Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

3D Depth Sensor on Mobile Devices

Acquisition, Calibration and 3D Reconstruction

Bachelor Thesis

Lukas Bischofberger

lukasbi@ethz.ch

Computer Vision and Geometry Group
Institute for Visual Computing
ETH Zürich

Supervisors:
Dr. Amael Delaunoy
Prof. Dr. Marc Pollefeys

April 26, 2015

Acknowledgements

I would like to thank my supervisor Amael Delaunoy for his support and great help I received during the weekly meetings. I also want to thank Professor Marc Pollefeys for offering me the opportunity to write my bachelor thesis at the Computer Vision and Geometry Group.

Abstract

The introduction of Kinect has made depth sensors available to the consumer world. This made it possible for basically everyone to have a 3D scanner at home. Systems like Kinect Fusion would give the possibility to easily get a 3D reconstruction of everything in front of the lens.

The structure sensor by Occipital is the first mobile depth sensor and thus brings the possibility to have a mobile 3D scanner. We want to show that mobile depth data acquisition is possible and easy and also that this data performs well in 3D reconstruction applications.
Therefore a pipeline from data acquisition on a mobile device up to 3D reconstruction on a desktop computer has been developed.

Contents

Chapter 1

Introduction

We want to use a state of the art mobile device for depth data acquisition. For this we take advantage of the first mobile depth sensor, the structure sensor by Occipital[1]. Further we will combine the depth sensor with the color camera of the device for texturing a 3D model.

We want to show that this data then can be used to create a 3D reconstruction. For this step we will use camera tracking and motion estimation algorithms like semi-direct monocular visual odometry.

1.1 Motivation

We want to show that with a mobile device like a smartphone or a tablet, which today are omnipresent and a fairly low cost sensor we can perform high quality 3D scanning. This scans could be used in numerous applications in daily life.

Scans of rooms can for example be used to measure their dimensions and check for furniture to fit in. This is partly implemented in the IKEA Catalog application whereas there no real measurements and room structure are available[2]. On the other hand such a scan could also be used for real estate advertisements, e.g. by showing reconstructed interiors depicted with real dimensions. In general every reconstructed object could also be printed by a 3D printer. A last intention is also to show that 3D reconstruction with depth data performs better and faster than such with color images only.

[1] https://occipital.com/

[2] https://play.google.com/store/apps/details?id=com.ikea.catalogue.android

1.2 Contribution

The contributions of our work are the following:

- An iOS application for the depth and color data acquisition which is based on the Structure SDK[3]. It allows viewing and recording of RGB-D[4] data as well as camera calibration.
- An offline pipeline which consists of two parts. The first part will convert the raw data recorded on the mobile device to RGB-D pairs using the calibration results. The second part estimates the camera motion and performs a 3D reconstruction.

1.3 Related work

There have been different approaches on reconstruction, ranging between professional solutions, to consumer accessible products up to research projects. They also differ on their technical approach. NextEngine[5] for example uses high precision laser scanning and KinectFusion uses its proper sensor which is a combination of different cameras. The third approach described here solely relies on color images.

1.3.1 Kinect Fusion

Kinect Fusion [**?**] is an application which provides 3D scanning and model creation based on the Kinect sensor[6] The Kinect sensor is a horizontal bar which features a RGB camera, a depth sensor and a multi-array microphone[7]. The user can paint a scene with the Kinect camera and simultaneously see, and interact with, a detailed 3D model of the scene. Kinect Fusion runs on a Desktop PC with the sensor attached by wire. The reconstruction and estimation of the camera movement is based on the tracking of the camera through the depth map. Further there exist possibilities to color the constructed models or to apply shading.

[3] Software development kit
[4] RGB color space with an added depth channel
[5] ...
[6] https://msdn.microsoft.com/en-us/library/dn188670.aspx
[7] http://blog.seattlepi.com/digitaljoystick/2009/06/01/e3-2009-microsoft-at-e3-several-metric-tons-of-press-releaseapalloza/

1.3.2 Live Metric 3D reconstruction on Mobile Phones

A paper by Tanskanen et al. [1] proposes a complete on-device approach for 3D reconstruction. It uses a monocular hand-held device without any depth sensor. The reconstruction process is solely based on color images but the camera motion estimation takes advantage of the device own inertial sensors. Furthermore the paper proposes an efficient and accurate multi-resolution scheme for dense stereo matching which takes into account the restricted computational capabilities of the mobile device.

1.3.3 3D scanner

In the industry products like NextEngine 3 Scanner[8] have been available for some time. This scanner has a laser for high precision scanning and thus obtains far higher quality models than the approaches described before. It also features export to industry standard applications and allows easy printing of the scans. On the other side special equipment is needed which is considerably more expensive than consumer electronic devices.

Our approach is different insofar that we use generally available hardware as the iPad's color camera and an affordable depth sensor. However our data is quite similar to that [**?**] with the difference that the data capturing is entirely done on a mobile device. In theory the reconstruction could also be done on the mobile device as described in chapter 5.

[8] http://www.nextengine.com/

Chapter 2

Data Acquisition

In this chapter we give a detailed description on how the data acquisition on a mobile device is performed. In a first step we describe the use of the structure sensor, it's mechanism and how we obtain the data in a meaningful way. Then we describe the relation between the depth data and color images taken from the device camera. In a third step we will show how to perform camera calibration and combine the data to RGB-D image pairs.

2.1 Mobile Depth Acquisition

The structure sensor is a so called structured-light 3D scanner. It possesses an infrared structured light projector and a corresponding camera system[1]. The projector projects a infrared mesh onto the scene, which the camera records from a slightly different angle. The three-dimensionally shaped surface produces a distorted mesh relative to the viewing camera which is then able to perform a geometric reconstruction of the surface shape.

The sensor is connected to the device by cable, in our case to the iPad through the lightning port. We use the Structure SDK[1] as an interface between our application and the sensor hardware. This allows us to receive depth frames or infrared images at different scales. Unfortunately the maximum resolution of the depth sensor is only 640x480 pixels. Additionally, the SDK provides the possibility to get synchronized infrared/color or depth/color pairs. The color images are taken by the mobile device's own camera, e.g. the iPad's camera through the AVFoundation framework[2].

Originally we wanted to perform the mobile data acquisition on an Android device. This had the advantage that we could rely on standard libraries like OpenNI [3] and OpenCV [4]. The drawback was that applications on Android are

[1] http://structure.io/developers
[2] https://developer.apple.com/library/mac/documentation/AVFoundation/Reference/AVFoundationFramework/ind
[3] openni
[4] opencv


[1]

Figure 2.1: Structure sensor on iPad

written in Java and the C++ libraries are only accessible through the native development kit (NKD) [5] interface. Unfortunately we weren't able to make the structure sensor work on this platform, thus we decided to work on the iOS platform where a SDK for the sensor was provided.

On the data acquisition screen we see the infrared image on the top left, the color image on the top right, the depth map on the bottom left, whereas red/yellow pixels indicate a close location and represent blue/black represent the far away areas. Finally we can also see the surface normals on the bottom right.

2.2 RGB and Depth Camera Calibration

For the camera calibration we make use of the pinhole model which will briefly be explained here. The next section describes how the depth sensor and the device's RGB camera are calibrated. This will provide the camera matrices and lens distortion coefficients for both cameras. After the single camera calibration for both cameras the third section explains how these results can be used to perform a stereo camera calibration, which finds the extrinsic parameters.

[5] ndk

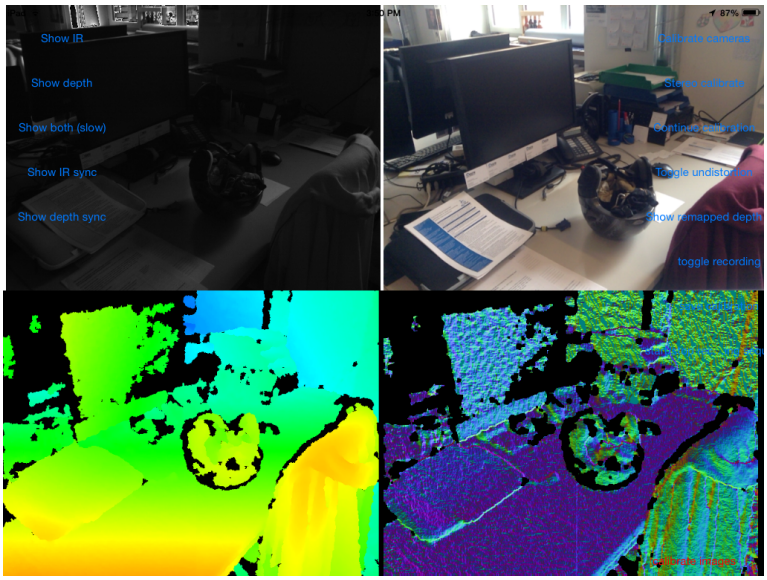


Figure 2.2: Data acquisition screen

2.2.1 Pinhole Camera Model

The pinhole camera model[6] describes the mapping of a 3D point to a 2D image. Equation 2.1 explains the basic relationship between a 3D point and its projection onto the image plane by using perspective transformation.

$$sp = K(RP + t) \tag{2.1}$$

where $P = (X, Y, Z)^T$ is a 3D point in the world coordinate space and $p = (u, v, 1)^T$ is the projection of P in the camera frame in pixels.

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

K is called the camera matrix or a matrix of intrinsic parameters, f_x, f_y are the focal lengths expressed in pixel units and (c_x, c_y) is the principal point which is usually located at the image center. Furthermore we have the rotation matrix

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{2.3}$$

and the translation vector

[6] http://docs.opencv.org/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html

$$t = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} . \tag{2.4}$$

We will refer to $[R|t]$ as the matrix of extrinsic parameters. It is used to describe the camera motion around a static scene in our case. In a more general application it would also describe the rigid motion of an object in front of a stationary camera.

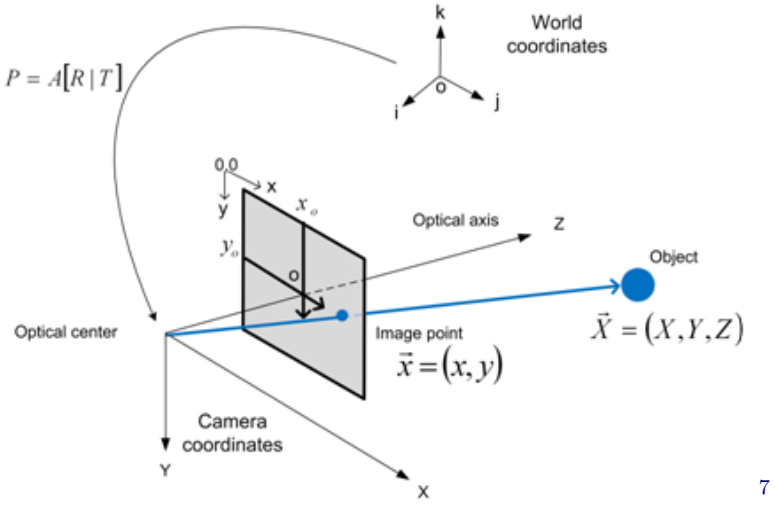


Figure 2.3: Pinhole camera model

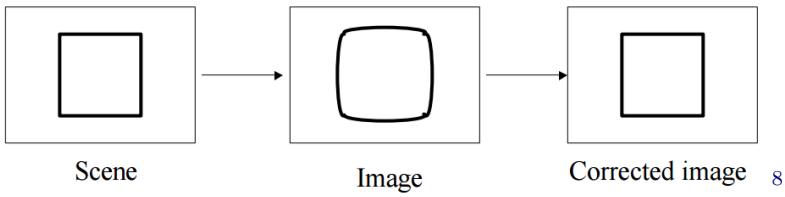
Real lenses usually have some radial and tangential distortion. To account for this the distortion coefficients k_1 to k_6 and p_1 and p_2 are introduced and the camera model is extended as follows.

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = R \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} + t \tag{2.5}$$

$$x' = \frac{x}{z}, y' = \frac{y}{z} \tag{2.6}$$

$$\begin{aligned} x'' &= x' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + 2p_1 x' y' + p_2(r^2 + 2x'^2) \\ y'' &= y' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + p_1(r^2 + 2y'^2) + 2p_2 x' y' \\ &\text{where} \quad r^2 = x'^2 + y'^2 \end{aligned} \tag{2.7}$$

$$\begin{aligned} u &= f_x * x'' + c_x \\ v &= f_y * y'' + c_y \end{aligned} \tag{2.8}$$



[8]

Figure 2.4: Distortion

2.2.2 Single Camera Calibration

We record a set of images of a specific pattern from different viewpoints to do the camera calibration. We choose a chessboard pattern which is a standard choice for calibration because corners can easily be detected. Now a corner detection algorithm can search for a specified amount of corners. Thus we find the projection of the scene points on the image plane. Because we assume a chessboard we know the location of the corners in a respective coordinate system (which is assumed to be the world coordinate system for the calibration). Accordingly we have several images and thus several equations and we can solve for 2.1 and find 2.2, 2.3 and 2.4.

Furthermore the calibration process will also find the distortion coefficients by minimizing

$$f(k_1, k_2, ...) = \sum_i (x'_i - x_{ci})^2 + (y'_i - y_{ci})^2 \tag{2.9}$$

where

$$\begin{aligned} x_c - x_0 &= L(r)(x - x_0) \\ y_c - y_0 &= L(r)(y - y_0) \end{aligned} \tag{2.10}$$

and

$$L(r) = 1 + k_1 r + k_2 r^2 + ... \tag{2.11}$$

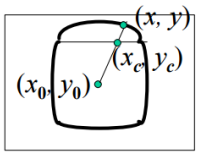


Figure 2.5: Undistortion

We do this calibration separately for the color camera and the infrared sensor. Typically a good calibration has a RMS of the back-projection error of about 0.1 and 1 pixels. We obtained a RMS of around 0.2 pixels for the color camera and a little worse result of about 1.2 for the infrared sensor. We were able to keep the error down by for example fixing the principal point of the image.

We know found the camera matrix and the distortion coefficients for both cameras. In our case this is only an intermediate result, we will continue with the stereo calibration where we can use the single camera calibration results as input.

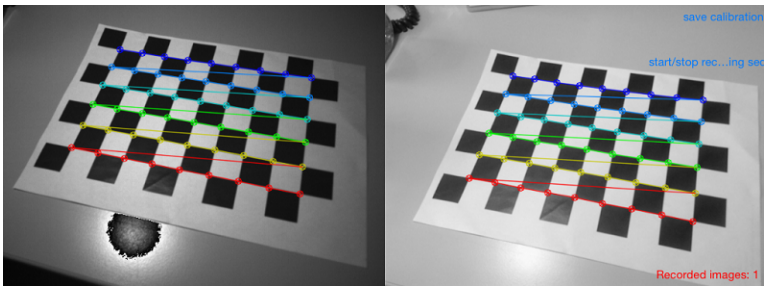


Figure 2.6: IR and RGB images with the calibration pattern

better images

2.2.3 Stereo Calibration

In the stereo calibration we try to estimate the extrinsic parameters. So we want to find the rotation and translation between the two camera centers. We will use

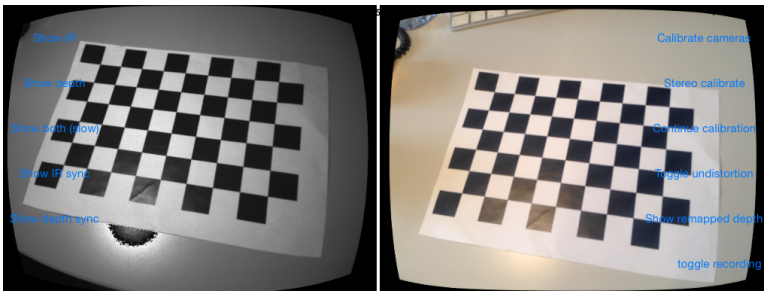


Figure 2.7: Undistorted IR and RGB images

the precomputed camera matrices and intrinsic parameters from the previous section as well as synchronized RGB-D pairs as input.

Stereo correspondence is usually found using epipolar geometry. Two cameras recording the same scene from different viewpoints span the epipolar plane. It is bounded by the line from the camera center c_0 through the projection of p onto the image plane to p and vice versa for camera center c_1. The third bounding line is the connection between the two camera centers through the epipoles e_0 and e_1. Along that we get a pair of epipolar lines, which are the line segments x_0 to e_0 and x_1 to e_1 extended to infinity.

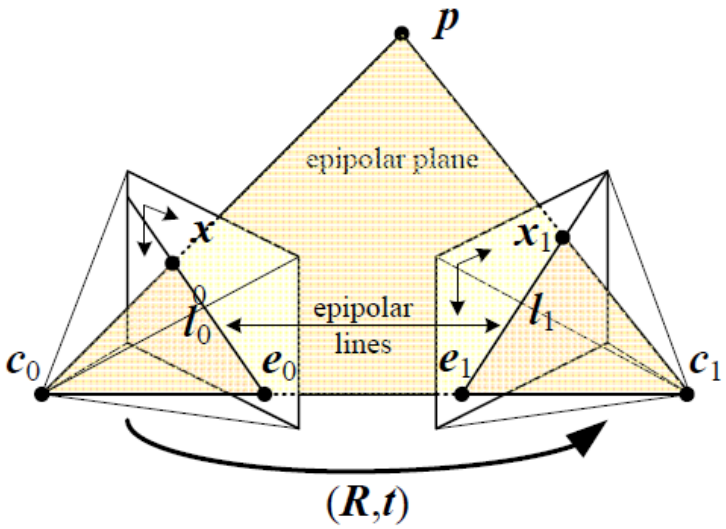


Figure 2.8: Epipolar geometry

The epipolar lines are the intersection of the image planes with the epipolar plane. Then using the epipolar constraint we can define the essential matrix

$$E = [t]_{\times} R \tag{2.12}$$

which can be used to recover the camera motion. The exact procedure is omitted here but using several images it is possible to compute the rotation matrix R and translation vector t [**?**].

The algorithm uses the calibration points to construct the epipolar plane. Furthermore we pass the intrinsics computed in the previous single camera calibration as input to reduce the dimension of the parameter space. Additionally some flags can be passed to the algorithm, e.g. to fix the intrinsics, to further restrict the solution. The quality of the stereo calibration can be controlled by looking at the RMS like in the single camera calibration. Here the error is usually a little higher but values between 0.5 and 1.5 yielded good results.

Todo: side by side of rectified image pair?

2.3 Depth re-projection

The last step in the data acquisition is to use the results obtained in the previous steps and re-project the depth onto the color image. Thus we want to create an RGB-D pair with depth and color information. For storage reasons we will not create a four channel image but use a three channel RGB and a single channel depth image.

Hence now, instead of recording infrared / color image pairs we change to record depth / color pairs. It must be ensured that the sensor is fixed at the same position relative to the color camera as with which the calibration was performed. The algorithm in A.1 is used alongside with the calibration results to project every pixel in the depth frame to its corresponding location in the color camera image plane.

For every pixel p_{depth} this is done as follows, first we project it into its cameras coordinate system using the respective camera matrix (2.13). Then we apply the rotation and translation obtained during stereo calibration to P_{depth} to transform the point into the color camera coordinate system (2.14). And last we back-project the point P_{rgb} into the color camera frame to obtain its pixel coordinates using the color camera matrix (2.15). Besides the z coordinate of the point in the color camera coordinate system represents the depth of p_{rgb} (2.16).

Mathematically this procedure can be formulated the following way:

$$P_{depth} = K_{ir}^{-1} \cdot p_{depth} \tag{2.13}$$

$$P_{rgb} = R \cdot P_{depth} + t \tag{2.14}$$

$$p_{rgb} = K_{rgb} \cdot P_{rgb} \tag{2.15}$$

$$\text{depth of } p_{rgb} = P_{rgb}.z \tag{2.16}$$

(a) Depth (b) Color (c) Re-projected depth

Figure 2.9: Comparison between recorded depth and re-projected depth frame

Because we project the depth values in the color frame we lose some information. E.g. those values which would be projected outside the color frame can't be used anymore, for that reason the back-projected depth frame has black borders. We also see some aliasing effects throughout the entire image due to sampling. This could be solved by a triangulation approach.

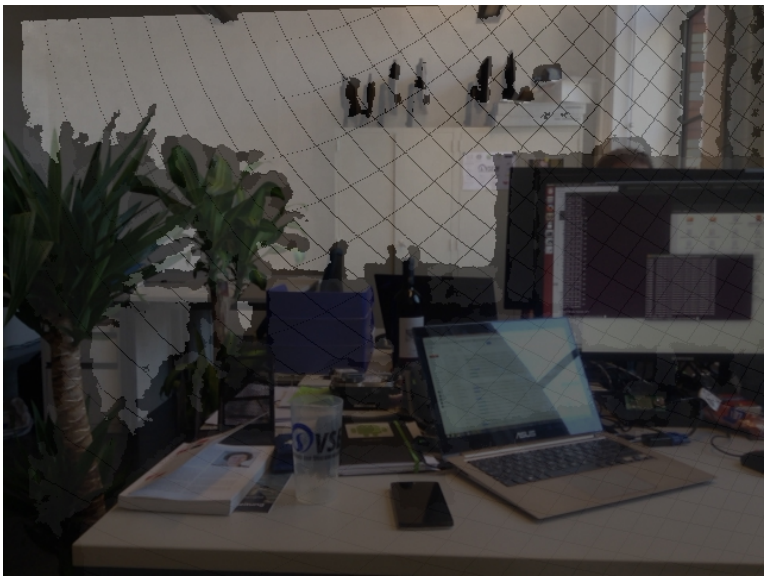


Figure 2.10: RGB-D pair as a 4-channel image

Chapter 3

3D Reconstruction

The first step of the 3D reconstruction is to obtain the camera trajectory. For this matter we use a semi-direct motion estimation approach. This pipeline will be slightly extended to output some more results which will be used in the scale estimation step. Due to the fact that the motion estimation is only based on color images we need this step to correct the discrepancy between the absolute metric of the depth map and the relative coordinates of the motion estimation. The final step is to perform the volumetric 3D mapping and texturing which is done by the fastfusion pipeline.

3.1 Camera Tracking - SVO

For the camera trajectory estimation SVO [**?**] (Semi-Direct Monocular Visual Odometry) is used. This is a pipeline that implements motion estimation in am semi-direct fashion. It uses two parallel threads, one for estimating the camera motion and the second one for mapping.

The motion estimation thread implements the semi-direct approach, which means that it estimates motion directly form intensity values. In a non-direct approach this would be done through robust feature matching. In the first step it initializes the camera pose based on sparse model-based image alignment, it minimizes the photometric error between pixels of different frames projecting to the same 3D location. Then feature-patches are aligned to refine the 2D coordinates of the reprojected points. Finally pose and structure are refined by minimizing the reprojection error introduced in the feature alignment step.

The mapping thread initializes a probabilistic depth-filter for every 2D feature for which the corresponding 3D is to be estimated. When a new keyframe is selected in regions of the image with few 3D to 2D correspondences new depth-filters are initialized. The filters are initialized with a large uncertainty in depth and then updated at every subsequent frame until its uncertainty becomes small enough when a new 3D point is inserted in the depth map.

As mentioned before the pipeline is extended, it now also outputs the $(u, v)^T$

coordinates and the corresponding Z coordinate of every tracked feature. In other words we obtain the 2D feature's coordinates and the estimated depth. Unchanged is the rest of the pipeline which outputs the camera pose for every frame.

Several challenges needed to be overcome when using the SVO, because there were several datasets where no camera poses could be estimated. On one side it was discovered that the device movement largely impacted the performance of SVO. Because of compatibility reasons depth maps were saved as XML files on the iPad which strongly influenced the frame rate at which datasets could be recorded. Thus moving the camera and sensor slowly and steady was required. On the other hand good lighting conditions and a complex scene were also crucial to get good results. The preceding point is especially important because the quality of the recorded image was not high (VGA). Hence higher resolution images could improve the results but might need better hardware.

3.2 Scale Estimation

Up to this point we know the camera pose for every frame and the coordinates of the tracked features over all the frames. The arising problem is that the estimated 3D coordinates of the features are in a coordinate system with a relative scale. On the other hand the depth values recorded by the structure sensor are saved based on an absolute scale, in millimeters. This means the relative coordinates must be scaled to match the absolute coordinates.

That means that for a feature at position $(X, Y, Z)^T$ which is projected to $(u_i, v_i, z_i)^T$ in the camera frame, z_i usually doesn't equal the z value of P_{rgb} in 2.14, here denoted as d_i. Therefore we need to introduce a scale factor which corrects the depth difference. We used a simple least squares approach:

$$\sum_{i=0}^{pixels \cdot images} (d_i - s \cdot z_i) \cdot z_i = 0 \tag{3.1}$$

Subsequently we will scale every camera pose by this factor s. Because the camera pose is computed by

$$c_j = -R_j^T \cdot t_j \tag{3.2}$$

where c_j is the camera center of the j'th camera and $[R_j|t_j]$ is the camera to world coordinate system transformation matrix. Therefore it is enough to scale the vector t_j by a factor of s to account for the false depth.

3.3 Pointcloud

To obtain a visual confirmation that the scale estimation succeeded, the RGB-D pairs were converted to pointclouds. The model from 2.1 is used along with the camera intrinsics to project to the camera coordinate system. Then the rotation and the translation of the camera pose are used as extrinsic parameters and the image is projected to the world coordinate system. Equation 3.3 shows how the pinhole model is reversed.

$$P_i = R_{rgb}^T(K_{rgb}^{-1}p_i - t_{rgb}) \tag{3.3}$$

where p_i is the vector $(u_i d_i, v_i d_i, d_i)^T$. Here u and v are the pixel coordinates in the color camera frame with depth d in the depth frame. Accordingly the color of the pixel with depth d is the RGB value at $(u_i, v_i)^T$.

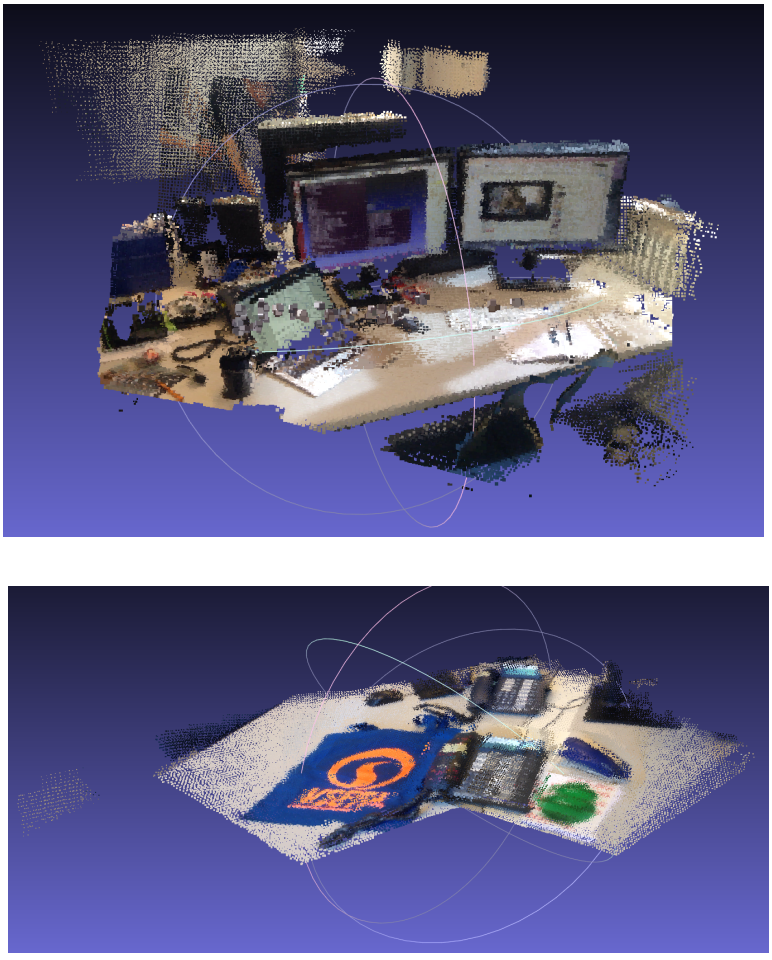


Figure 3.1: Pointclouds in MeshViewer

3.4 Dense reconstruction

The final step of the reconstruction process is the volumetric mapping. Here the fastfusion [2] pipeline is used because it doesn't need a special sensor, as the Kinect Fusion, thus works with our data. Furthermore it runs on the CPU and doesn't need any special GPU, hence it can be run on any desktop computer or laptop.

Fustfusion uses an octree data structure to perform the volumetric meshing and mapping. Moreover it incorporates two different algorithms, the first for data fusion and the second for incremental meshing, each running in their own thread. The geometry of the reconstruction is represented using a signed distance function (SDF) which is saved in an octree. This choice was made because usually most of the space is either free or unknown. The octree is then able to save different distances at different levels in the tree, thus obtaining several resolutions. In the visualization the highest resolution is used and the lower resolutions serve as fall back. Furthermore every node in the tree contains a small cubic volume called brick which itself contains 8^3 voxels. Lastly the voxel stores the distance, weight and color.

In the fusing process the algorithm iterates over all pixels in an image, computes its 3D point and looks up its corresponding brick in the tree. Upon that the distance value in every concerning brick is updated. In the next step the geometry representation as an SDF needs to be converted to a triangle mesh. This type of storage is better because of a lower memory footprint. Here the Marching Cubes algorithm is applied and mesh cells are created which are associated with branches in the tree.

The challenges which were to overcome when using fastfusion were mostly about the data format. For example the image scale needs to be specified, since the default scale is 5000 (1 millimeter is multiplied by 5000), we chose to also use this format. Moreover the pipeline takes the camera center as in 3.2 and the camera rotation as input. Unfortunately this was not clearly specified, besides the rotation needed to be inverted first which resulted in some confusion.

Figure 3.2: reon1

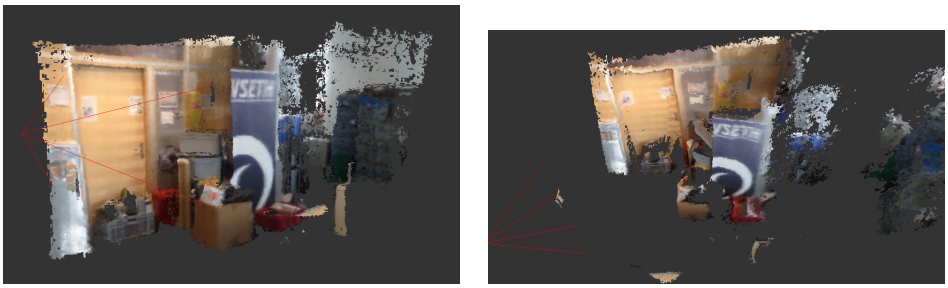


Figure 3.3: reon1

Chapter 4

App integration

Chapter 5

Conclusion

The original idea of this thesis was to obtain data from the Structure sensor on a mobile device, calibrate the sensor and the camera to be able to combine the two inputs and finally perform a 3D reconstruction. Additional ideas were to add model fitting or some body pose estimation and extract the skeleton. And there was the possibility to integrate the depth acquisition process into the application of [1].

Whereas we were not able to acquire data on an Android device this worked smoothly on iOS devices due to the available development kit. The main challenge was to extract the recorded data from the device. Especially for the depth frames this was harder then expected because iOS has a different data security model than Android and doesn't allow to export every data format as easily. The resulting workflow of how to transfer the recorded data from the mobile device to the desktop device showed to be easy but could be improved by using a cloud storage or by an all on device approach like described in 6.

The calibration step was first done offline and ported to the mobile device. Fortunately this worked quite well, the on device calibration has the advantage that the application can easily be run on different devices and there is no additional configuration work to be done offline. Furthermore it has the advantage that the sensor can be moved arbitrarily between different recordings and only a quick calibration needs to be performed to update the configuration. Next, the depth re-projection proofed to be more cumbersome. First there were different challenges with the algorithm itself, like using the closest depth or eliminating the points which projected outside the image plane. Then re-projecting on the device didn't work, at least not real-time, for performance reasons. So this step needed to be performed offline which is unfortunate in particular with regard to the all on device approach.

Regarding the 3D reconstruction pipeline we note that the original intention was to optimistic. This step showed to be the most challenging and time consuming. Originally the dense visual odometry pipeline should have been used because it conforms with the fastfusion pipeline. Due to the fact that this pipeline was

running on ROS[1] the usability was hindered. Therefore it was decided to use SVO instead, the drawback being that the camera motion estimation solely relies on color images and not on depth information. As a result this the camera motion was estimated in some arbitrary coordinate system, whereas the depth coordinates are in millimeters. Therefore some scaling must be performed. The approach taken compares the estimated depth values with the real depth and computes an averaging scaling factor. This is then applied to the camera center translation, in practice this proofed to be accurate enough and good results could be obtained. The other drawback of using SVO was that it doesn't work as well in less textured environments. Thus care must be taken when choosing scenes.

The additional tasks like model fitting and body pose estimation were left out because time constraints wouldn't have allowed a profound commitment to the matter. Instead the application described in [1] was available in iOS and the depth data acquisition could be integrated. This integration nicely completed this work on depth sensors on mobile devices and allowed to show the practicality of mobile depth sensors by enhancing the current application.

Conclusively I can wind up that the original objectives could not all be obtained due to unforeseen challenges and maybe too ambitious prospects. On the other hand different approaches have been tried and therefore and in general a lot about the matter has been learned. Besides, especially the integration of the sensor in an existing application encouraged the enthusiasm for further projects in the field.

[1] Robot Operating System

Chapter 6

Future work

In this chapter we describe possible future work or extensions to this project. Suggestions are based on efforts which didn't succeed, like for example the Android version. Also tasks which were out of scope because of time constraints will be addressed.

At first we wanted to use Android as platform for the Structure sensor. The problem at the time was that not many devices supported USB host mode which was necessary for the sensor to communicate with the mobile device. Now that more devices support the feature it would be worth to again work on data acquisition for Android devices. The other problem with Android was that the integration with OpenNI and the NDK platform proved to be cumbersome, in the future where Occipital is supposed to release additional resources for Android this could also be solved.

The second point concerns the pipeline. For usability reasons it would be great to everything in one application which runs on a mobile device. Therefore the data acquisition and camera calibration could be combined with a mobile visual odometry or a dense visual odometry approach. This could then be extended to also feature mobile reconstruction. In an next step consumer feature could also be added like measuring in the reconstruction or export to 3D file formats to be able to print the recorded object.

An extension to the project which we didn't have time for is model fitting. In the present state everything seen be the camera and sensor is recorded and reconstructed. Still there are cases where you actually only want to record a subset of the viewed scene, e.g. if the background is irrelevant and can be discarded. For that reason we propose to integrate model fitting into the application. The first possibility is to perform a 2D model fitting and then only track and reconstruct the matched region. Another more robust approach is to do 3D model fitting. This has the advantage that there are less features required, thus it works better in less textured environments.

Bibliography

[1] Petri Tanskanen, Kalin Kolev, L.M.F.C.O.S.M.P.: Live metric 3d reconstruction on mobile phones

[2] Petri Tanskanen, Kalin Kolev, L.M.F.C.O.S.M.P.: Live metric 3d reconstruction on mobile phones

Appendix A

Code

A.1 Depth reprojection

```
1  for (int row = 0; row < depthRaw.rows; ++row) {
2    for (int col = 0; col < depthRaw.cols; ++col) {
3
4      // not mapped depth image
5      ptrDepth[depthRaw.cols * row + col] =
6      (uint16_t)ptrOrig[depthRaw.cols * row + col];
7
8      // get depth for remapping
9      double depthVal = (double)ptrOrig[depthRaw.cols*row + col];
10
11     // Map depthcam depth to 3D point
12     Mat_<double> P3D = Mat_<double>(3,1);
13
14     P3D(0) = (col - camera1.at<double>(0,2))
15     * depthVal / camera1.at<double>(0,0);
16     P3D(1) = (row - camera1.at<double>(1,2))
17     * depthVal / camera1.at<double>(1,1);
18     P3D(2) = depthVal;
19
20     // Rotate and translate 3D point
21     Mat_<double> P3Dp;
22     P3Dp = (R*P3D) + T;
23
24     // Project 3D point to rgbcam
25     double xrgb = (P3Dp(0) * camera2.at<double>(0,0) / P3Dp(2))
26     + camera2.at<double>(0,2);
27     double yrgb = (P3Dp(1) * camera2.at<double>(1,1) / P3Dp(2))
28     + camera2.at<double>(1,2);
29     double nDepth = P3Dp(2);
30
31     // "Interpolate" pixel coordinates (Nearest Neighbors)
32     int px_rgbcam = cvRound(xrgb);
33     int py_rgbcam = cvRound(yrgb);
34
35     // Handle 3D occlusions
36     uint16_t &depth_rgbcam =
37     ptrMapped[depthRaw.cols * py_rgbcam + px_rgbcam];
```

```
38      uint16_t &depth_rgbcamScale =
39      ptrMappedScale[depthRaw.cols * py_rgbcam + px_rgbcam];
40
41      if(px_rgbcam - depthRaw.cols < 0
42      && py_rgbcam - depthRaw.rows < 0) {
43        if(depth_rgbcam == 0 || (uint16_t)nDepth < depth_rgbcam) {
44          depth_rgbcam = 5*(uint16_t)nDepth;
45        }
46      }
47    }
48  }
49
```