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 15, 2015

Acknowledgements

I would like to thank my supervisor Amael Delaunoy for his support and great help I received during the weekly meetings. Also I 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.

Contents

Chapter 1

Introduction

We want to use a state of the art mobile device and use it for depth data acquisition. For this we take advantage of the first mobile depth sensor, the structure sensor by Occipital. Further we will combine the depth data with the color images of the device for texturing the 3D model.
We want to show that this data then can be used to easily create a high quality 3D reconstruction. For this step we will use todays camera tracking and pose estimation algorithms like semi-direct monocular visual odometry or dense 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 a daily life. Scans of rooms can for example be used to measure their dimensions and check for furniture to fit in. On the other hand such a scan could also be used for real estate advertisements. A last intention is also to show that 3D reconstruction with depth data performs better and faster than such with color images only.

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. It allows viewing and recording of RGBD data as well as camera calibration.
- An offline pipeline for the recorded data to perform estimation of the camera motion and 3D reconstruction.

1.3 Related work

The following work has already been done in either mobile 3D reconstruction without the use of depth sensors or in 3D reconstruction with depth data in general.

1.3.1 Kinect Fusion

Kinect Fusion[1] is an application which provides 3D scanning and model creation based on the Kinect sensor. 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.

1.3.2 Live Metric 3D reconstruction on Mobile Phones

This paper [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[2] 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 you need special equipment which is considerably more expensive than consumer electronic devices.

[1] https://msdn.microsoft.com/en-us/library/dn188670.aspx
[2] http://www.nextengine.com/

Chapter 2

Data acquisition

In this chapter we give a detailed description on how the data acquisition on mobile devices 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 all the data to RGBD images.

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.


1

Figure 2.1: Structure sensor on iPad

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 provides us with an infrared / color image pair for camera calibration and a depth map / color image pair for the data acquisition process.

Originally we wanted to perform the mobile data acquisition on an Android device because the structure sensor as well as Android support OpenNI. Including the OpenNI libraries, drivers and the OpenCV libraries in the Android NDK turned out to be far harder than expected. Even though the iOS platform is more restricted and it's harder to extract data from the mobile device, it was easier to go with that platform.

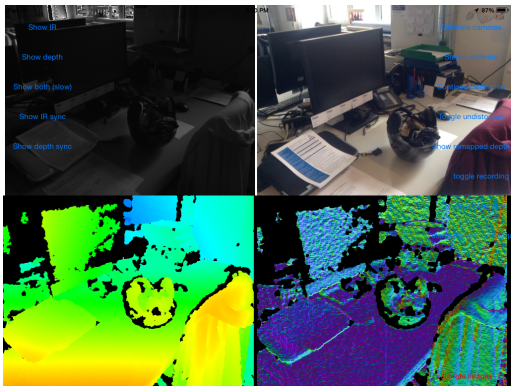


Figure 2.2: Data acquisition screen

2.2 RGB and Depth Camera Calibration

For the camera calibration we make use of the pinhole model. Hence we want to calibrate the sensor's depth camera and the device's RGB camera for their intrinsics. As a result we will find the camera matrices and lens distortion coefficients for both cameras. After having obtained a satisfying calibration for both cameras we can use the computed intrinsics to perform a stereo camera calibration, thus finding the camera extrinsic parameters.

2.2.1 Pinhole camera model[2]

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

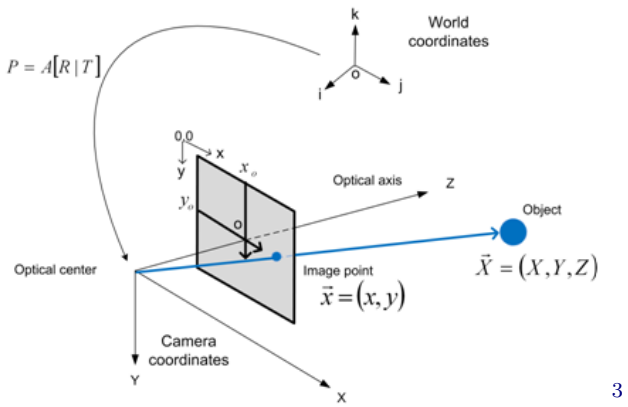
where $P = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$ is a 3D point in the world coordinate space and $p = \begin{pmatrix} u \\ v \\ 1 \end{pmatrix}$ is the projection of P in the camera frame in pixels.

[1] http://structure.io/developers
[2] http://docs.opencv.org/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html

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

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 that is usually located at the image center. Furthermore we have the rotation matrix R and the translation vector t. 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 the camera.

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}, t = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} \tag{2.3}$$



[3]

Figure 2.3: Pinhole camera model

Because real lenses usually have some radial and tangential distortion k_1 to k_6 and p_1 and p_2 are introduced and the camera model is extended as follows.

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = R \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} + t \tag{2.4}$$

$$x' = \frac{x}{z}, y' = \frac{y}{z} \tag{2.5}$$

$$
\begin{aligned}
x'' &= x' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + 2p_1 x' y' + p_2(r^2 + 2x'^2) \\
y'' &= y' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + p_1(r^2 + 2y'^2) + 2p_2 x' y' \\
&\text{where} \quad r^2 = x'^2 + y'^2
\end{aligned} \tag{2.6}
$$

$$
\begin{aligned}
u &= f_x * x'' + c_x \\
v &= f_y * y'' + c_y
\end{aligned} \tag{2.7}
$$

2.2.2 Single Camera Calibration

We perform calibration on a chessboard pattern, say we're looking for a certain amount of corners in a picture. Those corners must be arranged in a certain grid with specific width and height. Furthermore we specify the real distance between two corners in millimeters. When looking at a series of images which all contain the same pattern from different viewpoints we can estimate the distortion coefficients of the lens and the camera matrix.

We can now use the calibrations of both cameras and display an undistorted color image and an undistorted depth map. But we can also use the results of this step for the upcoming stereo calibration as input parameters. We can control the quality of the calibration by looking at the RMS of the back-projection error. It usually should lie between 0.1 and 1.0 pixels. For the color image we usually get something around 0.2 which is quite good and for the infrared image an error of around ??.

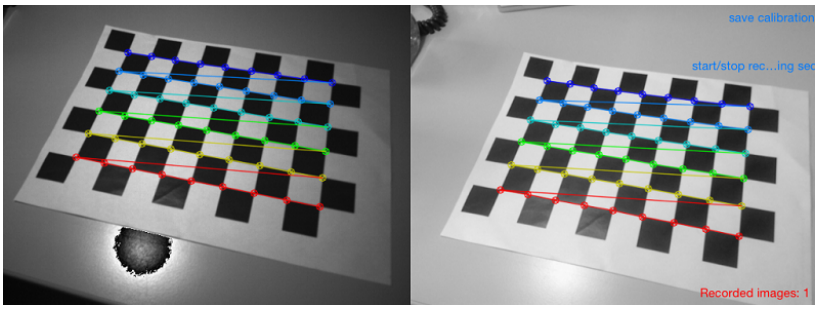


Figure 2.4: IR and RGB images with the calibration pattern

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 the precomputed camera matrices and intrinsic parameters from the previous section as well as synchronized infrared / color image pairs as input.

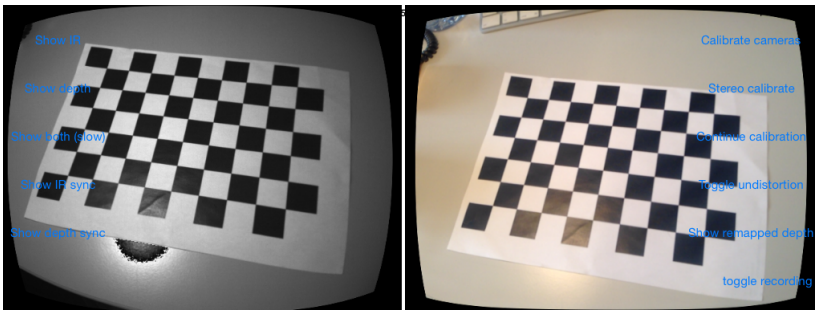


Figure 2.5: Undistorted IR and RGB images

The algorithm estimates the $\begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$ coordinates for all points of the calibration pattern in the world coordinate system based on a series of images from a certain camera. On the other hand it also creates stereo pairs of infrared and color images, thus taking images form different cameras and so estimates the relative transformation between the two cameras. Therefore we will retrieve the rotation matrix R and the translation vector t which describe the dislocation of the infrared sensor from the color camera or vice versa.

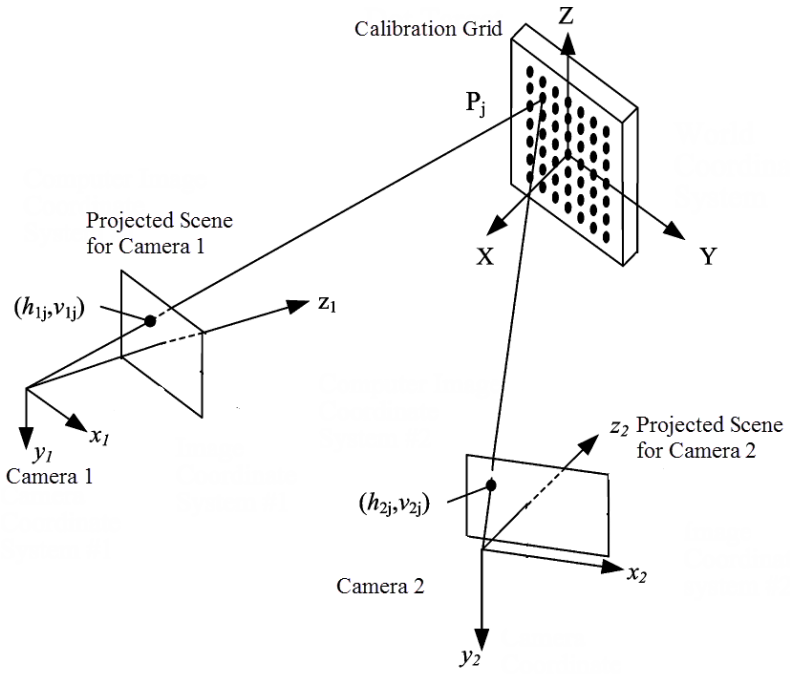


Figure 2.6: Stereo camera calibration

For better understanding we can also represent R and t as a combination of two camera to world transformations.

$$R = R_{rgb}(R_{ir})^T \tag{2.8}$$

$$T = T_{rgb} + RT_{ir} \tag{2.9}$$

Where R_{rgb} and t_{rgb} are the extrinsic parameters of the color camera and vice versa for the infrared camera. Thus R and t define the transformation from the infrared coordinate system to the color camera coordinate system.

We can also control the quality of the stereo calibration by looking at the RMS like with 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

As we now have obtained the camera specific parameters and also the extrinsic parameters the only thing left is to combine this information. Thus we want to create an RGBD image 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.

Thus for the first time now we record a depth image with the sensor staying at the same spot relative to the color camera as for the calibration. Hence we have a depth map / color image pair which we combine using the pinhole camera model and all the parameters we obtained throughout calibration. We will do the following procedure for every pixel in the depth image. First we project it into its cameras coordinate system using the camera matrix. Then we apply the rotation and translation obtained during stereo calibration to transform the point into the color camera coordinate system. And last we back-project the point into the color camera frame to obtain its pixel coordinates, whereas the z coordinate of the point in the color camera coordinate system represents its depth.

Mathematically this operation is formulated the following way whereas we use these variables:
p_{ir} : Point in the infrared camera frame
P_{ir}: 3D point in the infrared camera's coordinate system
P_{rgb}: 3D point in the color camera's coordinate system
p_{rgb}: Projection of P_{rgb} onto the color camera frame

$$P_{ir} = inv(K_{ir}) * p_{ir} \tag{2.10}$$

$$P_{rgb} = R * P_{ir} + t \tag{2.11}$$

$$p_{rgb} = K_{rgb} * P_{rgb} \tag{2.12}$$

$$depthof p_{rgb} = P_{rgb}.z \tag{2.13}$$

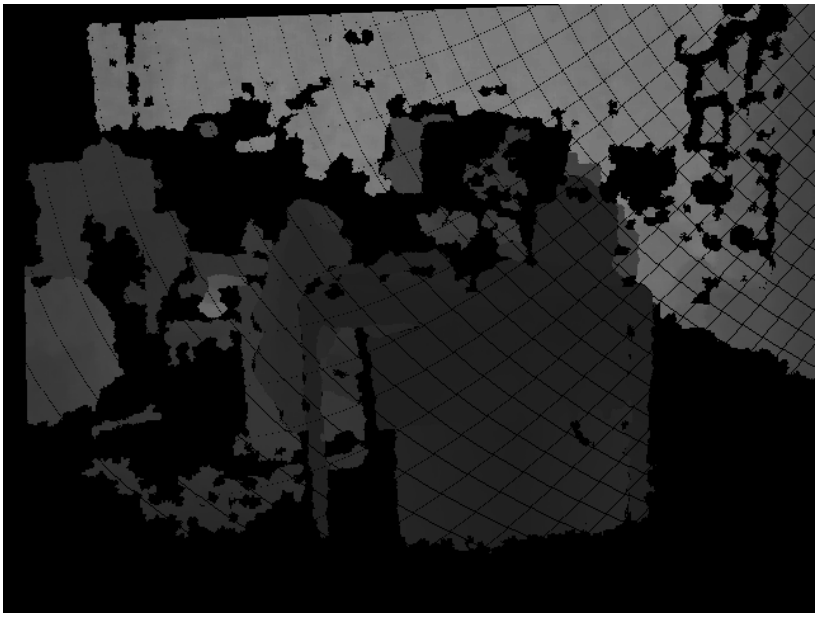


Figure 2.7: Re-projected depth image

Because we project the depth values in the color frame we always lose some information. E.g. those values which would be projected outside the color frame can't be used anymore. We also see some aliasing effects throughout the entire image. This could be solved by a triangulation approach.

Todo: show depth and depth re-projected next to each other

Todo: Show RGBD.

Chapter 3

3D reconstruction

To obtain a 3D reconstruction we will rely on the fastfusion pipeline [2]. It takes as input color image / depth map pairs and their corresponding camera poses. For that reason we use the semi-direct visual odometry pipeline to track the camera center based on the color image sequence and estimate the camera trajectory.

3.1 Camera tracking - SVO

SVO: Semi-Direct Monocular Visual Odometry is an algorithm that performs motion estimation without relying on costly feature extraction and robust matching [**?**]. Instead it operates on pixel intensities which results in subpixel precision at high frame rates. In general we leave the procedure of the SVO pipeline unchanged. We just extended the code to also export the $\begin{pmatrix} u \\ v \end{pmatrix}$ coordinates and the corresponding Z values of the tracked features. Besides that we obtain the world to camera translation and rotation matrices as a result from the algorithm.

Todo: How was the performance of SVO on the dataset? -¿ iPad not fast enough, thus too large gaps between images -¿ bad tracking

3.2 Camera tracking - bundler / visualsfm

Operates with sift, no consecutive frames, matches all images. Takes longer, but also bad performance

3.3 Scale estimation

Because SVO operates in its own coordinate system, which is based on the features it is tracking, the Z values which is the estimated depth of the features don't align with the real depth.

That means that for a feature at position $\begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$ which is projected to $\begin{pmatrix} u_i \\ v_i \\ z_i \end{pmatrix}$ in the camera frame z_i usually doesn't equal the z value of P_{rgb} in 2.11, 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:

$$\sum_{i=0}^{width*height*images} (d_i - s * z_i) * z_i = 0 \tag{3.1}$$

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

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

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.

3.4 Pointcloud

To obtain a visual conformation that the scale estimation succeeded we converted the depth map / color image pairs to pointclouds. For this we computed the camera pose as in 3.2, to get a visual of the image origin. Then we used the extrinsic parameters of the color camera to transform the RGBD image to a pointcloud.

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

where p_i is the vector $\begin{pmatrix} u_i d_i \\ v_i d_i \\ d_i \end{pmatrix}$, u and v being the pixel coordinates of the depth d in the color camera frame. Accordingly the color of p_i is the RGB value at (u_i, v_i).

Todo: put pointcloud img

3.5 Dense reconstruction

To perform the dense 3D reconstruction we use the fastfusion pipeline [2].

Todo: some more information on fastfusion

3.6 App integration

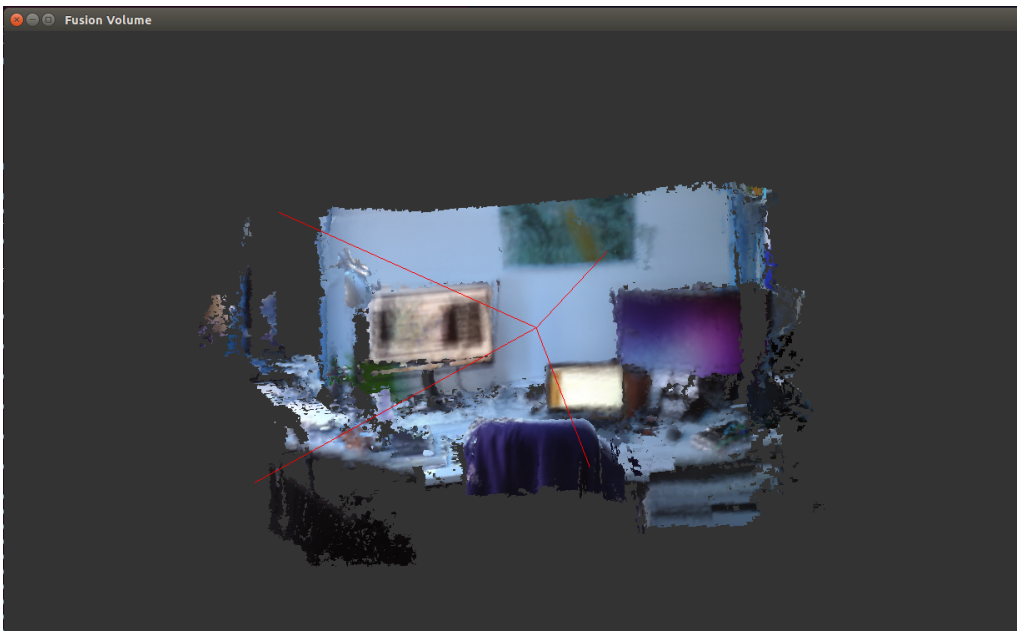


Figure 3.1: Re-projected depth image

Chapter 4

Conclusion

Chapter 5

Future work

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
```