Walk&Sketch: Create Floor Plans with an RGB-D Camera

Ying Zhang
Palo Alto Research Center
3333 Coyote Hill Rd.
Palo Alto, CA
yzhang@parc.com

Chuanjiang Luo
Ohio State University
2015 Neil Avenue
Columbus, OH
luo.75@osu.edu

Juan Liu
Palo Alto Research Center
3333 Coyote Hill Rd.
Palo Alto, CA
jjliu@parc.com

ABSTRACT
Creating floor plans for large areas via manual surveying is labor-intensive and error-prone. In this paper, we present a system, Walk&Sketch, that creates floor plans of an indoor environment by a person walking through the environment at a normal strolling pace and taking videos using a consumer RGB-D camera. The method computes floor maps represented by polylines from a 3D point cloud based on precise frame-to-frame alignment. It aligns a reference frame with the floor and computes the frame-to-frame offsets from the continuous RGB-D input. Line segments at a certain height are extracted from the 3D point cloud, and are merged to form a polyline map, which can be further modified and annotated by users. The explored area is visualized as a sequence of polygons, providing users with the information on coverage. Experiments have done in various areas of an office building and have shown encouraging results.

Author Keywords
2D/3D mapping, RGB-D cameras, floor plans.

ACM Classification Keywords
H.5.2 Information interfaces and presentation: Miscellaneous.

General Terms
Algorithms, Design, Experimentation, Measurement.

MOTIVATION AND INTRODUCTION
Recent years have observed a dramatic cost reduction of RGB-D cameras (e.g., Microsoft's Kinect [17]) in the consumer market. The cost of a 3D camera is only a fraction of a 2D laser scanner commonly used in robotic mapping and localization. Although RGB-D cameras are often limited in range and viewing angles, the low cost has encouraged research on using these devices for 3D modeling and mapping [3, 11, 13, 19]. In addition, images associated with the depth information provide extra visual features for frame-to-frame matching, which can be used to improve the robustness of mapping. Image understanding can also be used for automatic labeling of objects in the environment. Compared to 2D laser range-finders, mapping using 3D cameras does require much higher cost in computation. However, with the increasing availability and decreasing cost of GPU-equipped laptops, 3D modeling and mapping can achieve real-time performance on mobile platforms [3]. Comparing to dense tracking and mapping with 2D cameras [20, 14, 2], RGB-D cameras can be more robust to lighting and image features due to the additional depth images.

Figure 1. Platform for Walk&Sketch.

Prior research has achieved the capability of the 3D modeling of an indoor environment using an RGB-D camera (e.g., [3]). In this paper, we present a relevant and yet different mapping application that creates a 2D *floor plan* by a person walking through the environment at a normal leisurely walking pace with an RGB-D camera. This application is motivated by the fact that floor plans are important in space planning, navigation, and other location-based services, yet not all buildings have floor plans. Furthermore, remodeling or repartition is common in residential and commercial buildings, which may render an existing floor plan obsolete. Currently the common practice to obtain a floor plan is by manual surveying. This is labor-intensive and error-prone. There is also a new app called `MagicPlan` [27] which creates floor plans using smart phones or tablets, however, such a method would not work for long hallways or multiple rooms. Our application is designed to facilitate the building mapping using technology advances in RGB-D cameras. The resulted floor plan can be modified and annotated by humans, just as those in [27] or [5].

UbiComp '12, Sep 5-Sep 8, 2012, Pittsburgh, USA.

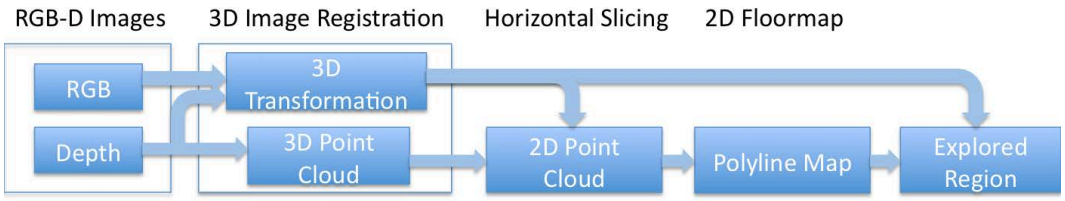

Figure 2. From RGB-D images to 2D floor plan and explored region

Our system, Walk&Sketch, consists of a backpack with an RGB-D camera mounted on the top (Figure 1). The user carries the backpack and walks around the building at a normal strolling pace. As the user walks, the floor plan is incrementally constructed, hence the name Walk&Sketch. The camera is mounted slightly tilted ($\sim$ 5 - 10 degrees) toward the floor. Such a setting results in stable heights frame-by-frame with the floor plane in view from time to time as a horizontal reference. Our method first aligns the reference frame with the floor using the first few frames, i.e., we choose to initially stand in an area so that the camera can see the floor and walls, then computes the frame-to-frame offsets from the continuous RGB-D inputs while walking along the walls in the environment. Instead of creating 3D models as many other research has done [3, 11, 13, 19] using RGB-D cameras, we select a horizontal slice at a certain height and construct a 2D floor plan using connected line segments, i.e., *polylines*. Along with the polyline representation for floor plans, we represent the explored area by a sequence of polygons, which is computed from the resulting polyline map and the trajectory of the camera obtained by frame-to-frame matching. The explored area polygons provide visual feedback and guidance to the user (the person carrying the backpack-mounted camera). The user may decide to navigate an already-explored area to further improve mapping accuracy, or navigate to new areas to further grow the floor plan. Figure 2 shows the overall design of our system, and Figure 3 shows an example of a multi-room floor map and explored areas.

Most existing work in mapping of indoor environments used occupancy grids [4], point clouds [11] and scattered line segments [21], we advocate polyline maps [15], i.e., representing a map using a set of connected linear segments, as used in floor plans. Polyline maps provide a compact and abstract representation and can be easily manipulated by humans using an interactive interface. Polyline maps can also be used for indoor localization in ubiquitous applications [26]. There has been research on getting 2D line sketches from 3D laser scans in cluttered indoor environments [30] or 3D line sketches from vision-based SLAM [6], where lines were considered as features in SLAM computation and the resulting map would be a set of line features. In our case, lines are post-extracted from matched 3D point clouds of the current frame using both RGB and depth images, and grouped as polylines, i.e., a connected set of directional segments. Polylines of the current frame are then merged with the previous polyline map to form a new partial map.

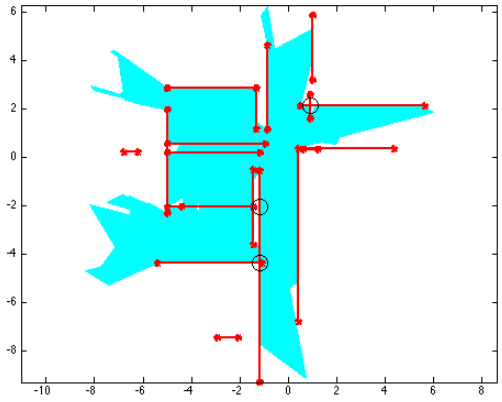

Figure 3. An example of polyline maps: explored area is represented by overlapping polygons and problematic corners circled.

To reduce the effect of noise and increase the computation efficiency, we have used the assumption of orthogonality [21] in indoor environments. Such an assumption has been used in previous work such as [10, 15] with promising results. The rectification greatly improves the quality of the resulting map with simple computation for map merging.

A prototype has been implemented in Matlab using pre-recorded RGB-D images saved at 10Hz from a Microsoft Kinect, with walking speed (0.5 meters per second). We have tested the system in various areas of our office building: a conference room, hallways, and a set of connected offices (Figure 3). The results are encouraging.

The major contributions of this paper are as follows: 1) Design and development of a system that efficiently creates a 2D floor plan by walking and taking videos from a consumer RGB-D camera, 2) design and implementation of a polyline extraction algorithm from 3D point clouds, and 3) design and implementation of explored area representation by a sequence of polygons.

In the subsequent sections, we first describe the method of 3D image registration, followed by the algorithms of polyline map generation and explored area computation. We then show some results from experiments in an office environment. Finally, we conclude the paper with some future directions.

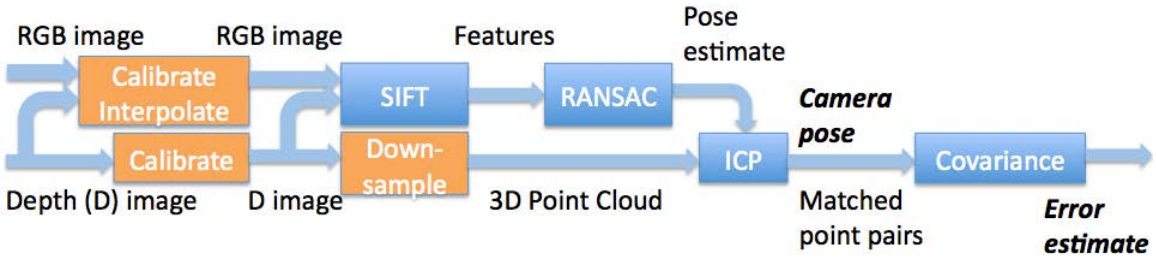

Figure 4. Preprocessing (yellow blocks) and local matching (blue blocks): from RGB-D images to camera pose and error estimates

OBTAIN CAMERA TRAJECTORY FROM RGB-D IMAGES

There has been active research on 3D modeling and mapping using RGB-D cameras. The approach we developed is similar to that in [11], except our uncertainty modeling for local matching fits better for ambiguity situations. Our method consists of three steps: *preprocessing*, *local matching* and *global matching*. Preprocessing handles RGB-D image un-distortion and alignment, and 3D point cloud down-sampling. Local matching applies to two consecutive frames to find a consistent offset between them. Global matching seeks to improve alignment accuracy by registering two key frames far apart in time yet sharing similar image content. The reoccurence of image content is due to revisits, for instance, the camera returns to a position that it has viewed before. In this case, a matching pair of key frames are detected, and we apply an error distribution method to improve the 3D image registration results. Loop closure is performed to generate a globally consistent map.

Preprocessing

Kinect consists of an RGB camera, an infrared camera, and an infrared projector [17]. The distortion from the Kinect sensors is non-negligible. We have used the method implemented in [1] to calibrate the cameras and use the calibration result to undistort both RGB images and depth images. Then we apply the following method to align the RGB image with the depth image: The stereo configuration gives the relative pose between the RGB camera and the infrared camera so that mapping a point in the depth image to the RGB image is straightforward. Nevertheless, mapping a point in the RGB image back to the depth image needs costly search operations. To avoid this complication, for every point in the depth image, we find its corresponding location in the RGB image and calculate its RGB value by interpolation, i.e., for each point in the depth image, we get its RGB values, resulting a pair of RGB-D image.

Kinect's depth image is obtained by correlating the image from the infrared camera with the pattern projected from the infrared projector. The correlation procedure is far from perfect, especially in regions with depth discontinuity. In particular, suppose two adjacent points a Kinect camera sees are from two different surface, instead of an abrupt depth jump, it will look like there is another plane connecting the two points. In other words, the 3D space looks smoother than it actually is. This artifacts is detrimental to the subsequent matching phase. As such, we have to identify and remove those fake depths whose points lie collinear with the optical center of the infrared camera.

The original RGB-D images from the Kinect are in a resolution of 640x480, or, more than 300,000 points. Computationally, it is challenging to perform frame-to-frame matching such as the iterative closest point (ICP) algorithm on this many point samples in a real-time or close-to real-time system; down-sampling is beneficial. In order to have better accuracy for the matching, instead of down-sampling points uniformly in the image coordinate, we down-sample points according to the depth of the points, i.e., points within the valid range (0.8m- 5m) are down-sampled less, and hence denser samples surviving, than points out of range, because points within the valid range are more accurate for matching. In particular, the size of the down-sample grid varies with the depth of the points. Each down-sampled point takes the average of its depth values within the down-sample grid. Down-sampling rates vary from frame-to-frame. After down-sampling, we normally get from 1000 to 20000 points for ICP and 2D polyline extraction.

Local matching

Local matching aligns adjacent frames. We take a similar approach as in [11, 3] with some modification. Figure 4 shows an overall flowchart of the preprocessing and local matching procedures. The output of this step is a 6D camera pose $X = (x, y, z, \alpha, \beta, \gamma)$ or its 3D transformation T for the current frame, and a 3D covariance error matrix C for the translation part of the pose estimation, reflecting the uncertainty.

We first extract features from the two RGB images, using scale invariant feature transform (SIFT) [16] descriptors implemented in [28]. SIFT identifies a set of key-points based on local image features that are consistent under illumination conditions, viewing position, and scale. SIFT features are distinctive and are commonly used for image matching. SIFT features between frames are matched using random sample consensus (RANSAC) [23], an algorithm that iteratively estimates the model parameters from a data set containing inliers and outliers. A model is fitted using inliers and then other data are tested against the model fitted and classified again into inliers and outliers. The model with the most inliers will be chosen. However, SIFT and RANSAC are error-prone in indoor environments. For instance, repetitive patterns such as patterns on carpet and wallpaper can

result in ambiguities in matching. To account for repetitive patterns in the indoor environment, we modify RANSAC to be a maximum likelihood algorithm. Specifically, we assume that the speed of the movement of the camera is constant with some Gaussian noise, i.e., if X_k is the pose of frame k,

$$X_{k+1} - X_k = X_k - X_{k-1} + \Delta X \tag{1}$$

where ΔX are random noise with Gaussian distribution. Let the matching score for pose X be $m(X)$. Instead of choosing $\arg_X \max m(X)$, we use $\arg_X \max[m(X)p(X = X')]$ where X' is the predicted pose for this frame according to Eq. 1. In other words, we are using prior knowledge regarding the continuity of motion to eliminate ambiguities in local frame matching.

Using RANSAC, we find the best camera rotation and translation between the two feature sets. While not accurate enough, this gives a good initial estimate. We will then use ICP [12] to refine the estimation. The idea of ICP is that points in a source cloud are matched iteratively with their nearest neighbors in the target cloud and a rigid transformation is estimated by minimizing error between matched pairs. The process iterates until convergence. Because of the partial overlap, we only focus on point pairs within the intersection. There are several variants of ICP. Theoretically, generalized ICP proposed in [25] is the best, but it is also pretty costly. In our experiments, we observed that the point to plane version [24] works well enough. The down-sampled 3D point clouds of the two neighboring frames are used for the marching, starting from the initial estimate from RANSAC. The output of ICP is the final pose estimate and a set of matched points.

While the 3D transformation between frames is our target for local matching computation, it is almost equally important to keep track of the uncertainty in the estimation process. In the global matching stage (the next subsection), we will be detecting loops between far apart frames. When a loop is detected, the cumulative error between two ends of the loop could be significant. To make the two ends meet, the cumulative error need to be distributed to individual frames. This improves the overall mapping accuracy. The uncertainty in the local matching stage gives us an intuitive guideline to how error should be distributed.

The uncertainty of the pose estimation is represented by its covariance matrix. Given the matched point pairs p_j and p'_j, suppose p_j is in the first frame, we approximate its local tangent plane with the k nearest neighbors by principle component analysis (PCA). It gives the normal direction n_1 and two other basis vector n_2, n_3 in the tangent plane. Because in the ICP algorithm, we adopt the point to plane strategy, points can slide arbitrarily in the tangent plane. As a result, we do not have any confidence how well p_j matches p'_j in the tangent plane. The distance between the two local tangent planes provides a quality measure along the normal direction. As such, we calculate the covariance matrix of the match between p_j and p'_j as follows. Let the distances of the k-nearest neighbors of p'_j to the tangent plane of p_j be

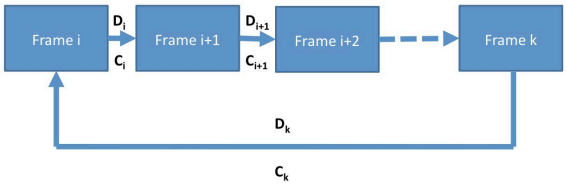

Figure 5. Frame-to-frame constraints along a sequence of frames.

$d_i, i = 1, \ldots, k$, the average of d_i^2 is taken as the variance σ_j^2 in the direction of n_1. To bring the covariance matrix to the coordinate system of this frame, the rotation matrix R_j from PCA needs to be applied.

$$C_j = R_j \begin{pmatrix} \sigma_j^2 & 0 & 0 \\ 0 & \infty & 0 \\ 0 & 0 & \infty \end{pmatrix} R_j^T \tag{2}$$

where $R_j = [n_1, n_2, n_3]$ is the rotational matrix for the plane. To obtain the covariance matrix of the pose estimation between the two adjacent frame, the average of C_j of all matched pairs should be taken. We use the harmonic mean of C_j, as the standard arithmetic average is not feasible due to the ∞ entry in C_j. We have the covariance matrix between the matched frames as:

$$C = \frac{n}{\sum_j C_j^{-1}} \tag{3}$$

where n is the total number of matching points between the two frames and $C_j^{-1} = R_j \begin{pmatrix} \sigma_j^{-2} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} R_j^T$.

Global matching

Due to imprecise pose estimation, over time, error can accumulate, which causes the estimation of the camera position to drift over time, leading to inaccuracies in the map. This is most noticeable when camera moves over a long path, eventually return to the location previously visited. To address this issue, global optimization is needed. We can represent constraints between frames with a graph structure. If a loop is detected, it is represented as constraints between frames that are not adjacent (from frame k to frame i in Figure 5).

We design a two-step global matching process:

- *Key frame identification*: this step involves the identification of *key frames*. Let the first frame be a key frame F_0. At each time after local matching with the previous frame, we also check if this frame can match the previous key frame F_{i-1}. If it matches, continue to process the next frame, if not, set the current frame be the next key frame F_i. We compute the covariance matrix between F_{i-1} and F_i using C_j between the two frames and throw away all the frames between the two to save storage.
- *Loop identification and closure*: after a new key frame F_k is identified, we check if this key frame matches any previous key frames $F_{k-1}, ..., F_0$. If we can identify a match from F_k to F_i, loop closure will be performed on the identified loop (Figure 5).

There are several methods in the literature to do loop closure [18, 7, 8, 9]. However, the full six degree constraint is nonlinear, optimization is expensive. Instead of doing a general optimization as in [9], we have used a simple strategy to do the optimization for the rotational part and the translation part separately, as described in [18]. We argue that for our applications in which most images are walls or rectilinear objects, the main error source comes from the ambiguity in the uncertainty in translations, due to the lack of features, either visually or geometrically. The rotational part is much less noisy than the translation part. We simply distribute the rotational error among the frames evenly to close the loop of rotational offset, using the method described in Section 5.1.3 in [18]. In particular, assume the rotation matrix from key frame i to key frame k be R, which can be transformed into a quaternion $\dot{q} = (q_0, q_x, q_y, q_z)$, with the rotation axis $\mathbf{a} = (q_x, q_y, q_z)/\sqrt{1-q_0^2}$ and the rotation angle $\theta = 2\arccos q_0$. $R_j, i < j < k$, is obtained by setting the angle offset $\theta/(k-i)$, along the axis $\mathbf{a}$, between any two consecutive key frames in the loop.

Instead of evenly distribute the translation error along the frames in the loop, we spread the error according to their covariance matrix, computed in the local matching stage (described in the last subsection). Specifically, if the translational offset from frame j to $j+1$ is D_j with covariance matrix C_j, and from frame k to frame i is D_k with the covariance matrix C_k (Figure 5), we simply minimize the following energy function:

$$W = \sum_{j=i}^{k-1}(X_{j+1} - X_j - D_j)^T C_j^{-1}(X_{j+1} - X_j - D_j) + (X_i - X_k - D_k)^T C_k^{-1}(X_i - X_k - D_k)$$

where X_j is the translation of the frame j. It turns out that this is a linear optimization and $X_j, i < j \leq k$, can be obtained by a simple matrix inversion [18].

The 3D transformation T of the current frame is obtained by combining the rotation matrix R and the translation X. The sequence of transformations $\{T_0, T_1, T_2, ...\}$ of the key frames represents the trajectory of the camera poses. We use the 3D trajectory to compute 2D polyline maps for floor plans.

GENERATE POLYLINES FROM CAMERA TRAJECTORY

Given the transformation of the current key frame T, and the set of 3D points P_c in the current camera frame, we obtain a subset of points corresponding to the plane at height h, i.e., $P_h = \{p \in P_c : |y(T \cdot p) - h| < \epsilon\}$ where $y(p)$ is y value of a 3D point p. We then extract and rectify line segments in P_h, and merge the new set of line segments with the partial map obtained from the previous key frames to form a new partial map. The choice of line segment representation is motivated by convenience of users — human is used to line drawing rather than occupancy grid and point cloud, and such a representation is already used in floor plan drawings. The line segment is also concise, and hence efficient in terms of memory and computation.

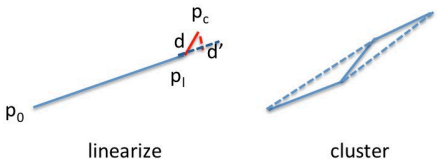

Figure 6. Procedures for segmentation: linearization and clustering.

Segmentation

Given a 2D point cloud, there exist many methods for line extractions [22]. We present here a simple method that applies to an ordered set of points. Segmentation on an ordered set of points is much more efficient than that on an order-less set of points. Given the set of points in a horizontal plane P_h from the current camera frame, we order the points by the viewing angle in the x-z plane, i.e., $p_1 < p_2$ if and only if $x(p_1)/z(p_1) < x(p_2)/z(p_2)$. Let P_o be the ordered set of points P_h in the reference frame, i.e., $P_o = \{T \cdot p : p \in P_h\}$, we get an ordered set of 2D points P in the reference frame by projecting P_o to the x-z plane in the reference frame.

Given the set of ordered points P in the 2D plane, we obtain a set of *polyline* segments. Two consecutive points belong to the same polyline segment if and only if the distance between them is smaller than a given threshold, e.g. 0.3 meters. The goal of linearization is to generate line segments, which are primitives of map construction. There is a balance between two considerations: (1) a line segment is preferred to be long to give a concise representation, and (2) a line segment need to have a small linearization error to maintain high accuracy. Our algorithm is designed to strike a suitable balance, described as follows.

- *linearize:* Start from the first point in the polyline segment. For each point in the segment in the sequence, we extend the line segment as follows. Let $p_0 = \langle x_0, y_0 \rangle$ be the first point in the line segment, and $p_l = \langle x_l, y_l \rangle$ be the previous point and $p_c = \langle x_c, y_c \rangle$ be the current point. Let the distance between p_0 and p_l be l, the distance between p_l and p_c be d and the distance from p_c to the line p_0-p_l be d' (Figure 6 left). The line p_0-p_l will be extended to p_0-p_c, if and only if, $l < l_{\min}$ or $d/d' < K$, where $l_{\min}$ is the minimum length of a line and $0 < K < 1$ is a constant. In our setting, $l_{\min} = 0.3$ meters and $K = 0.1$. If p_c extends the line segment, we have $p_l \leftarrow p_c$ and continue. If p_c does not extend the current line, we start a new line with $p_0 \leftarrow p_l$ and $p_l \leftarrow p_c$ and continue.

- *cluster:* Given a sequence of line segments after linearization, we compute the triangular areas formed by two consecutive lines. Let A_i be the area formed by (p_{i-1}, p_i) and (p_i, p_{i+1}) (Figure 6 right). The two lines can be merged into a new line (p_{i-1}, p_{i+1}) if and only if (1) $A_i = \min_j A_j$, i.e., A_i is the smallest among all other areas A_j formed by two lines at j, and (2) $A_i < A_{\min}$, i.e., it is smaller than the minimum size, in our case, $A_{\min} = 0.1$ square meters. The algorithm iterates until all areas are larger than $A_{\min}$ or there is only one line left in the polyline segment.

After segmentation, we obtain a set of polyline segments from the current 2D point cloud, representing mostly wall segments.

Rectification

We make an assumption that most wall segments in indoor environment are rectilinear [15, 21]. In fact, we position the camera such that it will target at walls most of the times. The rectilinear assumption greatly reduces computation for map merging. We will point out an extension to this assumption at the end of this section.

Unlike previous work on rectification [15, 21], we first use the statistical distribution of the directions of line segments to compute the orientation of the walls using the first few frames, which we are sure are shooting at walls. Given the direction of walls, all line segments of the future frames are then rectified. The details follow:

- *Orientation:* Let $L = \{(l_i, \alpha_i) : i \in 1..N\}$ be the set of N line segments where l_i is the length and $-\pi < \alpha_i \leq \pi$ is the angle of the i'th segment, respectively. We compute the wall direction $\alpha = \arctan(s, c)/4$, where $s = \sum_i w_i \sin(4\alpha_i)$, $c = \sum_i w_i \cos(4\alpha_i)$ and $w_i = \frac{l_i}{\sum_j l_j}$. In other words, the wall direction $\alpha \in (-\pi/4, \pi/4]$ is the weighted average of the normalized directions in $(-\pi/4, \pi/4]$ of all line segments in this frame. To verify if this frame consists of mostly wall segments, we compute variance $\sigma_s^2 = \sum w_i(\sin(4\alpha_i) - \sin(4\alpha))^2$ and $\sigma_c^2 = \sum w_i(\cos(4\alpha_i) - \cos(4\alpha))^2$. If we have large σ_s or σ_c, either we have a noisy frame or wall segments are not rectilinear. We will use first few frames to compute the wall orientation and use value α that has minimum uncertainty, i.e., smallest variances, as the result.

- *Rectification:* To make sure that the frame consists of mostly wall segments, we compute the wall orientation for each frame. If the orientation differs a large amount from the wall orientation α, the frame will not be used for map merging; otherwise, we rotate the current frame by $-\alpha$ to align with the walls. The set of line segments in the rotated frame will then be rectified to one of the rectilinear angles, $\{0, \pi/2, \pi, -\pi/2\}$, that is closest to its direction.

The result of rectification is a set of rectified polyline segments. We will then merge the set of polyline segments with the partial sketch map generated by the previous frames to form a new partial sketch map. In cases where many diagonal or circular walls are present, we can extend the rectification to eight or more directions instead of four. For the rest of this paper, however, we focus only on rectilinear walls.

Map merging

A polyline map is represented by a set of polylines. Instead of merging two polyline maps, we first decompose the polyline maps to four groups of line segments, each in one of the directions $\{0, \pi/2, \pi, -\pi/2\}$. We merge lines that are overlapping in the *same* direction. Finally, we clean up the sketch map by removing small segments and recreate the polyline map from the merged sketch map. The details follow.

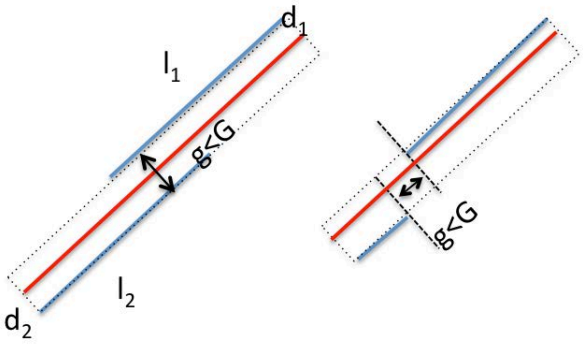

Figure 7. Merging two lines close-by in the same direction.

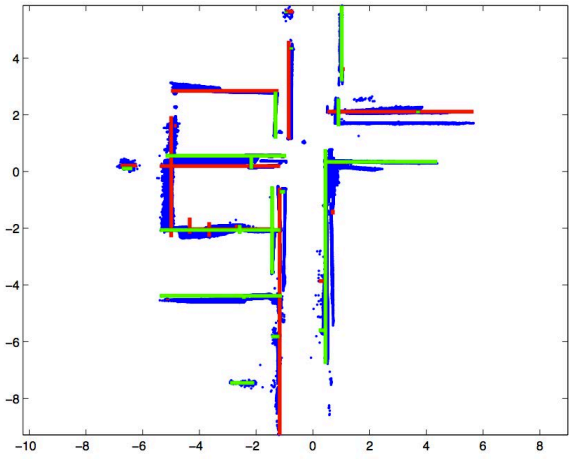

Figure 8. Sketch map: red and green lines are of opposite directions.

- *Sketch merging:* For each of the four directions, we repeat merging two overlapping lines until no lines are overlapping. Two lines are *overlapping*, if and only if, (1) the vertical gap between the two lines are less than G, and (2) either the line are overlapping or the horizontal gap between the lines is less than G. In our case, G is 0.2 meters. Given l_1 and l_2 the the lengths of two overlapping lines in the same direction, the merged line l is parallel to the two lines with the two ends at the boundary of the rectangular box formed by these two lines (Figure 7). Let d_1 and d_2 be the distances from the merged line to l_1 and l_2 respectively, we have $d_1 l_1 = d_2 l_2$. In other words, the merged line is closer to the longer line. Figure 8 shows an example of a sketch map out of the 2D point cloud (blue dots), where red and green show lines of opposite directions.

- *Intersection identification:* Due to noise in images and approximation in processing, we will get sketches with lines crossing each other in different directions (Figure 9 left). We identify three types of intersect, X-Y, Y-X and *bad* intersect (Figure 10), where X-Y intersections start from X-line and end at Y-line, Y-X intersections start at Y-

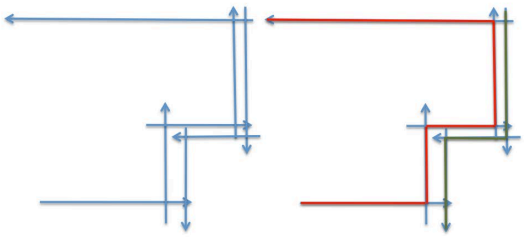

Figure 9. Sketch and correspondent polylines.

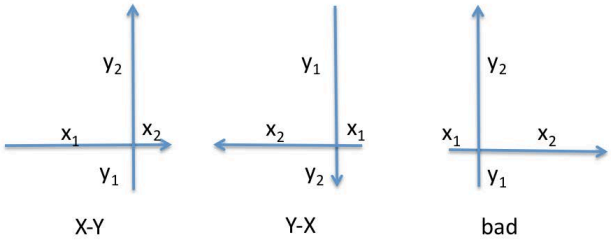

Figure 10. Intersection types for sketches.

line and end at X-line, and bad intersections are invalid, resulted from noisy images and misalignments in trajectory matching. In particular, an intersection type is X-Y if and only if $x_2 < \epsilon$ and $y_1 < \epsilon$, it is Y-X if and only if $x_1 < \epsilon$ and $y_2 < \epsilon$. One important feature of this work is to identify those bad intersects so that human users can adjust the map afterwords.

- *Polyline extraction:* Start from any valid intersect P of lines l_x and l_y – without loss of generality, we assume it is of type X-Y – we do forward search as follows. An intersect P' of l'_x and l'_y follows P, if and only if, P' is of opposite type, i.e., Y-X, $l_y = l'_y$, and P' is the first intersect along the direction of Y. Similarly, we search backward to find P'' of l''_x and l''_y, which precedes P, if and only if, P'' is of type Y-X, $l''_x = l_x$, and P'' is the first intersect along the opposite direction of X. We mark all the intersects used in this search and create a polyline of all the intersects in the chain. The process continues until no more valid intersects exist. We then add back all unused lines, which do not have any intersects with other lines. The result of this process produces a set of polyline segments which will be viewed as a partial map at the time (Figure 9 right). The corresponding polyline map of the sketch map in Figure 8 is shown in Figure 3.

In implementation, we keep both the partial sketch map and the partial polyline map, so that the sketch map will be merged with the new sketch produced from frame to frame and polyline map can be viewed in real time.

OBTAIN EXPLORED AREA FROM MAP AND TRAJECTORY

Explored area is defined to be the area swiped by a sensor along its motion trajectory. In exploration or search and rescue situations, it is important to obtain the explored area so

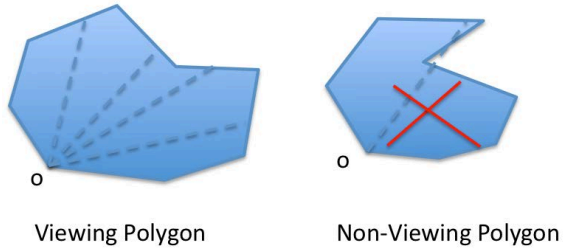

Figure 11. Viewing polygon vs. non-viewing polygon.

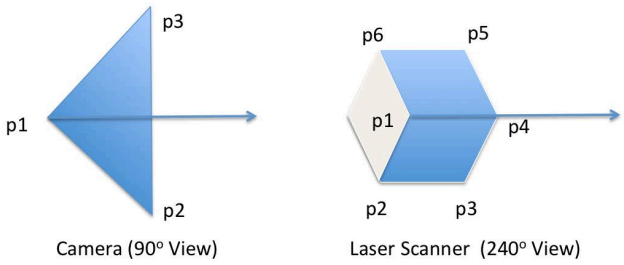

Figure 12. Viewing polygons of sensors.

that it is more efficient to discover the space unexplored. In previous work, the explored area is mostly represented by a grid map, with 0 as empty, 1 as occupied and 0.5 as unknown. Although we can convert a polyline map to a grid map, and compute the explored area by the line of sight of sensor coverage, it will not be efficient. In this work, we compute a *viewing* polygon directly given a polyline map and the camera's position and orientation. We then obtain the explored area by a sequence of viewing polygons along the trajectory.

A viewing polygon is a polygon, such that, (1) there is an origin point O, and (2) there is only one intersection for any line from the origin towards any edge of the polygon (Figure 11). We call any line from the origin a *viewing line*. A viewing polygon will be represented by a sequence of points, starting from the origin. Note that viewing polygons may not be convex. A view of any directional sensor, such as a camera or a laser scanner, can be approximated by a viewing polygon (Figure 12). Given an initial viewing polygon of a sensor, we clip the viewing polygon by the lines in the polyline map that intersect with it and obtain a new viewing polygon that is not blocked by any line. Given any line that intersects with the viewing polygon, there are three cases (Figure 13): (1) both ends are in the polygon, (2) one end in and one end out, and (3) both ends are out. For each in-end, we compute the intersection of the viewing line extended from the in-end to one of the edges of the polygon. For the out-end, we compute the intersection of the line with one of the edges of the polygon. We order all the boundary intersections and in-ends in an increasing order of the viewing angles. Let the initial viewing polygon $\mathbf{P} = (P_1, P_2, ..., P_N)$ be the or-

dered set of N points starting from the origin. The *index* of a boundary intersection point is k if the point is between P_k and P_{k+1}. A new viewing polygon cut by a line is obtained as follows:

- *Both ends in:* there are zero or even number of intersections of this line with the polygon. Let I_1 and I_2 be the two in-ends of the line and B_1 and B_2 are the correspondent boundary points. Let the index of B_1 be k and B_2 be j, $k \leq j$. If there are zero number of intersections (top of Figure 13), the new polygon would be $\{P(1:k), B_1, I_1, I_2, B_2, P(j+1:N)\}$. If there are even number of intersections (Figure 13 bottom), we have to insert additional boundary points and remove the original points that are outside of the line. Let $\{i_1, i_2, ...i_{2n}\}$ be the indexes of the intersection points in order. The points between i_{2k-1} and i_{2k} remain in the new polygon, but the points between i_{2k} and i_{2k+1} are outside of the new polygon.

- *One end in and one end out:* there is one or odd number of intersections of this line with the polygon. Without loss of generality, assume the in-end I is before the out-end, i.e., B_1 has index k and B_2 has index j, $k \leq j$. If there is only one intersection, B_2, the new polygon is $\{P(1:k), B_1, I, B_2, P(j+1:N)\}$. If there are more than one intersections, i.e., $\{i_1, i_2, ...i_{2n+1}\}$, similar to the previous case, the points between i_{2k-1} and i_{2k} remain in the new polygon, but the points between i_{2k} and i_{2k+1} are outside of the new polygon.

- *Both ends out:* there are two or even number of intersections of this line with the polygon. Let B_1 and B_2 be the two intersection points of the line and the polygon. Let the index of B_1 be k and B_2 be j, $k \leq j$. If these are the only two intersections (top of Figure 13), the new polygon would be $\{P(1:k), B_1, B_2, P(j+1:N)\}$. If there are more than two intersections (Figure 13 bottom), we have to insert additional boundary points and remove the original points that are outside of the line. Let $\{i_1, i_2, ...i_{2n}\}$ be the indexes of the intersection points in order. The points between i_{2k-1} and i_{2k} are outside of the new polygon, but the points between i_{2k} and i_{2k+1} remain in the new polygon.

To compute a viewing polygon for a camera pose, the original viewing polygon (Figure 12) will be cut by each of the lines that intersect with it in the current polyline map. Viewing polygons are computed only when the camera moved significantly since the last time a polygon was computed. To calculate the difference between two frames, we use

$$d = \lambda ||x - x'||^2 / D^2 + (1 - \lambda) \sin^2(\alpha - \alpha') \qquad (4)$$

where x and α (x' and α') are the 2D location and heading of the current (previous) frame, respectively. In our case, we have D set to 5 meters, λ is 0.5 and $d > 0.1$ triggers a new polygon computation. A sequence of viewing polygons forms the explored region by the camera. For the example in Figure 3, we get 1000+ key frames, but only 70+ polygons for representing the explored space.

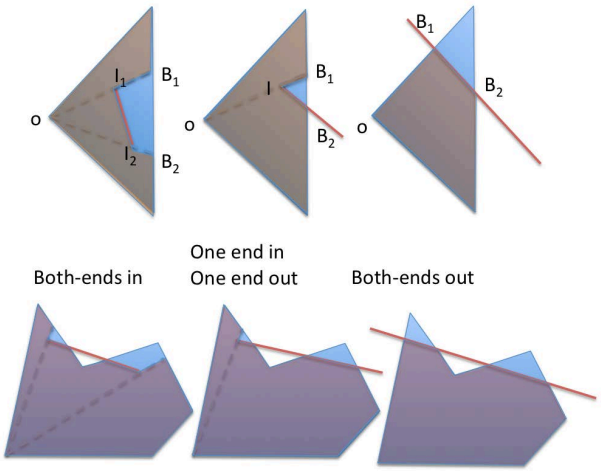

Figure 13. Clipping viewing polygons: top - simple; bottom - complex.

EXPERIMENTS IN OFFICE ENVIRONMENTS

Experiments were performed on various areas of our office building, including a conference room, corridors, and a set of connected offices. Our data were taken at 10Hz in walking speed (0.5m per second) and processed off-line using Matlab. The processing time for generating camera trajectory is about 2-3 seconds per frame on ThinkPad T420 (3M Cache, 2.3 GHz). The time from camera trajectory to 2D polyline map is only 0.007 seconds per key frame, almost negligible comparing to the computation time for camera poses.

Figure 14 shows four additional examples with polyline mapping and explored areas, from top: a conference room, a looped corridor, an open space, and an open space with two connected rooms. The sizes of areas of our experiments so far are under 20 meters by 20 meters.

Kinect cameras after calibration (with undistorted images) offer good accuracy (error within 1% depth values). Image registration or camera pose computation introduces error when the matching frames are lack of visual or geometric features. Line segmentation, rectification and map merging introduce additional error when the image has non-rectilinear lines. Overall, we found that the 2D floor plans generated so far have very good accuracy in measurement scales, less than 0.3 meters in most cases. In addition, our system offers a unique feature that identifies problematic line crossings, which provides areas for human correction after the map generation.

The explored area provides information on holes that need further exploration. In two of the examples in Figure 14, the camera did not cover the space in the middle, so it would be good to take additional images in the middle in order to cover the whole space. It also shows if an area has been explored before so that one can avoid wasting time taking videos of previous explored areas. The efficient representation of the explored areas can be implemented on small handheld devices and used for search and rescue applications.

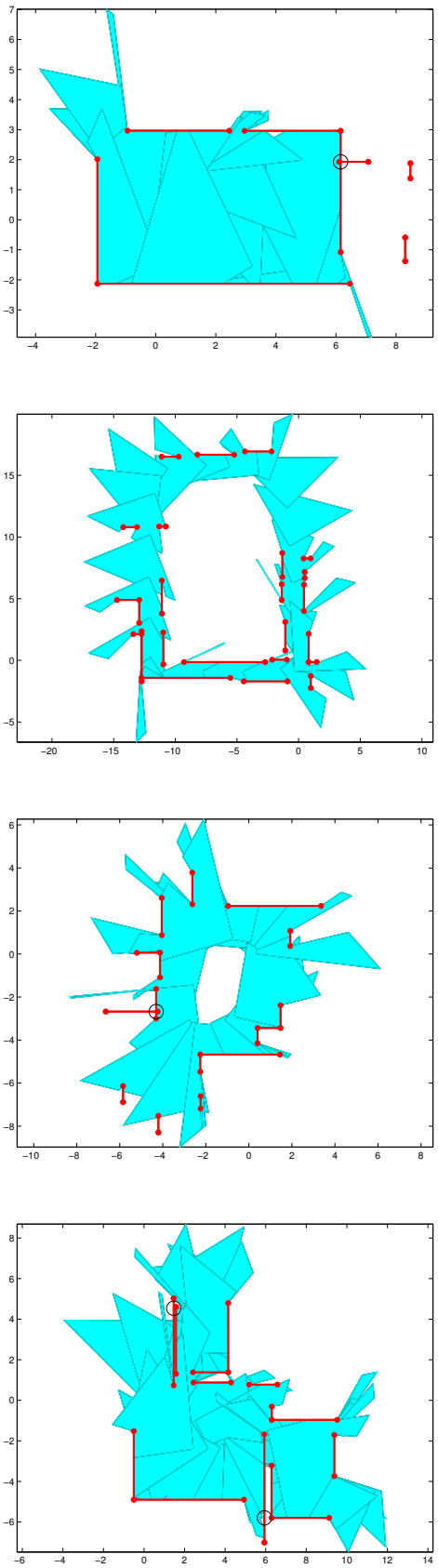

Figure 14. Four mapping examples, from top: a conference room, a looped corridor, an open space, and an open space with two connected rooms.

CONCLUSIONS AND FUTURE WORK

We have presented a system, Walk&Sketch, to create indoor polyline maps for floor plans by a consumer RGB-D camera via walking through the environment. The prototype has a proof of concept implementation in Matlab using saved video images, with processing time about 2-3 seconds per frame. Videos have been taken in various areas of our office building. The resulting 2D floor plans have good accuracy with error less than 0.3 meters.

One challenge to create maps in real-time is to extract reliable camera trajectory in real-time, while 2D floor plans can be already generated in real-time given the camera trajectory. According to the work presented in [3], one can process 3-4 frames in a second if a gaming laptop is used with GPU SIFT implementation [29]. We have used similar and simplified computations as [3], and our implementation is in Matlab, which is known to be several magnitude order slower than a C implementation for iterative operations. We are confident that real-time processing is possible, and this will be one direction for our future work. Another direction is to speedup the image registration algorithm by optimizing or simplifying some of the computations in frame-to-frame matching.

After we have a real-time mapping system, we will work on larger areas to demonstrate the reliability and robustness of such a system. With real-time mapping, we can have interactive procedures similar to [3] to hint the quality of frame-to-frame matching during video taking. For a large building, one may stop and restart, from the same spot, using image matching to the last frame as described in [3]. We will also extend our algorithms to cases where diagonal or circular walls exist for more general floor plans.

We will develop a user interface so that one can view, modify, and annotate maps being generated, similar to what can be done in [5]. We would also like to add features in the environment, such as windows and doors, and objects such as tables and chairs, automatically to the floor plan, using some image classification techniques. In addition to visible features, one can also embed radio fingerprint to the map for localization purpose. Such extensions would make Walk&Sketch more effective and efficient.

ACKNOWLEDGEMENTS

We thank Dr. Craig Eldershaw for putting together a prototype system (Figure 1) for our experimentation. We are also grateful for valuable comments from anonymous reviewers, which helped us improving the quality of this paper. This project was sponsored by Palo Alto Research Center via supplementary research funding.

REFERENCES

1. J.-Y. Bouguet. Camera calibration toolbox for Matlab. `http://www.vision.caltech.edu/bouguetj/calib_doc/`, 2012.

2. A. J. Davison, I. D. Reid, N. D. Molton, and O. Stasse. MonoSLAM: Real-time single camera SLAM. *IEEE*

Transactions on Pattern Analysis and Machine Intelligence, 29, 2007.

3. H. Du, P. Henry, X. Ren, M. Cheng, D. B. Goldman, S. Seitz, and D. Fox. Interactive 3D modeling of indoor environments with a consumer depth camera. In *ACM Conference on Ubiquitous Computing*, 2011.

4. A. Elfes. Using occupancy grids for mobile robot perception and navigation. *Computer*, 22:46 – 57, June 1989.

5. Floor Planner. The easiest way to create floor plans. `http://floorplanner.com/`, 2012.

6. A. P. Gee and W. Mayol-Cuevas. Real-time model-based SLAM using line segments. In *Advances in Visual Computing*, volume 4292. Springer, 2006.

7. G. Grisetti, R. Kümmerle, C. Stachniss, U. Frese, and C. Hertzberg. Hierarchical optimization on manifolds for online 2D and 3D mapping. In *IEEE International Conference on Robotics and Automation*, pages 273–278, 2010.

8. G. Grisetti, C. Stachniss, and W. Burgard. Non-linear constraint network optimization for efficient map learning. *IEEE Transactions on Intelligent Transportation Systems*, 10:428–439, 2009. ISSN: 1524-9050.

9. G. Grisetti, C. Stachniss, S. Grzonka, and W. Burgard. A tree parameterization for efficiently computing maximum likelihood maps using gradient descent. In *Robotics: Science and Systems (RSS)*, 2007.

10. A. Harati and R. Siegwart. Orthogonal 3D-SLAM for indoor environments using right angle corners. In *Third European Conference on Mobile robots*, 2007.

11. 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 *Advanced Reasoning with Depth Cameras Workshop in conjunction with RSS*, 2010.

12. ICP. `http://www.mathworks.com/matlabcentral/fileexchange/27804-iterative-closest-point`, 2012.

13. S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison, and A. Fitzgibbon. KinectFusion: Real-time 3D reconstruction and interaction using a moving depth camera. In *ACM Symposium on User Interface Software and Technology*, 2011.

14. G. Klein and D. Murray. Parallel tracking and mapping for small AR workspaces. In *IEEE Int'l Symp. Mixed and Augmented Reality (ISMAR '07)*, 2007.

15. J. Liu and Y. Zhang. Real-time outline mapping for mobile blind robots. In *IEEE International Conference on Robotics and Automation*, 2011.

16. D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, Nov. 2004.

17. Microsoft Kinect. `http://www.xbox.com/en-US/kinect`, 2012.

18. A. Nchter. *3D Robotic Mapping: The Simultaneous Localization and Mapping Problem with Six Degrees of Freedom*. Springer Publishing Company, Incorporated, 1st edition, 2009.

19. R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. Fitzgibbon. KinectFusion: Real-time dense surface mapping and tracking. In *IEEE Int'l Symp. in Mixed and Augmented Reality (ISMAR)*, 2011.

20. R. A. Newcombe, S. J. Lovegrove, and A. J. Davison. DTAM: Dense tracking and mapping in real-time. In *13th International Conference on Computer Vision*, 2011.

21. V. Nguyen, A. Harati, Agostino, and R. Siegwart. Orthogonal SLAM: a step toward lightweight indoor autonomous navigation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2006.

22. V. Nguyen, A. Martinelli, N. Tomatis, and R. Siegwart. A comparison of line extraction algorithms using 2D laser rangefinder for indoor mobile robotics. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2005.

23. RANSAC. `http://www.csse.uwa.edu.au/~pk/research/matlabfns/Robust/ransac.m`, 2012.

24. S. Rusinkiewicz and M. Levoy. Efficient variants of the ICP algorithm. In *Third International Conference on 3D Digital Imaging and Modeling (3DIM)*, June 2001.

25. A. Segal, D. Haehnel, and S. Thrun. Generalized-ICP. In *Proceedings of Robotics: Science and Systems*, Seattle, USA, June 2009.

26. SenionLab. `http://www.senionlab.com`, 2012.

27. Sensopia. MagicPlan: As easy as taking a picture. `http://www.sensopia.com/english/index.html`, 2012.

28. SIFT. `http://sourceforge.net/projects/libsift/` (contains a DLL library), 2012.

29. C. Wu. SiftGPU: A GPU implementation of scale invariant feature transform (SIFT). `http:///cs.unc.edu/~ccwu/siftgpu`, 2007.

30. O. Wulf, K. O. Arras, H. I. Christensen, and B. Wagner. 2D mapping of cluttered indoor environments by means of 3D perception. In *IEEE International Conference on Robotics and Automation (ICRA04)*, 2004.