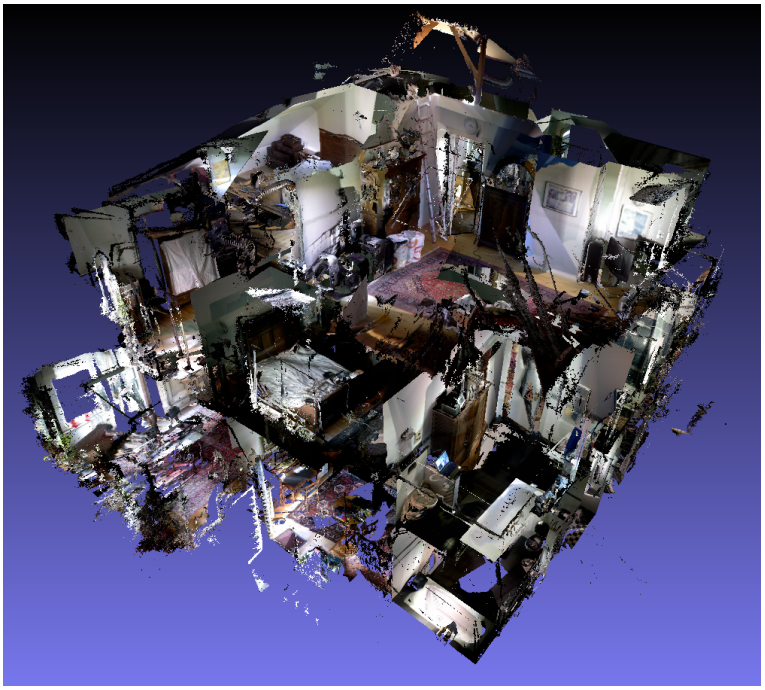

Figure 3.23: View from the back of the final model.

3.7 Implementation

This section discusses the Software Development aspects of *HouseScan*, and focuses on its main part: the room fitting and alignment program.

3.7.1 Programming languages and libraries

> *"If you do it in Haskell, of course, your problems disappear."*
>
> – Tony Field

HouseScan is written in Haskell.

When writing a Master's Thesis of an experimental nature, goals and approaches can vary quickly. Haskell allowed me to stay flexible, performing radical changes when needed, and be confident that this would introduce few bugs into my code.

I initially considered writing *HouseScan* entirely in C++, because *KinFu*, being part of the PCL library, is written in that language, and I assumed that I would have to interact with *KinFu* a lot. I also wanted to make use of PCL's extensive libraries for registration, segmentation, feature detection, 3D data structures and algorithms. However, once I started writing the `detect_planes` program (also using PCL) I noticed that I could not work quickly enough with it. Because PCL makes extensive use of templated header libraries, compiling my single file took almost 20 seconds, restricting me to verifying only three changes per minute. I tried to improve the situation by using pre-compiled headers, but changing PCL's build system to have this work reliably is a major undertaking and outside of the scope of this project. In addition, C++ does not lend itself to quickly prototype many different approaches, which was important for me. PCL carries significant legacy code, and some parts of it were one-off contributions of academics that chose to make PCL-based code available with the publication of a paper. As a result, PCL code can be difficult to understand and lack maintainers[19]. To give an example, many different types in PCL are type-aliased to `PointT`, different types are used for the same things in different places, and the core point data types are diabolic combinations of templates, `unions`, low-level optimisations, backwards compatibility and preprocessor macros. While PCL's utility is beyond question, these issues make exploring it very time consuming, especially for doing supposedly trivial things.

Eventually *KinFu* and the rest of *HouseScan* did not need to be as intertwined as I originally expected, and saving the reconstructed point clouds and meshes were enough of an interface between them.

By now, I am glad to have switched to Haskell after the experience described above, and it did make prototyping much easier.

3.7.2 Design and approach

I started with a simple OpenGL application to display point clouds in order to evaluate how accurately my RGB-D camera displays distances.

[19] In any case, this is still much better than not making the code available at all!

To get camera data into my program, together with Patrick Chilton I wrote the `honi` library [43], a Haskell binding[20] to the *OpenNI* C/C++ library for *PrimeSense* based RGB-D camera. To obtain the correct USB device from the operating system, we also wrote a binding [44] to the cross-platform `hidapi` USB library [45].

When I found out that PCL did not have functionality for my cuboid fitting, I extended my point cloud viewer with the BFGS method described in section 3.4. I also made it possible to visualise the intermediate steps in the BFGS optimisation. This revealed how unnaturally the cuboid moved when the rotation was defined around the origin, and led me to change it to rotate around its centre instead.

After having added plane and point selection (see section 3.7.3.1), I implemented wall constraints and the room pose optimisation (section 3.5). While my problem was specialised to rooms, thus carrying a type signature like

```
lstSqDistances :: Map (Room, Room) Double -> Maybe (Map Room Double)
```

Here, the input is a `Map` that expresses the desired distances between rooms, one for each wall constraint, and the output a `Map` that contains the optimal position for each room. Since the equation system solved inside might be singular, potentially having no solution, the result is wrapped in a `Maybe` type.

In Haskell it is common practice to write functions in their most general form. The constraint graph I'm working with is obviously not tied to rooms, so the actual type signature I used is

```
lstSqDistances :: (Ord a) => Map (a, a) Double -> Maybe (Map a Double)
```

thus optimising positions between any set of things that can be given an order.

This introduces another task: Each room in *HouseScan* has an ID over which this order is defined, but the IDs are not necessarily contiguous. However, since we need to contain them in a matrix at some point to solve the inconsistent equation system (see 3.5), we would prefer to transform the IDs to a contiguous range like $(1 \dots n)$. Since the result must be in terms of the originally passed-in room IDs again, we need a bijection.
This is done by the following `biject` function, which, given a list of orderable things, returns the *forward* and *backward* functions of a bijection between those things and $(0 \dots n-1)$:

```
biject :: (Ord a) => [a] -> (a -> Int, Int -> a)
biject list = (forward, backward)
  where
    set         = ordNub list -- removes duplicates
    forwardMap  = Map.fromList    $ zip set          [0..]
    backwardMap = IntMap.fromList $ zip [(0::Int)..] set
    forward  x  = let Just v = Map.lookup    x forwardMap  in v
    backward x  = let Just v = IntMap.lookup x backwardMap in v
```

Haskell makes working with mathematical concepts concise and convenient.

The final architecture of the program is shown in Figure 3.24.

[20] A *binding* is a bridge between programming languages. It allows to call library functions to be called from a different programming language that the one in which library is written.

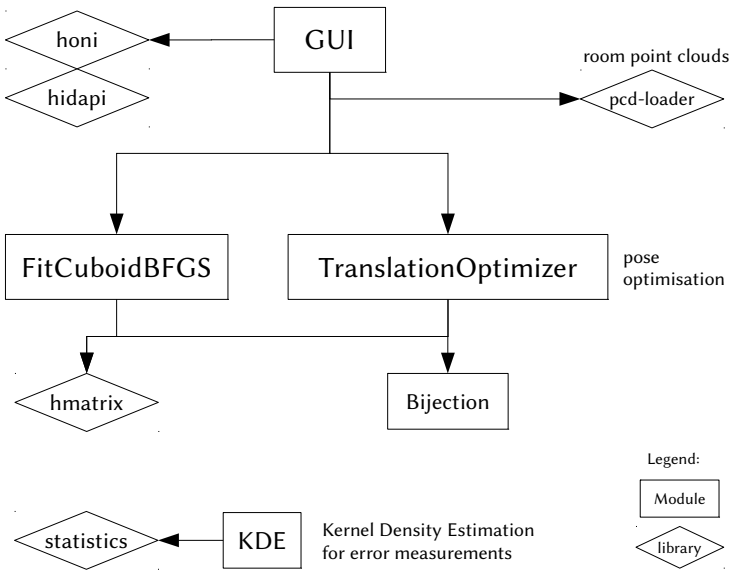

Figure 3.24: The architecture of *HouseScan*'s cuboid fitting and room alignment program. Boxes are Haskell modules, diamonds are external libraries used or written for this project.

3.7.2.1 Testing I made use of the excellent `QuickCheck` [46] library property testing of automatically generated test cases.

`QuickCheck` allows me to write first-order logic expressions, like

```
\(list :: [Int]) -> mySort list == sort list
```

which says that for all lists of integers, the sort function that I wrote does the same as Haskell's inbuilt sorting function.

`QuickCheck` will then generate millions of test cases for me, and check if the above property holds. If it doesn't, it will try to find the smallest counter example that still fails, in order to make debugging easier.

During the development of *HouseScan*, I used `QuickCheck` in particular to make sure that improved versions of my cuboid fitting algorithm (mentioned at the end of section 3.4) did not introduce bugs.

3.7.2.2 Backwards compatibility When adding new features to *HouseScan*, it was often necessary to change important data structures, such as the `Room` data type.

This posed a backwards compatibility problem. When working with rooms and assembling buildings, I often saved my progress to disk to continue working later or to make snapshots of each step for documentation purposes. If I just went ahead and changed a data structure, the contents of the files saved so far would become inconsistent with what my program expected them to contain.

I solved this problem using the `safecopy` library for Haskell [47].

It uses type families to express *migrations* between data types, allowing me to declare how past versions of a type evolved into newer ones.

If an old save file is encountered, the declared migrations are replayed up to the most recent version of each type. This way, *HouseScan* can open all versions of all data types ever created during its development.

3.7.3 Interesting technical aspects

3.7.3.1 Object picking For allowing the user to work with planes and points, I had to make it possible to select the 3D objects with the cursor. This is called *object picking* and there are various ways to do it. If the objects are simple geometric objects that have closed-form solutions for being cut with lines, a *ray casting* method can be used; for this, a line is constructed that goes through the pixel the user clicked on, and it is tested for intersection with all available objects.
However, I was interested in picking points which essentially do not possess any geometric volume; it would thus be very unlikely for any ray to run exactly through a point. Consequently, I used another technique instead: *colour picking*. It works by rendering the same scene to an invisible screen, but exchanging the objects' true colours by a unique colour value per object. One can inspect what colour is at the pixel under the cursor, and look up which object was rendered with that colour. This method has the benefit that it is *pixel-correct*: independent from custom geometry code, it will always return the object that was drawn at a pixel. In addition, since it is just another rendering run, this method is automatically GPU-accelerated, while *ray casting* is usually done on the CPU.

3.7.3.2 Floating point colours and very fast loops For legacy reasons, colours in PCL point clouds are represented in saved files using a single `float` value instead of the usual `uint32_t` (as mentioned in the documentation of the `pcl::PointXYZRGB` data type in PCL version 1.7).

Therefore, if a PCL point is loaded from disk, it has to be converted from `float` to a 32-bit integer. In the C programming language, this is called a "memory reinterpretation cast" and performed using

```
float f;
uint32_t i = * (uint32_t *) &f;
```

Haskell is a language that does not by default allow access to low-level memory layout in this fashion.

After benchmarking to find the fastest way to do it, I published a Haskell library [48] that performs the equivalent of the above.

To make sure that my casting function is correct, I wrote an *exhaustive test* that makes sure that casting any possible 32-bit unsigned integer to a `float` and back would give me the same result:

```
forM_ [0 .. maxBound :: Word32] $ \w ->
  floatToWord (wordToFloat w) `shouldBe` w
```

This is possible because the 2^{32} possible values can be enumerated on a modern processor in only a few seconds.

When writing this test, I discovered that Haskells *list fusion*, which is supposed to turn lists like the above into efficient loops in the generated machine code, does not work well for `Word32`.

This spawned an interesting discussion [49] on the corresponding mailing list and eventually led to a fix [50] in the core Haskell `vector` package, which improved `Word32` performance in cases like this by factor 8.

3.7.3.3 Hot code reloading

HouseScan implements a form of *hot code reloading*, that is, changes on its source code can take immediate effect on the state of the running program without having to restart it.

At any point in *HouseScan*'s 3D fitting and alignment program, the commands

```
:reload
restart mainState
```

can be issued to restart it in-place, without resetting the current position, rotation etc.

This feature is exploiting the fact that the `ghci` interpreter is not allowed to garbage collect an object during a `:reload` if it is referenced by foreign memory. The `foreign-store` library [51] uses this to make named objects survive reloads.

When two versions of a function with the same name are around, it is difficult to get hold of the right one.
The `restart` function of type `(State -> IO ()) -> IO ()` is carefully crafted to hand the internal state of the old code to the reloaded code. It is important that the entry point to the program, `mainState :: State -> IO ()`, is passed into `restart` from the interpreter shell, since this is the only way we can refer to the freshly loaded `mainState` function.
Writing a reference to `mainState` into the `restart` function itself would always bind the old function, which forbids writing an interactive restarting function of type `restart :: IO ()`.

Since the application is multi-threaded and uses OpenGL, which comes with non-trivial external resource allocation, we cannot simply call the new `main` function on the new state right away. Instead, we have to use a reference of type `restartFunction :: IORef (IO ())` accessible from the old state, into which we write a continuation:

```
restart :: (State -> IO ()) -> IO ()
restart mainStateFun = do
  oldState <- readForeignStore _STATE_STORE_NAME
  -- [...] - clear parts of state that can never survive
  --         a reload, like GPU buffers objects
  sRestartFunction oldState $= mainStateFun newState
```

which is be executed by the OpenGL display loop in the old code once the next frame has finished rendering.

4 Ubierhouse: A new RGB-D video data set with architectural ground truth

The usefulness of a mesh model as the 3D equivalent of a floor plan is determined by its deviation from reality.

To measure this deviation in models created by *HouseScan* and to allow for a sound comparison with potential future systems, I have recorded a new RGB-D data set of a three-level residential house containing 23 rooms, and measured ground truth within each room and across multiple rooms.

We will refer to it as the *Ubierhouse* data set.

The Ubierhouse is a detached house with rectangular foundation. It was built in 1932 and since then has seen a series of major refurbishments. Its rooms exhibit a variety of shapes, colours, and lighting[21].

The floor plan of the Ubierhouse is included in the Appendix.

4.1 Video recording

One RGB-D video in the `.oni` format was recorded for each room, with a resolution of 640×480 at 30 FPS for both the depth and colour streams.
Traversals through doors were not recorded.
To make sure a *KinectFusion* reconstruction is actually possible from these videos, they were recorded *in parallel to performing a reconstruction* with *KinFu*.

To provide some extra utility, coloured point clouds, colour triangle meshes, and TSDF and colour volumes created by *KinFu* are also contained in the data set.

The concrete command line used to create each video is

```
pcl_kinfu_app --registration --integrate-colors --normals -volume_size
5.0
```

where the volume size was adjusted per room to fit it as well as possible. This size for each room is saved in the header of the corresponding TSDF volume file.
The simultaneous recording option does not have its own command line switch at the time of writing, and was enabled inside the *KinFu* source code. It will hopefully be added to upstream *KinFu* soon.
The reconstruction machine was a gaming-grade laptop containing a NVIDIA GeForce GTX 780M GPU, an Intel Core i7-4700MQ CPU, and 16 GB RAM, operating under Ubuntu 14.04. The *KinFu* reconstruction ran at frame rates between 22 and 30 FPS.
The size of the reconstruction grid was 512^3 voxels. The initial camera pose was in the centre of the reconstruction cube, directed at the centre of one of the cube faces.
The remaining parameters were unchanged *KinFu* defaults, quoting:

```
setInitalCameraPose (pose);
volume().setTsdfTruncDist (0.030f);
setIcpCorespFilteringParams (0.1f, sin ( pcl::deg2rad(20.f) ));
setCameraMovementThreshold(0.001f);
```

[21] And sometimes contain some peculiar objects.

The recordings were done with an *Asus Xtion Pro* RGB-D camera, with calibration parameters obtained by using a checkerboard as described in section 3.2. The focal lengths were set in *KinFu* as:

```
setDepthIntrinsics(535.002029, 535.002086);
```

The automatic white balance and brightness correction of the RGB camera firmware was left enabled, which was necessary to compensate for the large variations in brightness when scanning past windows.

All mirrors in the house were covered to avoid phantom rooms.

Most doors were left half open to allow a natural exploration of the combined model[22], and were not moved between scans if at all possible, but this was not always so.

The camera was moved in roughly a 360° turn, following the *scanning tips* from section 3.2. Occasional diversions were taken to capture places in corners, under tables and on top of high furniture.

All videos are between 2 and 6 minutes long. The total amount of video data is 200 GB.

4.2 Ground truth

The ground truth for the Ubierhouse was obtained using a *Bosch PLR 50* laser distance meter.

For each of the 23 rooms, at least three distances were measured, including the horizontal width and length as well as the height of the room. Each distance was measured three times, and the mean was taken as the ground truth[23].

Further, distance measurements through multiple rooms were taken when the architecture allowed a direct line of sight. In particular, distance values through the *entirety* of the house are known along all three axes.

The *PLR 50* has a measuring range from 5 cm - 50 m and per its data sheet an accuracy of ± 2 mm. Its display has millimetre resolution.
Its correct function on distances up to two meters was verified using a folding ruler bought from a German hardware store.

[22] Of course, this is only to not confuse end users. Experts in Computer Vision can look through doors and apparel.

[23] The *PLR 50* reports distances very consistently when it is pressed against a wall and not moved: The repeated measurements did not vary by more than $1mm$.

Figure 4.1: Left: A *Bosch PLR 50* handheld laser rangefinder. Right: A *WORKZONE* commodity ultrasonic sensor.

Each measurement carries a descriptive name to identify it in the data set.

To make it clear where measurements were taken, both laser and sonic measurements are recorded into the floor plan of the house, which can be found in the Appendix.

However, the geometry of a real room is not precisely captured by a floor plan. Inaccuracies during the construction, ageing of materials and subsidence led to displacements, such that walls which are parallel in the floor plan are not parallel in reality, giving different distances between the same walls depending on where the measurement is taken.
For this reason, each measurement comes with a screen shot of a 3D model of the house, in which a virtual tape measure indicates precisely where the measurement was taken (see Figures 4.2 and 4.3 for examples).

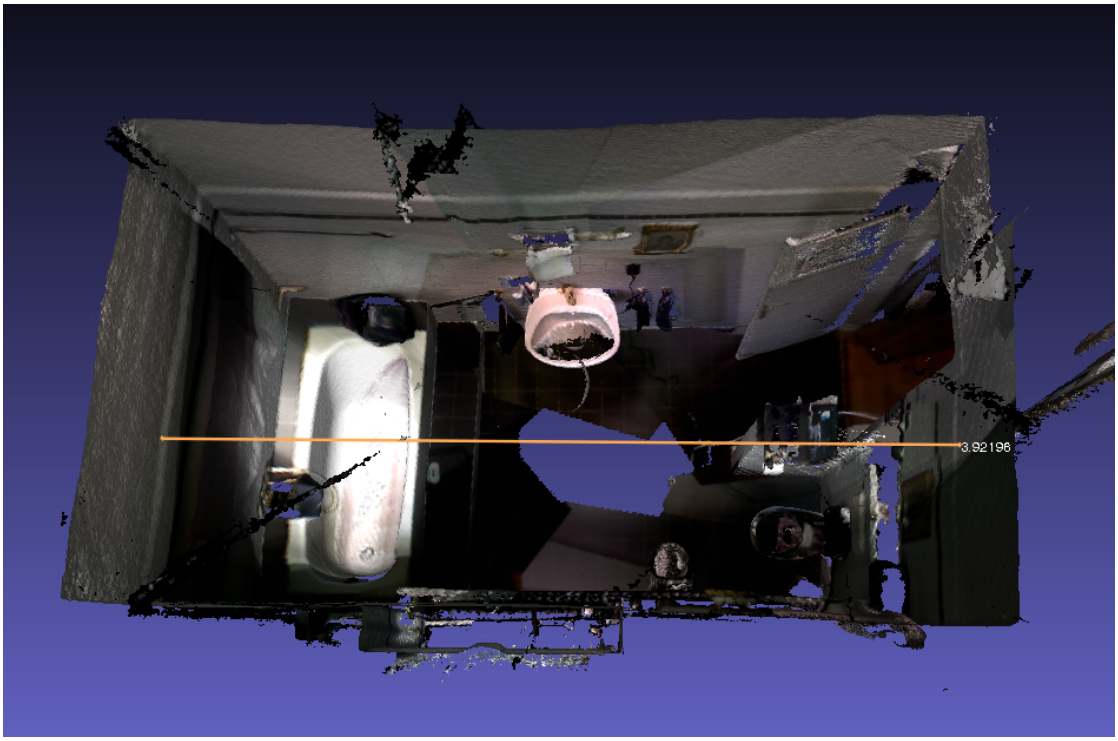

Figure 4.2: A screen shot of the reconstructed room "bad1". The yellow line indicates where ground truth was measured.

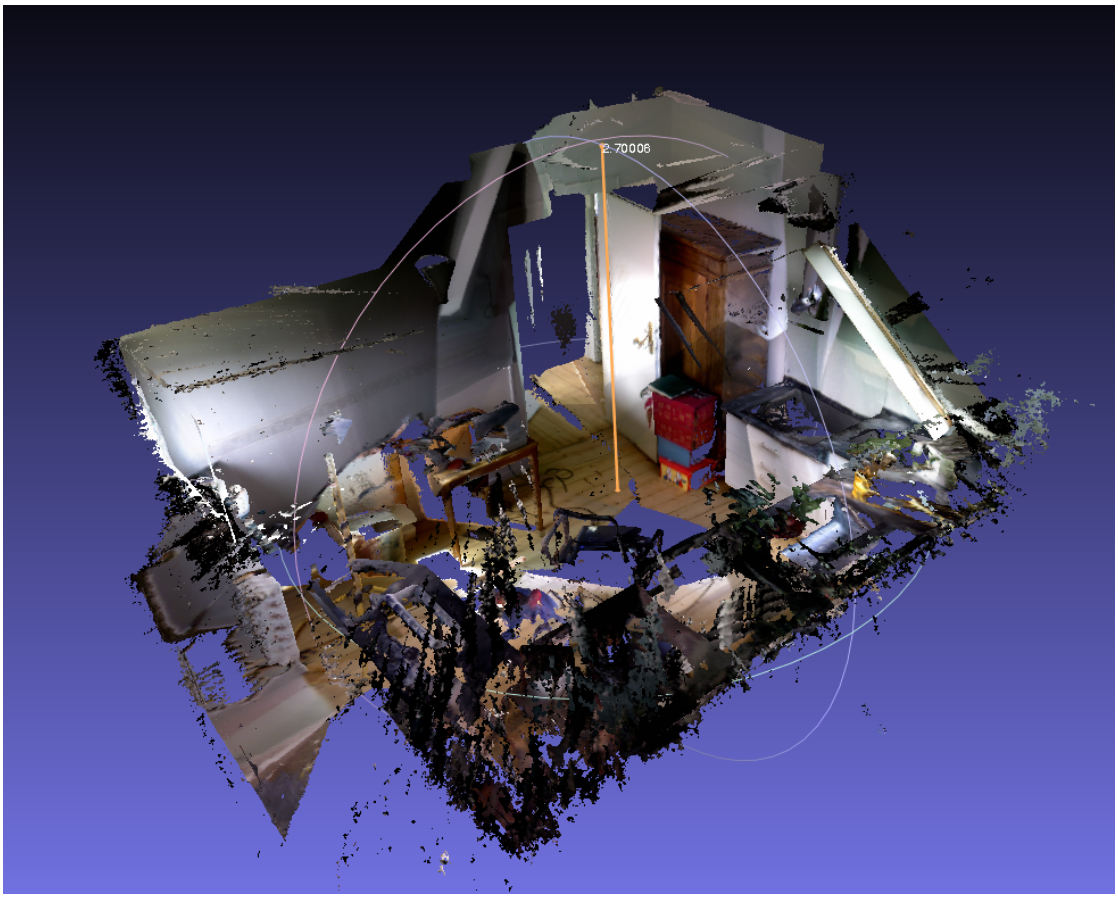

Figure 4.3: A screen shot of the reconstructed room "kuecheganzoben". The yellow line indicates where the ground truth for the height was measured. The indicator is especially important for this room because the ceiling has different heights at different places.

4.3 Additional measurements: Ultrasonic distances

Consumers do not usually work with high-precision laser measurement tools.
To give an idea of what measurements one would get using more common tools, for most distances taken with the *PLR 50* I have recorded a corresponding value with a *WORKZONE* consumer-grade ultrasonic distance meter (see Figure 4.1), which was bought from a local supermarket for 13 €. Its measuring range is 60 cm - 16 m. A laser pointer at its front allows to orient it towards the target. It is advertised with an error of $\pm 1\%$ and its display has centimetre resolution.
The ultrasonic sensor cannot measure as robustly as the laser measure, and its sound signal is erroneously reflected especially by edges and corners close to the desired measuring line.
Measurements that are likely incorrect for this reason are marked with a special label in the data set.

4.4 A challenge to unpublished implementations

I invite the authors of *KinectFusion* implementations that are not publicly available (and thus eluding critical scrutiny) to reconstruct the rooms of the Ubierhouse from this video data set, and report their reconstruction accuracy.

I aim to maintain a public list of benchmark results at nh2.me/ubierhouse, similar to how the *MNIST database of handwritten digits* [52] does for Machine Learning.

Also called for are the results of implementations which, like *HouseScan*, reconstruct a combined model of the house, so that the error of such systems can be compared.

5 Evaluation

This chapter evaluates the performance of *HouseScan* as described in chapter 3 on the Ubierhouse data set from the last chapter.

It is important to make a difference between the reconstruction error *within* rooms, which is entirely attributed to the *KinFu* scan (section 3.2), and the error *between* multiple rooms in the final model of the whole house, which is a combination of the individual room error plus errors (or corrections) introduced by cuboid fitting (section 3.4) and room pose optimisation (section 3.5).

5.1 Within-room error

All 23 rooms of the Ubierhouse were reconstructed using *KinFu*.

To my knowledge, this is the first time that the *reconstruction accuracy* of any *KinectFusion* implementation is measured. Existing work either focused on the development of the technique itself [1], on how to extend the scanning range [2], or on the error of the camera trajectory [25].

In the absence of an established methodology for measuring the accuracy of dense reconstructions, I propose the following method that is tailored to rooms:

- For each reconstructed room, consider at least three distances: the two principal horizontal room dimensions and a vertical height.
- Compute the *relative error* for each distance as

$$e = \frac{d_{\text{model}} - d_{\text{true}}}{d_{\text{true}}}$$

- Compute the mean μ_e and sample standard deviation σ_e of these relative errors over all distances considered.

μ_e is a *reconstruction bias*, indicating whether the reconstruction method over- or underestimates distances. Once μ_e is known, models created with the method can be *bias-corrected* by scaling them by $1/(1 - \mu_e)$.

σ_e is the most important result and invariant to bias correction. It indicates how much reconstruction error we have to expect for any scanned room.

73 of the ground truth values in the Ubierhouse data set are within-room measurements. For each of them, the equivalent distance in the *KinFu*-reconstructed room was measured by using *Meshlab*'s "Measuring Tool"[24] on the exported `.ply` file. An example of this can be seen in Figure 5.1. The *KinFu* parameters were those of section 4.1: Video recording.

Table 5.1 shows sample values for d_{true}, d_{model}, the relative error e and the bias-corrected relative error $e - \mu_e$. It also shows the corresponding distances obtained from the floor plan (see Appendix) and with the commodity ultrasonic distance meter (see section 4.3). The complete table can be found in the Appendix.

[24] It is useful to enable *Meshlab*'s "Backface culling" option for this task in order to peek through outer walls.

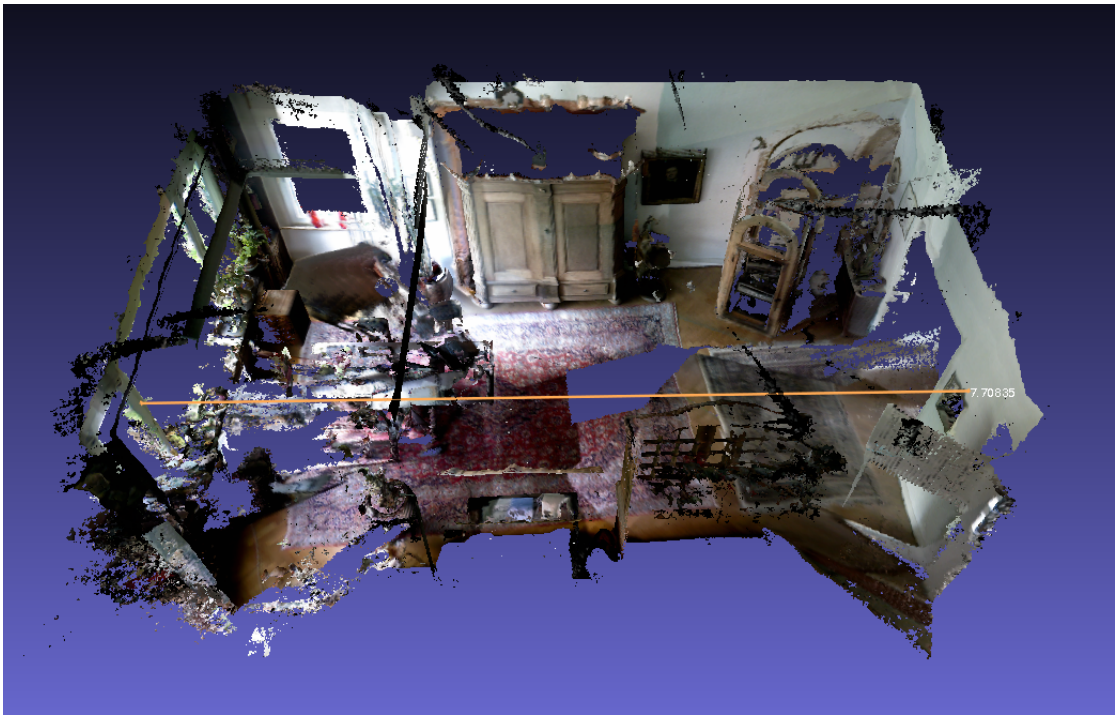

Figure 5.1: *Meshlab*'s "Measuring Tool" is applied to a living room.

room/measurement ID	d_{true}	d_{model}	$d_{\text{floorplan}}$	d_{sonic}	e	$e - \mu_e$
wc1-wall-...	1.543	1.609	1.50	1.54	0.043	-0.003
wc1-wall-...	1.240	1.269	1.25	1.14	0.023	-0.023
wc1-height	2.753	2.811	-	2.76	0.021	-0.025
arbeiten1-...	4.400	4.379	4.47	4.40	-0.005	-0.051
arbeiten1-...	3.715	3.922	3.70	3.71	0.056	0.010
arbeiten1-height	2.742	2.854	-	2.75	0.041	-0.005
wohnen1gross-...	4.296	4.239	4.28	4.28	-0.013	-0.059
wohnen1gross-...	7.275	7.708	-	7.26	0.060	0.013
wohnen1gross-height	2.715	2.840	-	2.71	0.046	0.000
...	...	...	...	...	...	...

Table 5.1: Excerpt of distances in the Ubierhouse and its reconstruction. All distances are in m. d_{true} is the ground truth obtained with a laser rangefinder. $e - \mu_e$ is the bias corrected relative error. Value names were truncated for brevity. Missing values were not present in the floor plan.

The results for *within-room* reconstructions are:

$$\sigma_e = 0.031 \qquad \mu_e = 0.046$$

A standard deviation of 3.1% relative error makes *KinFu* slightly less accurate than the ultrasonic sensor, whose standard deviation from the ground truth is $\sigma_{\text{sonic}} = 1.2\%$ with a bias of $\mu_{\text{sonic}} = -0.3\%$.

It is important to note that the ultrasonic sensor sometimes measures wrong distances as discussed in section 4.3. This happened for 6 out of the 73 measurements in the Ubierhouse data set. The above σ_{sonic} *excludes* any such erroneous distances. However, for a user who does not also have a laser distance measure[25] it can be difficult to check which displayed values are erroneous, especially for longer distances. If *all* values reported by the ultrasonic sensor were used in the calculation of σ_{sonic}, we would obtain a much higher standard deviation of 10.6%.

As a conclusion, *KinFu* creates low *within-room* reconstruction error. Its accuracy is slightly worse than that of the ultrasonic sensor, but it does not report spurious results. *KinFu* currently needs *bias*

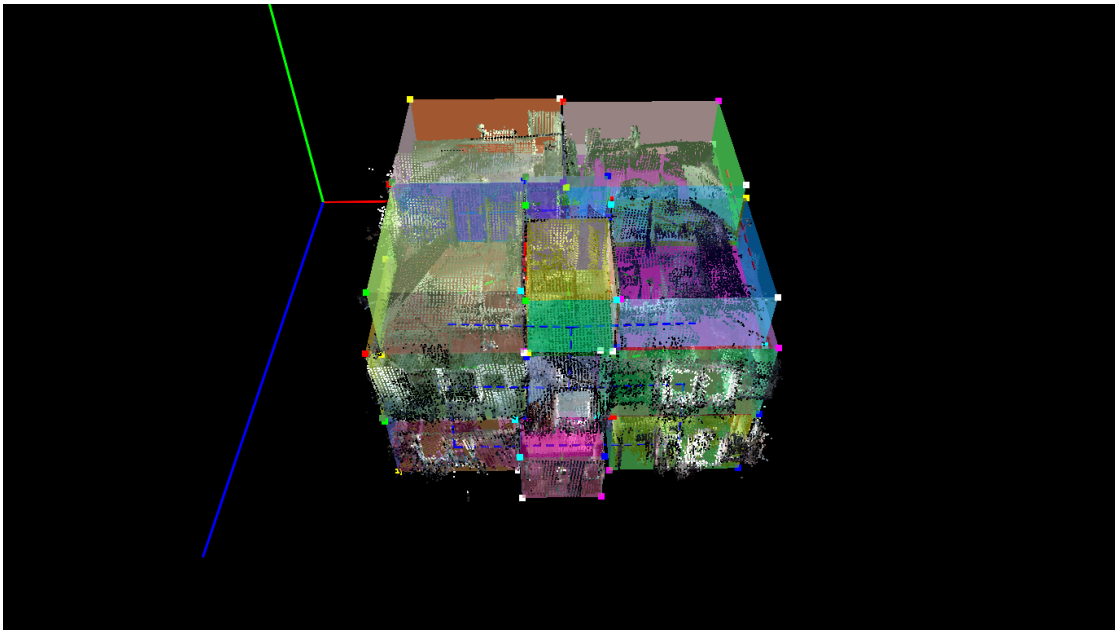

Figure 5.2: The Ubierhouse reconstructed with *HouseScan*

The results for *between-room* reconstructions are:

$$\sigma_e = 0.024 \qquad \mu_e = 0.027$$

Measurements with the ultrasonic sensor could not be obtained, since the spread of the sonic signal was too large, and consequently it was reflected too early, giving short distances.

In conclusion, *HouseScan*'s cuboid fitting and room pose optimisation do not introduce significant error above the *within-room* error. Building models created with *HouseScan* are accurate enough to provide 3D floor plans.

5.3 Error distribution

To give a better idea of the error of measurements taken from a model created with *HouseScan*, Figure 5.6 shows a histogram of the absolute error in metres across the 73 distances within the 23 rooms of the Ubierhouse (that is $d_{\text{model}} - d_{\text{true}}$ from section 5.1). The model distances were not bias-corrected. A rough Gaussian shape can be observed.

I have also compiled a *Kernel Density Estimate* of the relative errors of all model distances measured in this chapter (Figure 5.7) using the `kde` function from Haskell's `statistics` package. It clearly shows a *KinFu*'s bias towards overestimating distances. The error has a Gaussian shape with a fat tail on the negative side of the mean.

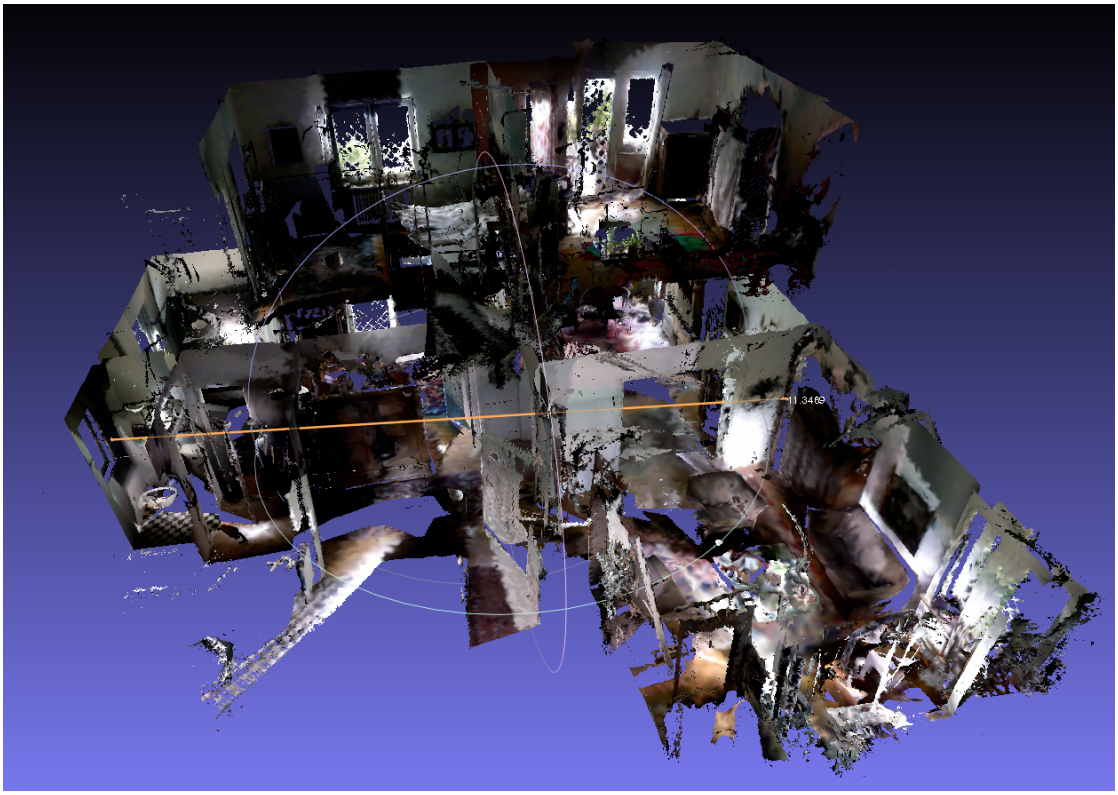

Figure 5.3: The width of the complete house is measured through 3 rooms: A bathroom, a corridor and a living room (from left to right).

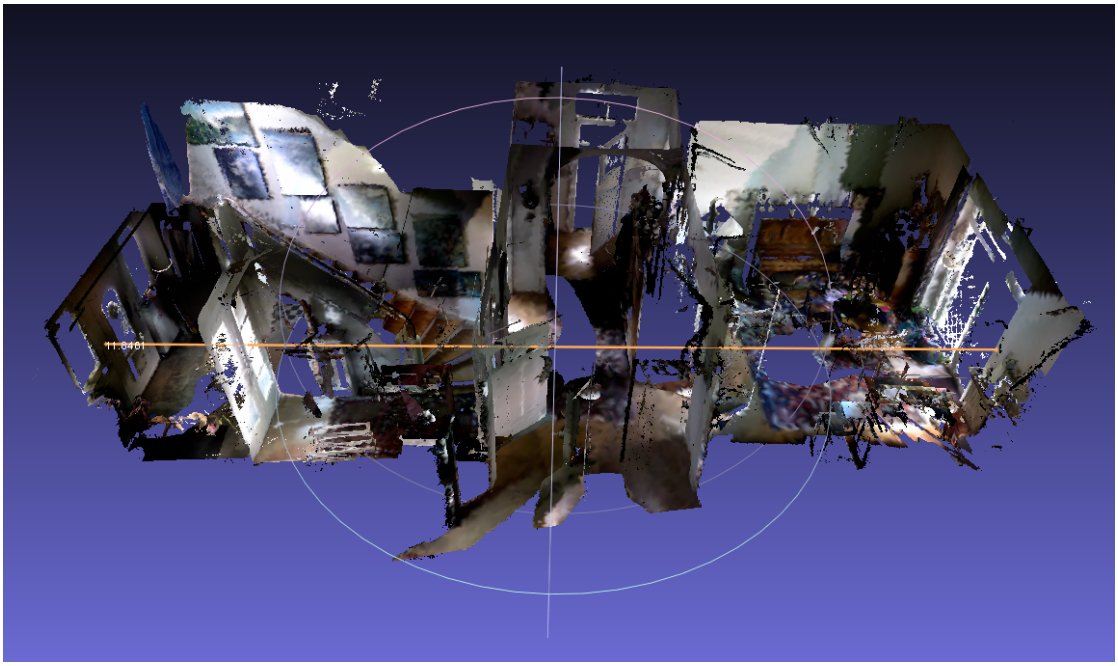

Figure 5.4: The length of the complete is measured across 4 rooms: The entrance, the staircase, a corridor and an office (from left to right).

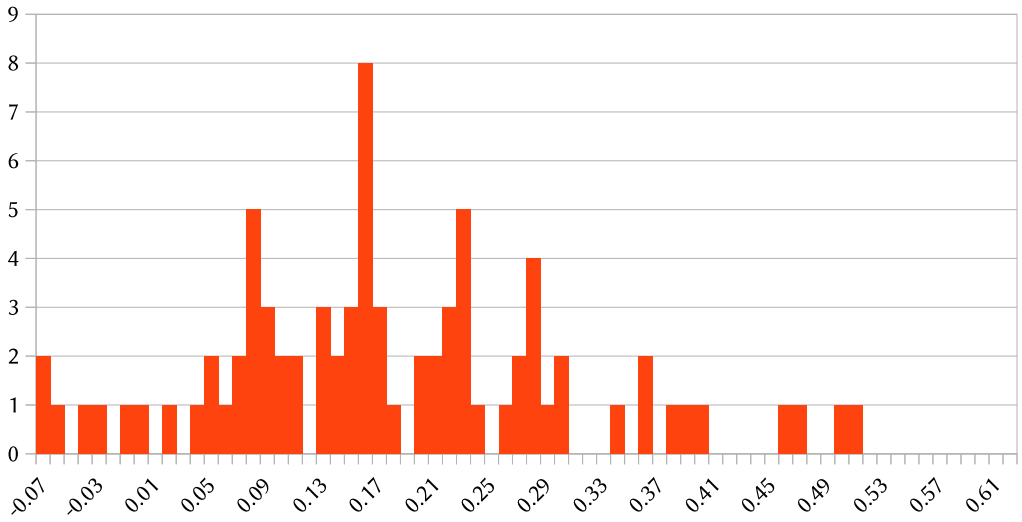

Figure 5.6: Histogram of absolute model errors $d_{\text{model}} - d_{\text{true}}$ *within rooms*. The vertical axis shows how many counts fall into each error bucket.

Figure 5.5: The height of the complete house is measured in the staircase across 3 levels.

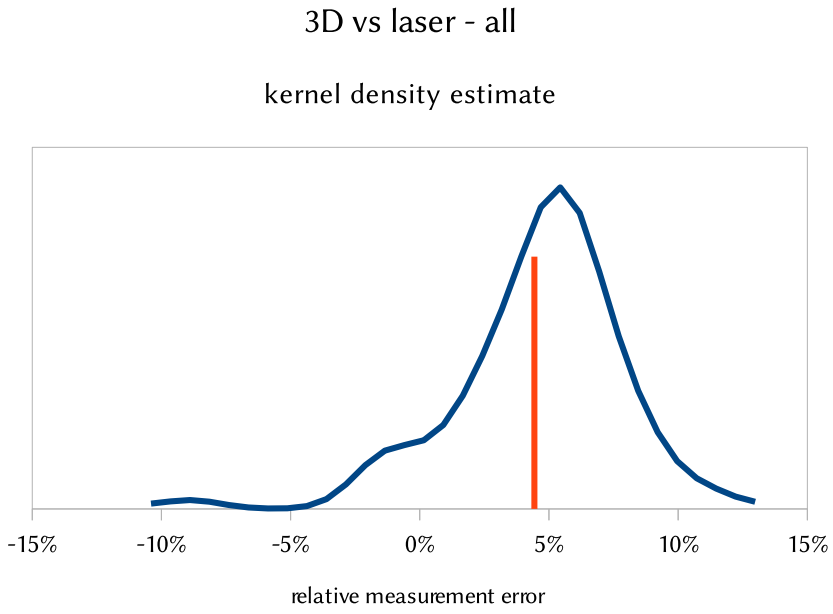

Figure 5.7: Kernel Density Estimate of relative error e across all measurements. The red line is the mean error; it shows *KinFu*'s bias towards distance overestimation.

5.4 Depth camera error

The positive bias in the reconstruction errors raises the suspicion that the RGB-D camera reports too high depth values.

To confirm this suspicion, the camera centre was pointed towards an orthogonal, plain wall with a diffuse surface[26], and depth measurements were compared with true distances. As before, the camera is an *Asus Xtion Pro*.

An ideal camera would report the true distances. In practice, one might encounter a combination of the following types of distortion:

1. A *constant offset* from true values.
2. A constant *scaling factor*, stretching or compressing distances.
3. Nonlinearities.
4. An independent choice of the above *for each pixel* in the depth image.

For this experiment, the depth at only the centre pixel was measured, so a statement about point 4 cannot be made.

Figure 5.8 shows the result of a linear regression for distances close to the camera, in the range of 0.75–2 m (typical for *KinectFusion* reconstructions of researchers' tables and offices). The low R^2 value

[26]Specular surfaces create a *hole* in the centre depth image as a result of the strong reflection of the IR emitter of the camera.

indicates that depth values are almost linear in this range. Yet a type 2 distortion is present, since the reported distances are too large by about 4.6%, and there is a minor constant offset of 1.5 cm.

When incorporating larger distances, as is needed for scanning larger rooms, nonlinearities appear. Figure 5.9 demonstrates that large depth values are stretched overproportionally, and that a polynomial approximation close to $x^{1.05}$ fits the reported distances much better than a linear one.

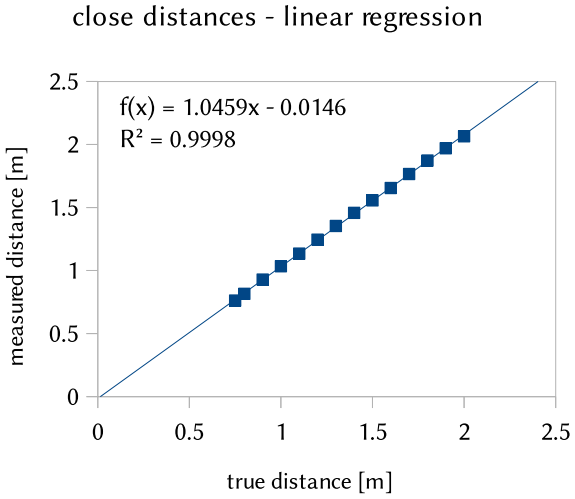

Figure 5.8: Linear regression through depth camera measurements at close distance.

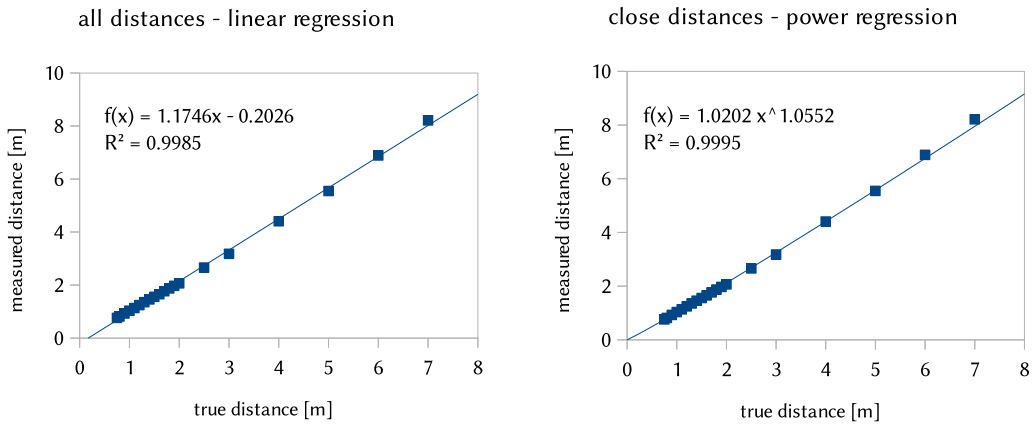

Figure 5.9: Depth camera measurements of short and long distances, regressed linearly (left) and polynomially (right). Polynomial regression fits the data better.

Without correction, this obviously is a problem for creating accurate *KinFu* scans with a moving camera:

KinectFusion assumes a rigid transformation, but the size of objects appears to change depending on how close we are to them.

At the time of writing, it is unclear whether *KinFu* contains a nonlinearity correction (or in fact any type of distance correction for systematic distortions from the above list) or whether it is the robust nature of *KinectFusion*, likely due to *dense* tracking, that makes *KinFu* work despite these inconsistencies. I have not found any related code in *KinFu*'s source repository, and there has as yet been no confirmation from the PCL maintainers.

However, the scaling factor of 4.59% is sufficiently close to the *within-room* bias $\mu_e = 4.6\%$ from section 5.1 to suggest that *KinFu* takes distance values straight from the depth camera without any further software correction.

The constant-factor depth camera error on short ranges is not a significant problem for *KinFu* reconstructions; it does not make distances inconsistent and can be dealt with using bias correction on the finished models as a post-processing step (discussed in 5.1). But it would be interesting to see if a correction of the polynomial error on longer distances could reduce the average error σ_e.

I considered elaborating upon this, but a rigorous approach would include type 4 distortions, which exceeds the scope of this thesis. However, recent work by Teichman et al. [53] describes an unsupervised approach for distortion correction via SLAM, which would fit nicely into a *KinectFusion* context.

5.5 Qualitative aspects

The above measurements show that *HouseScan*'s accuracy is similar to that of standard consumer measurement tools. The result of $\sigma_e = 3.1\%$ for *within-room* errors and 2.4% for *between-room* errors makes a reconstructed building suitable for taking distance measurements, allowing it to be used as the 3D equivalent of a floor plan.

While an ultrasonic sensor is slightly more accurate, it fails to measure certain distances, reporting wrong values as described in section 4.3. Those distances could be measured in the model without problems, making it more reliable than the ultrasonic sensor.

I expect that a *per-pixel*, *per-distance* depth calibration of the RGB-D camera as proposed in [53] will improve *KinFu*'s error significantly, and perhaps bring it on par with the ultrasonic sensor.

When it comes to reliable precision, neither the ultrasonic sensor nor the model can compete with the millimetre accuracy of a laser rangefinder.

There are special cases which the ultrasonic sensor can handle and the laser rangefinder cannot, such as measuring distances to mirrors. Surprisingly, the *PLR 50* laser tool could accurately measure distances to window glass. *KinFu* cannot recognise either of those as surfaces.

Model-based measurements have some clear utility advantages:

- A 3D model contains *all* distances in a room, from any point to any point. Modelling a property with *HouseScan* takes significantly longer than taking a few measurements with a 1D tool, but can make subsequent trips to the property unnecessary.
- Colour information and detailed geometry are preserved.
- Distances can be taken between very angled surfaces, on which conventional tools often fail.
- Measuring from physically hard-to-reach places, such as gaps between furniture, is trivial with a 3D model.

- Distances *through walls* and other obstacles can be taken, while ultrasonic sensors, lasers and tape measures require a direct line of sight.

Despite their utility, there currently are some important disadvantages to *KinFu* reconstructions.

The lack of *loop closure* and *pose-graph optimisations* creates inconsistencies in the 3D models, resulting in unnatural artefacts and introducing reconstruction error. Other *KinectFusion* implementations address these issues (see section 2.3.3), but are not available to the public as of now.

From a usability point of view, track loss is the biggest problem for *HouseScan*. *KinFu* does not detect reliably when it happens (for reasons outlined in section 2.3.2) and the resulting model corruptions leads to an increased scanning time and can make reconstructing certain rooms quite frustrating.
During my experiments I discovered that *kfusion* (2.3.1.2) has a better tracking behaviour. Switching to this implementation will likely make scanning rooms much easier, but requires adding the missing point cloud and mesh extraction features to it.

Creating a model from a building with a size similar to the Ubierhouse currently takes about one day of scanning work. I expect that switching from *KinFu* to an implementation that features loop closure detection and robust tracking will reduce the scanning time to approximately 5 minutes per room.

To evaluate whether *HouseScan*'s cuboid fitting and room alignment program can be applied by non-technical users, I performed a field test with my father, asking him to reconstruct the Ubierhouse from its individual rooms. After having read the user manual (see 8.1) and asking a few clarifying questions, he successfully reconstructed the house within 3 hours. This is close to the two hours I originally needed for the same task. Using *HouseScan* semi-automatic approach is the fastest way known to me as of now for accurately combining non-overlapping scans of independently rooms, and can be done in less time than it takes a professional interior architect to produce a floor plan.

I conclude that building-scale interior 3D reconstruction works well enough for being used beyond Computer Vision research.

6 Future work

Using the presented work one can create 3D models of whole buildings with low error. Yet some challenges remain to improve upon what *HouseScan* can already do.

Improved single-room scans. The current abilities of the used *KinectFusion* implementation put a restriction on the quality of individual rooms models. Dense reconstruction techniques have advanced since the original *KinectFusion* paper in 2011 and the development of *KinFu* shortly afterwards. The new implementations discussed in section 2.3.3 have extended range, perform *loop closure* and *pose graph optimisation* for improved model consistency, and have reduced resource consumption, but as of writing, none of them are known to be available. An open-source publication that implements these ideas would benefit the quality of models created with *HouseScan*, and likely bring interior reconstruction to new heights.

As for short time goals to improve single-room scans within the current system, evaluating the effects of a *per-distance*, *per-pixel* depth calibration on the reconstruction error as described at the end of section 5.4 would be of great use.

Unrestricted pose optimisation. *HouseScan* performs pose optimisation on the room level, and like most of the related work for floor plan creation (2.1.1) it makes use of a *Manhattan* assumption that restricts rooms to cuboids and permits only axis-aligned orientations. It would be interesting to investigate how this restriction can be lifted to allow for rooms of arbitrary shape. A set of `Same` constraints (see section 3.5) which are not constrained to an axis formulates a *point-to-plane* ICP problem with a requirement for global consistency. [32] describes globally consistent ICP, but only for the *point-to-point* cost function.

Tracking holes. A problem for any *KinectFusion* based system is certainly the inability of the ICP algorithm to provide robust tracking in environments containing insufficient geometrical complexity. Existing systems like [2] and [24] try to compensate for this by adding (or falling back to) conventional RGB tracking based on sparse features in cases where dense tracking fails. But does state of the art really exploit all information in depth images?

The *point-to-plane* ICP algorithm as used in *KinectFusion* does not make a difference between a solid plane and a plane with holes, and so does not exploit open doors or window frames to benefit tracking. The *absence* of depth values is information that has been largely neglected so far.

I suggest an extension for tracking in which planes in the camera depth image are detected in real-time[27], and convex *holes* in those planes are identified. The centre points or principal components of these holes could be used as feature correspondences as it is done for conventional sparse RGB features, with the difference that they are generated from dense point clouds and, essentially, invisible.

Automatic guessing of wall constraints. Since the goal of *HouseScan* is to reconstruct buildings as architecturally correct as possible, the choice of wall constraints is left to the human user. However, building-scale maps can be useful for autonomous robots as well, in which case the constraints would have to be chosen automatically. Assuming a *KinectFusion* implementation that can traverse doors without track loss, it might be possible to create a wall, floor and ceiling constraint for each door traversal, and the robot might even be able to guess the correct wall thickness from the door frame.

[27] It has been shown that plane detection on RGB-D depth images using RANSAC real-time is feasible; systems like [54] and [55] use it for tracking.

Allowing the building to be scanned in a single traversal would naturally benefit human users as well, and having the system guess some constraints automatically would save time in the final reconstruction step.

7 Conclusion

To summarise the contributions of this thesis, I have

- discussed a method to fit perfect cuboids to rooms, by creating corners from automatically detected planes in a point cloud obtained using a *KinectFusion* implementation,
- presented an algorithm for pose optimisation of individual room scans by reformulating a constraint graph as a linear system with a least-squares solution,
- created a practical tool to perform the above effectively, featuring an easy user interface, loading and saving work, and scripting in Haskell, and
- made recent research available to end users, allowing them to use low cost, consumer-grade hardware to create high resolution 3D scans of multi-storey buildings.

I have demonstrated the effectiveness of *HouseScan* on a real world example by reconstructing a house with 23 rooms with low error.
The videos recorded for this are made available as a new RGB-D data set of indoor scenes, which to my knowledge is the first data set that captures a complete house with multiple levels. It includes extensive ground truth to encourage further research into building-scale 3D reconstruction.

As a side effect of the development of *HouseScan*, I further

- fixed *KinFu* bugs that prevented it from being used by non-programmers,
- extended *KinFu*'s marching cubes algorithm to include colour, which is required to obtain models that can draw level with the utility of photographs,
- extended *kfusion* so that there are now two *KinectFusion* implementations available that can be compared with each other, and
- added reproducible recordings to both implementations such that reconstructions can be repeated offline and the performance of either implementation can be evaluated deterministically on a frame-by-frame basis.

Consequently, I have accomplished the goals 1 - 6 as defined in section 1.1, and developed an idea to implement objective 7 as future work in section 6.

I will do my best to contribute the changes I have made back to the respective projects, and have already published the entirety of my development as open-source software at github.com/nh2.

I also hope that the Ubierhouse data set published along with this thesis will facilitate further research into building-scale reconstruction, and that it will spark a tradition of evaluating the reconstruction error of established and new systems.

HouseScan has shown that architectural constraints are useful for creating large-scale models from reconstructions of independent components. The presented way of assembling a building from individual rooms adds a new method to the existing work for floor plan reconstruction (2.1.1), and improves upon it by giving immediate feedback for individual room scans.

Finally, it is now possible to create highly detailed, coloured triangle models of any building whose rooms can be scanned with *KinectFusion* – in short time and at low cost.

8 Appendix

8.1 A *HouseScan* user manual

This section is to give a short overview on how to use the 3D fitting and pose optimisation program provided alongside this thesis.

The program is written in Haskell using OpenGL. Orientation in the 3D environment is done using the mouse; there are keyboard shortcuts for most tasks and all other actions can be performed with textual commands.

The program is run from a *REPL*[28] of a Haskell interpreter. This allows full programmatic access to the program if the user desires so. However, knowledge of Haskell is not required, and all necessary commands are detailed in this manual.

At the time of writing, *HouseScan* has only been tested on Linux.

8.1.1 Prerequisites

An installation of GHC 7.6 or 7.8 is assumed.

Required libraries can be installed using `cabal install --only-dependencies` from the project directory.

8.1.2 Startup, keys and commands

The program is started using `ghci Main.hs`, leaving us in a Haskell interpreter ready to receive commands.

The GUI is spawned with the command `restart mainState`, displaying an empty 3D coordinate system.

A room can be loaded by the command

```
run $ \s -> void $ loadRoom s "/path/to/scans/room1/"
```

where the directory `/path/to/scans/room1/` should contain the files

- `cloud_downsampled.pcd`, containing the downsampled point cloud
- `planes.txt`, containing the plane equations, and
- `cloud_plane_hull0.pcd`, `cloud_plane_hull1.pcd` etc., containing the plane boundaries

all of which were created by the command line program for plane detection, such as:

```
detect_planes room.pcd --num-planes 15 --max-iterations 1000
              --distance-threshold 0.05 --optimize_coefficients true
              --downsample_size 0.10
```

[28] REPL: Read–eval–print loop, the interactive part of a language interpreter.

The loaded room's points and planes are displayed.

Interactions with the program are performed by the following keys and commands. Keys are case-sensitive, so upper-case keys are created by using the `Shift` key. Especially important actions are highlighted in bold.

- Dragging with the left mouse button translates the 3D view.
- Dragging with the right mouse button rotates the 3D view.
- The mouse wheel controls the zoom factor.
- A middle click spawns the **options menu**, containing settings for
  - wall thickness,
  - whether moving affects rooms or single planes, and
  - how far one move step key press shall move a selected object.
- A wall or room can be **selected** by clicking on it.
- Commands working on multiple objects are performed by clicking the objects in order and then issuing the desired command.
  For this, the objects clicked on are added to the current *selection list*.
- If there are more objects in the *selection list* than the command can work with, a helpful error message is printed into the interpreter, stating the reason for the problem.
- Pressing `Space` clears the *selection list*.
- Clicking on objects prints information about the objects into the interpreter.
- The keys ↑, ↓, ←, → can move the selected wall or plane around. `PageUp` and `PageDown` controls its altitude.
- `p` turns displaying points on/off.
- `d` turns displaying planes on/off.
- `+` and `-` change the point size.
- Selecting three planes and pressing `c` creates a **corner** by plane intersection.
- Selecting a room and pressing `g` displays **corner suggestions** for this room. They can be activated with a single click.
- Once a room has 8 corners, a **cuboid can be fit** to them by pressing `f`.
- 'a' **rotates** the selected room by 360°, snapping to the coordinate axes if a cuboid has been fit to it. It also automatically guesses the floor plane.
- Connecting two walls of different rooms with a **`Same` constraint** is done by selecting them and pressing `W`.
- For an **`Opposite` constraint**, `w` is used instead. This respects the current `wallThickness` setting.
- The `wallThickness` can be changed by the command

```
run $ \s -> sWallThickness s $= 0.20
```

where `0.20` is the new wall thickness in meters.

- `Ctrl-w` disconnects the constraint between two selected walls.
- Pressing `o` **optimises the position of all connected rooms**.
- The command `run exportAllRoomXfFiles` saves the projection matrices of all optimised rooms to `.xf` files, which can then be used to transform the full resolution point meshes to their final position using the `plyxform` tool, using:

  ```
  plyxform -f projection.xf original.ply > final.ply
  ```

- `s` saves the current state of the program to disk, `l` loads it.
- A `Save as...` functionality is available using the commands

  ```
  run $ \s -> saveTo s   "/path/to/myhouse.bin"
  run $ \s -> loadFrom s "/path/to/myhouse.bin"
  ```

8.2 Ubierhouse floor plan

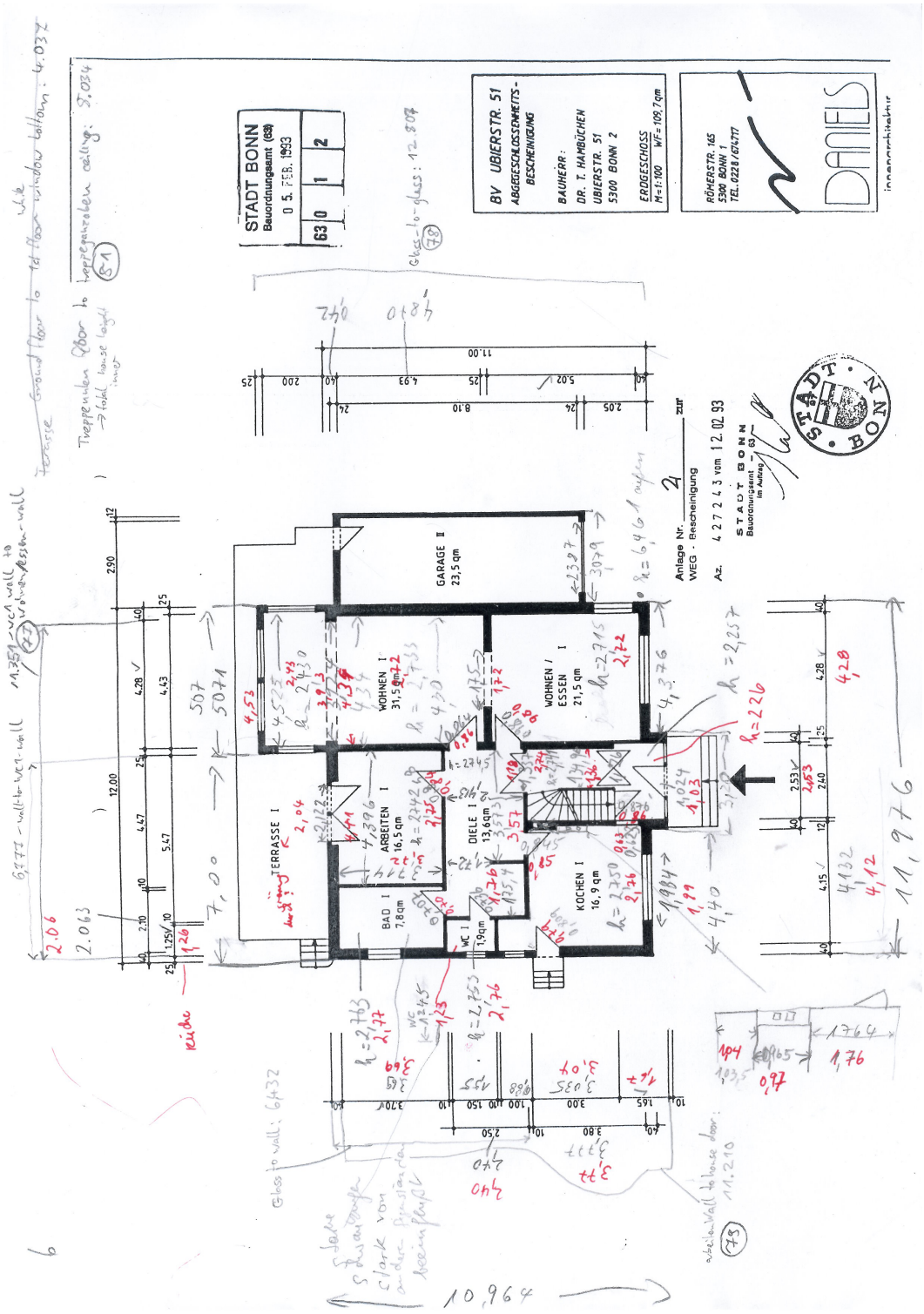

Figure 8.1: Floor plan of the ground floor of the Ubierhouse.

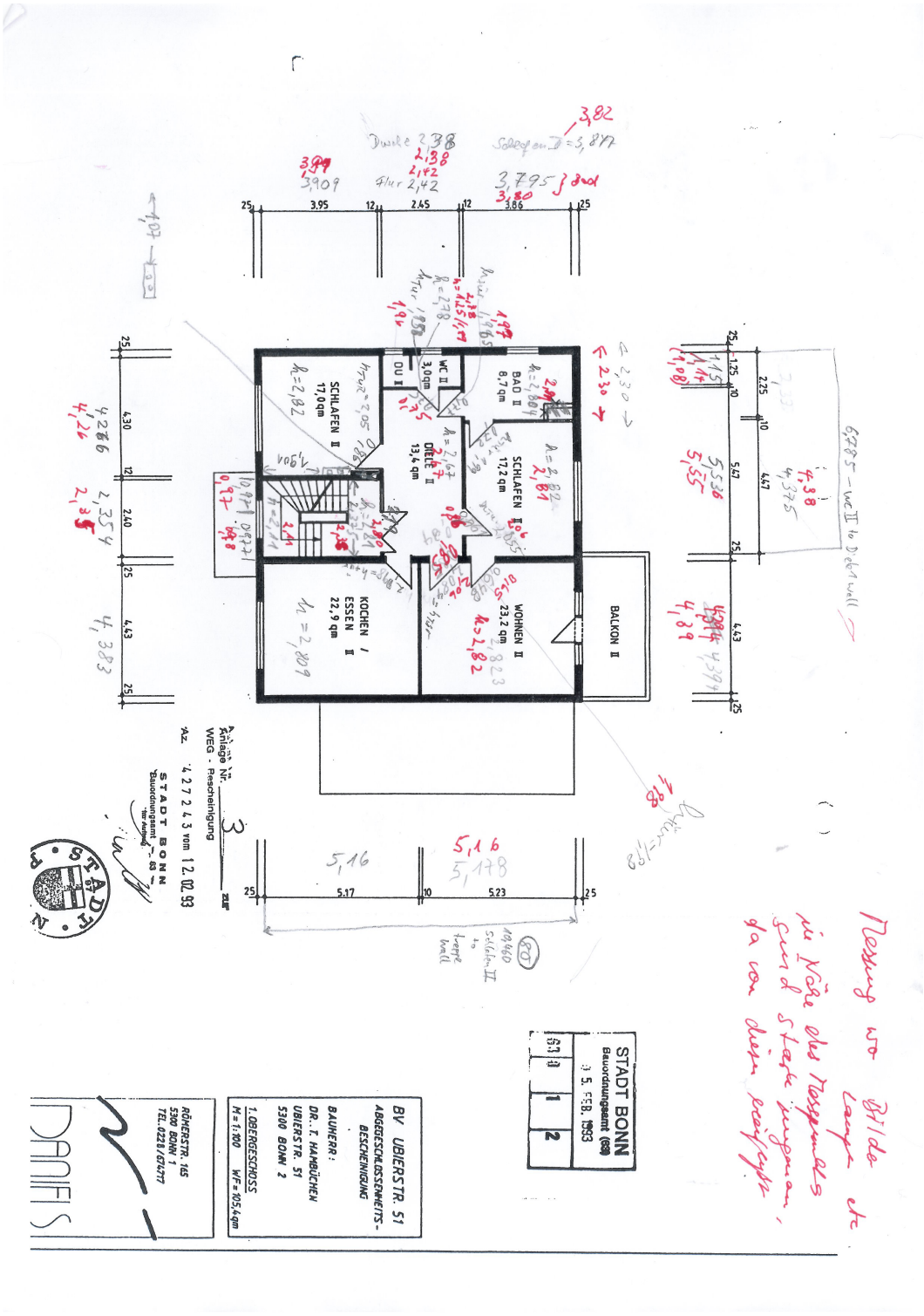

Figure 8.2: Floor plan of the middle floor of the Ubierhouse.

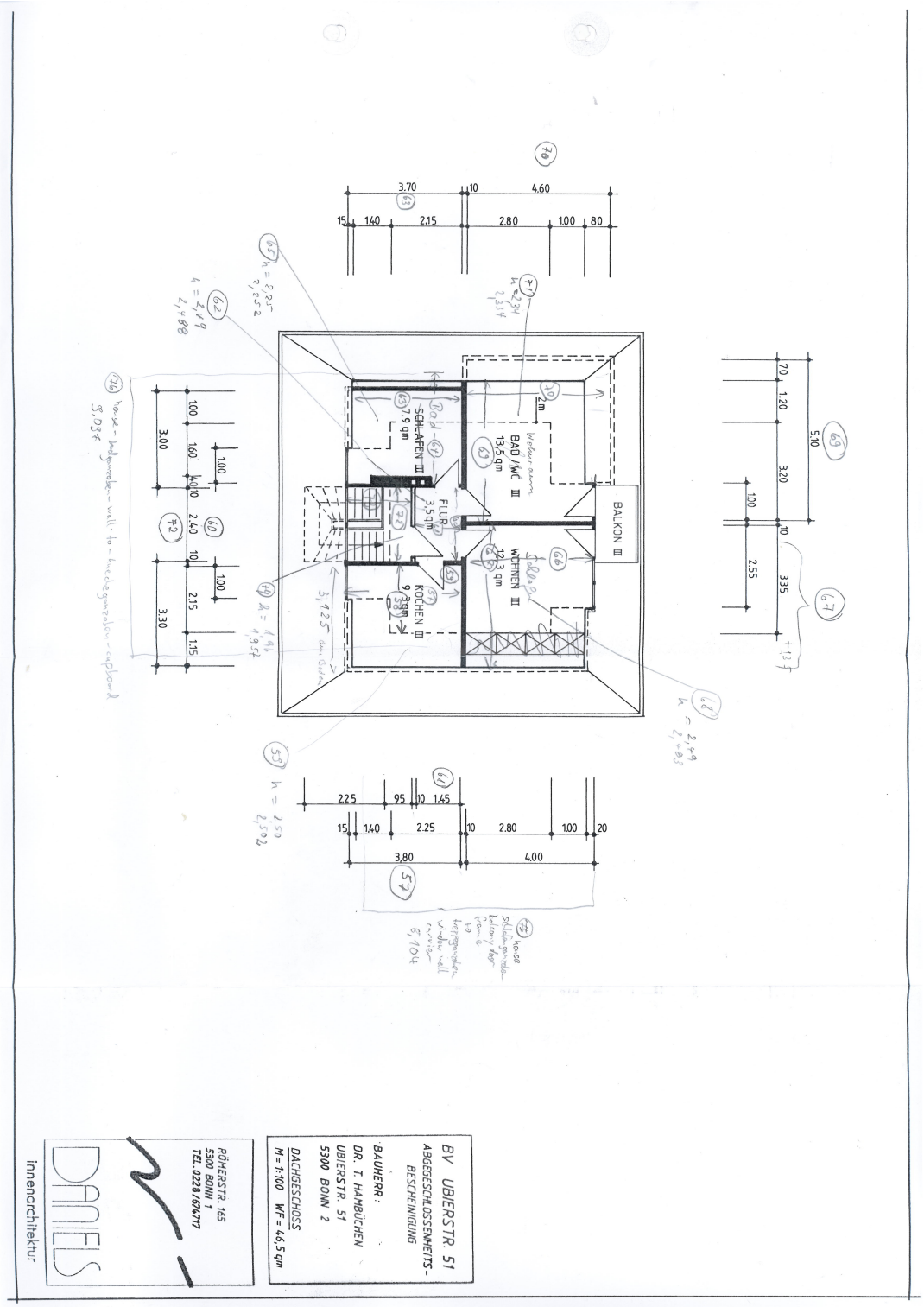

Figure 8.3: Floor plan of the top floor of the Ubierhouse.

8.3 Ubierhouse ground truth and measurements

Ubierhouse ground truth + measurements

room / screenshot file name	3d model	floor plan	laser	sonic	sonic problem	3D rel err.
treppeunten-wall-to-wall-low	2.480	2.40	2.350	0.85	x	0.055
treppeunten-height	2.819		2.740	2.75		0.029
treppeunten-frame-to-frame	3.167	3.00	3.034	3.03		0.044
treppeunten-step-8	1.008		0.969	0.96		0.040
kueche2-above-garden-door	4.126	4.15	4.132	4.12		-0.001
kueche2-hob-wall-to-window-frame	4.126		3.956	3.93		0.043
kueche2-height	2.900		2.753	2.76		0.053
kueche2-window-width	1.819		1.987	0.89	x	-0.085
kueche2-door-windfang-width	0.674		0.623	0.63		0.082
diele1-wall-to-wall-long-high	5.661	5.47	5.401	3.62	x	0.048
diele1-wall-to-wall-short-high	2.667	2.50	2.402	2.41		0.110
diele1-height	2.878		2.736	2.74		0.052
diele1-door-kueche2-width	0.888		0.845	0.85		0.051
wc1-wall-above-mirror	1.609	1.50	1.543	1.54		0.043
wc1-wall-beside-door	1.269	1.25	1.240	1.14		0.023
wc1-height	2.811		2.753	2.76		0.021
arbeiten1-wall-to-wall-over-piano	4.379	4.47	4.400	4.40		-0.005
arbeiten1-wall-to-wall-doors	3.922	3.70	3.715	3.71		0.056
arbeiten1-height	2.854		2.742	2.75		0.041
wohnen1gross-wall-to-wall-over-door	4.239	4.28	4.296	4.28		-0.013
wohnen1gross-long-window-frame-to-do	7.708		7.275	7.26		0.060
wohnen1gross-height	2.840		2.715	2.71		0.046
wohnen1-2-wall-to-wall-doorside	4.329	4.28	4.277	4.27		0.012
wohnen1-2-long-wall-to-wall	4.951	5.02	5.049	5.04		-0.019
wohnen1-2-height	2.907		2.707	2.70		0.074
bad1-long-wall-to-wall	3.922	3.70	3.688	3.69		0.063
bad1-short-wall-to-wall	2.238	2.10	2.060	2.06		0.086
bad1-height	2.838		2.763	2.77		0.027
windfang-long-wall-to-wall	2.667	2.53	2.529	2.53		0.055
windfang-short-door-wall-to-cellar-door	1.745	1.65	1.695	1.69		0.029
windfang-height	2.279		2.260	2.26		0.008
niklas-wall-to-wall-windowside	5.356	5.17	5.152	5.15		0.040
niklas-wall-to-wall-doorside	4.837	4.43	4.394	4.38		0.101
niklas-height	3.023		2.822	2.82		0.071
fluroben-long-wall-to-wall	5.800	5.47	5.529	5.53		0.049
fluroben-short-wall-to-wall	2.529	2.45	2.422	2.42		0.044
fluroben-height	2.789		2.650	2.51		0.052
salon-wall-to-wall-nowindow	4.598	4.30	4.262	4.25		0.079
salon-wall-to-wall-window	4.152	3.95	3.910	3.92		0.062
salon-height	3.019		2.821	2.82		0.070
clara-short-wall-to-wall	4.667	4.43	4.394	4.39		0.062
clara-long-wall-to-wall	5.522	5.23	5.172	5.16		0.068
clara-height	2.947		2.825	2.82		0.043
schlafzimmer-wall-to-wall-nowindow	4.628	4.47	4.375	4.37		0.058
schlafzimmer-wall-to-wall-windowside	4.077	3.86	3.819	3.81		0.068
schlafzimmer-height	2.946		2.809	2.81		0.049
wcoben-wall-to-wall-windowside	1.174	1.25	1.148	1.14		0.023
wcoben-wall-to-wall-nowindow	2.528	2.45	2.378	2.38		0.063
wcoben-height	2.918		2.779	2.78		0.050
badoben-long-wall-to-wall	4.043	3.86	3.796	3.79		0.065
badoben-wall-to-wall-windowside	2.417	2.25	2.299	2.30		0.051

Ubierhouse ground truth + measurements

badoben-height	2.923		2.784	2.79		0.050
treppemitte-wall-to-wall-nowindow	2.454	2.40	2.374	2.37		0.034
treppemitte-wall-windowside-to-doorfran	4.150	3.95	3.943	3.81	x	0.052
treppemitte-height	2.866		2.806	2.81		0.021
kuecheganzoben-wall-to-wall-nodoor	4.461	3.80	4.206	4.20		0.061
kuecheganzoben-wall-doorside-to-cupboa	2.794		2.607	2.60		0.072
kuecheganzoben-height	2.700		2.502	2.50		0.079
flurganzoben-long-wall-to-wall	2.574	2.40	2.385	2.38		0.079
flurganzoben-short-wall-to-doorframe	1.553	1.45	1.468	1.47		0.058
flurganzoben-height	2.553		2.488	2.49		0.026
badganzoben-wall-close-to-shower-to-hou	4.635	3.70	4.267	4.26		0.086
badganzoben-wall-to-doorwall	3.784	3.00	3.682	3.67		0.028
badganzoben-height	2.411		2.252	2.25		0.071
schlafenganzoben-wall-to-wall-doors	4.268	4.00	4.332	4.33		-0.015
schlafenganzoben-wall-to-wall-nowindow	5.458	4.50	5.078		x	0.075
schlafenganzoben-height	2.615		2.483	2.49		0.053
wohnenganzoben-long-wall-to-wall-nodo	5.511	5.10	5.173		x	0.065
wohnenganzoben-short-wall-to-wall-door	4.474	4.00	4.279	4.28		0.046
wohnenganzoben-height	2.441		2.334	2.34		0.046
treppeganzoben-wall-to-wall-nodoor	2.346	2.40	2.380	2.36		-0.014
treppeganzoben-windowframe-to-doorfra	2.433		2.294	2.28		0.061
treppeganzoben-height	2.016		1.957	1.96		0.030
house-schlafenganzoben-balconydoorfran	8.019		8.104			-0.010
house-badganzoben-wall-to-kuecheganzo	9.413		9.097			0.035
house-width-wc1-wohnen1-2	11.347		11.135			0.019
house-length-wohnen	13.283		12.807			0.037
house-length-arbeiten-housedoor	11.846		11.210			0.057
house-length-firstfloor-treppe-schlafen-w	10.950		10.460			0.047
house-height-treppe	8.084		8.034			0.006

9 References

[1] R. A. Newcombe, A. J. Davison, S. Izadi, P. Kohli, O. Hilliges, J. Shotton, D. Molyneaux, S. Hodges, D. Kim, and A. Fitzgibbon, "KinectFusion: Real-time dense surface mapping and tracking," in *Mixed and augmented reality (ISMAR), 2011 10th IEEE international symposium on*, 2011, pp. 127–136.

[2] T. Whelan, M. Kaess, M. Fallon, H. Johannsson, J. Leonard, and J. McDonald, "Kintinuous: Spatially extended kinectfusion," 2012.

[3] "KinFu tree of the PCL source code repository." https://github.com/PointCloudLibrary/pcl/tree/d20de6/gpu/kinfu, accessed: 2014-01-30.

[4] "MeshLab: an open-source 3d mesh processing system." http://meshlab.sourceforge.net, accessed: 2014-06-13.

[5] C. J. Taylor and A. Cowley, "Parsing indoor scenes using rgb-d imagery," *Robotics*, p. 401, 2013.

[6] Y. Zhang, C. Luo, and J. Liu, "Walk&Sketch: create floor plans with an RGB-D camera," in *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, 2012, pp. 461–470.

[7] Y. Furukawa, B. Curless, S. M. Seitz, and R. Szeliski, "Reconstructing building interiors from images," in *Computer Vision, 2009 IEEE 12th International Conference on*, 2009, pp. 80–87.

[8] R. Cabral and Y. Furukawa, "Piecewise Planar and Compact Floorplan Reconstruction from Images."

[9] "OpenNI - Wikipedia." http://en.wikipedia.org/wiki/OpenNI, accessed: 2014-06-16.

[10] A. J. Davison, I. D. Reid, N. D. Molton, and O. Stasse, "MonoSLAM: Real-time single camera SLAM," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 29, no. 6, pp. 1052–1067, 2007.

[11] T. Duckett, S. Marsland, and J. Shapiro, "Learning globally consistent maps by relaxation," in *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, 2000, vol. 4, pp. 3841–3846.

[12] K. L. Ho and P. Newman, "Detecting loop closure with scene sequences," *International Journal of Computer Vision*, vol. 74, no. 3, pp. 261–286, 2007.

[13] M. J. Milford and G. F. Wyeth, "Mapping a Suburb with a Single Camera using a Biologically Inspired SLAM System." http://www.youtube.com/watch?v=-0XSUi69Yvs, accessed: 2014-01-30, 2007.

[14] P. Henry, M. Krainin, E. Herbst, X. Ren, and D. Fox, "RGB-D mapping: Using depth cameras for dense 3D modeling of indoor environments," in *the 12th International Symposium on Experimental Robotics (ISER)*, 2010, vol. 20, pp. 22–25.

[15] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison, and others, "KinectFusion: real-time 3D reconstruction and interaction using a moving depth camera," in *Proceedings of the 24th annual ACM symposium on User interface software and technology*, 2011, pp. 559–568.

[16] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2011.

[17] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," in *ACM Siggraph Computer Graphics*, 1987, vol. 21, pp. 163–169.

[18] "kfusion - GitHub." https://github.com/GerhardR/kfusion, accessed: 2014-06-17.

[19] F. Heredia and R. Favier, "Kinfu Large Scale," *Point Cloud Library*. http://www.pointclouds.org/blog/srcs/fheredia, 2012.

[20] F. Heredia and R. Favier, "Using Kinfu Large Scale to generate a textured mesh," *Point Cloud Library*. http://pointclouds.org/documentation/tutorials/using_kinfu_large_scale.php.

[21] H. Roth and M. Vona, "Moving Volume KinectFusion." in *BMVC*, 2012, pp. 1–11.

[22] T. Whelan, "Kintinuous 2.0: Real-time large scale dense loop closure with volumetric fusion mapping." http://www.youtube.com/watch?v=D3yYjaLmiqU, accessed: 2014-01-31.

[23] "Discussion about the Kintinuous source code in the PCL forum." http://www.pcl-users.org/kintinuous-td4018943.html, accessed: 2014-06-15.

[24] F. Steinbruecker, C. Kerl, J. Sturm, and D. Cremers, "Large-Scale Multi-Resolution Surface Reconstruction from RGB-D Sequences," in *IEEE international conference on computer vision (iCCV)*, 2013.

[25] C. Kerl, J. Sturm, and D. Cremers, "Dense visual sLAM for rGB-d cameras," in *Proc. of the Int. Conf. on Intelligent Robot Systems (IROS)*, 2013.

[26] S. Rusinkiewicz and M. Levoy, "Efficient variants of the ICP algorithm," in *3-D Digital Imaging and Modeling, 2001. Proceedings. Third International Conference on*, 2001, pp. 145–152.

[27] K.-L. Low, "Linear least-squares optimization for point-to-plane icp surface registration."

[28] N. Gelfand, L. Ikemoto, S. Rusinkiewicz, and M. Levoy, "Geometrically stable sampling for the ICP algorithm," in *3-D Digital Imaging and Modeling, 2003. 3DIM 2003. Proceedings. Fourth International Conference on*, 2003, pp. 260–267.

[29] K. S. Arun, T. S. Huang, and S. D. Blostein, "Least-squares fitting of two 3-D point sets," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, no. 5, pp. 698–700, 1987.

[30] "Singular value decomposition: Intuitive interpretations - Wikipedia." http://en.wikipedia.org/wiki/Singular_value_decomposition#Intuitive_interpretations, accessed: 2014-06-11.

[31] "Singular value decomposition: Animation - Wikipedia." http://en.wikipedia.org/wiki/Singular_value_decomposition#mediaviewer/File:Singular_value_decomposition.gif, accessed: 2014-06-11.

[32] A. Nüchter, J. Elseberg, P. Schneider, and D. Paulus, "Study of parameterizations for the rigid body transformations of the scan registration problem," *Computer Vision and Image Understanding*, vol. 114, no. 8, pp. 963–980, 2010.

[33] J. Poppinga, N. Vaskevicius, A. Birk, and K. Pathak, "Fast plane detection and polygonalization in noisy 3D range images," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, 2008, pp. 3378–3383.

[34] "Hough Transform: Illustration - Wikipedia." http://en.wikipedia.org/wiki/Hough_transform#mediaviewer/File:Hough-example-result-en.png, accessed: 2014-06-17.

[35] D. Borrmann, J. Elseberg, K. Lingemann, and A. Nüchter, "The 3D Hough Transform for plane detection in point clouds: A review and a new accumulator design," *3D Research*, vol. 2, no. 2, pp. 1–13, 2011.

[36] "KinFu tutorial - Pointclouds Documentation." http://pointclouds.org/documentation/tutorials/using_kinfu_large_scale.php, accessed: 2014-06-17.

[37] "Gouraud Shading - Wikipedia." http://en.wikipedia.org/wiki/Gouraud_shading, accessed: 2014-06-08.

[38] "Vertical stripes in the depth image and GMCMode - GitHub issue." https://github.com/OpenNI/OpenNI2/issues/81, accessed: 2014-06-10.

[39] "kfusion Pull Request: Add OpennNI 2 driver." https://github.com/GerhardR/kfusion/pull/4, accessed: 2014-06-08.

[40] R. Fletcher, *Practical methods of optimization*. John Wiley & Sons, 2013.

[41] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, and F. Rossi, "GNU Scientific Library Reference Manual-(v1. 12)," *Network Theory Ltd*, 2009.

[42] "hmatrix - Hackage." http://hackage.haskell.org/package/hmatrix, accessed: 2014-06-08.

[43] "honi - Hackage." https://hackage.haskell.org/package/honi, accessed: 2014-06-15.

[44] "hidapi - Hackage." https://hackage.haskell.org/package/hidapi, accessed: 2014-06-15.

[45] "HID API for Linux, Mac OS X, and Windows." http://www.signal11.us/oss/hidapi/, accessed: 2014-06-15.

[46] "QuickCheck: random testing of program properties - Hackage." https://hackage.haskell.org/package/QuickCheck, accessed: 2014-06-15.

[47] "safecopy - Hackage." http://hackage.haskell.org/package/safecopy, accessed: 2014-06-15.

[48] "reinterpret-cast - Hackage." https://hackage.haskell.org/package/reinterpret-cast, accessed: 2014-06-15.

[49] "How to write fast for loops - Haskell Cafe mailing list." https://groups.google.com/d/msg/haskell-cafe/Ms4sKZBwTtw/uqKpLGnyYacJ, accessed: 2014-06-15.

[50] "V.enumFromTo is only fast for Int, not Word32 - GitHub issue." http://github.com/haskell/vector/issues/21, accessed: 2014-06-15.

[51] C. Done, "Reloading running code in GHCi." http://chrisdone.com/posts/ghci-reload, accessed: 2014-06-10.

[52] Y. LeCun and C. Cortes, "The MNIST database of handwritten digits." http://yann.lecun.com/exdb/mnist/, 1998.

[53] A. Teichman, S. Miller, and S. Thrun, "Unsupervised intrinsic calibration of depth sensors via slam," in *Robotics: Science and Systems (RSS)*, 2013.

[54] Y. Taguchi, Y.-D. Jian, S. Ramalingam, and C. Feng, "Point-plane SLAM for hand-held 3D sensors," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, 2013, pp. 5182–5189.

[55] E. Ataer-Cansizoglu, Y. Taguchi, S. Ramalingam, and T. Garaas, "Tracking an RGB-D Camera Using Points and Planes," 2013.