Volumetric Capture

Marcel Bruckner
marcel.bruckner@tum.de

Kevin Bein
kevin.bein@tum.de

Moiz Sajid
moiz.sajid@tum.de

Technische Universität München

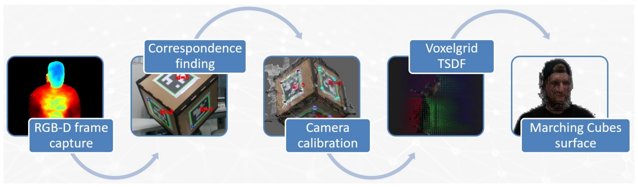


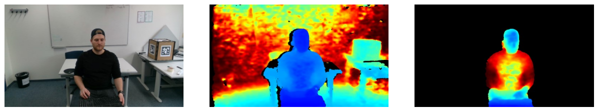
Figure 1: Reconstruction pipeline used in this project: RGB-D Frame Capture, Correspondence Finding, Camera Calibration, Voxelgrid TSDF, Marching Cubes

Abstract

Our goal in this project is to get a real time mesh reconstruction of dynamic scenes using a multi view camera capture setup. We use three Intel RealSense D415 depth cameras for RGB-D data capturing which are calibrated using known correspondences from markers. The data is fused into a voxelgrid representing the implicit surface and following a marching cubes algorithm is performed for the final surface extraction. Both qualitative and quantitative results are presented.

1. Introduction

Volumetric Capture is an extensively researched topic where the goal is to get an accurate 3D reconstruction of dynamic scenes in real time. The first work in the area was done by Cureless and Levoy [2] in which they integrated data from one range image into a cumulative weighted signed distance function. KinectFusion [9] was another milestone in this area which proposed a real time mapping system of indoor scenes using a single depth camera. DynamicFusion [8] handles non-rigid scenes and reconstruction over time. Our work mostly focuses on reconstructing dynamic scenes on a per frame basis.
One application area of our project is Holoportation [15] where 3D reconstructed models are used for VR interactions between two persons over huge distances.



(a) RGB Image (b) Depth Image (c) Threshold filter

Figure 2: The color and depth images captured by the cameras

2. Reconstruction Pipeline

To perform the real time 3D reconstruction we implemented the pipeline which is exemplary displayed in Figure 1 and described in the following section.
All tests were performed on a Laptop with an Intel® Core™ i7-6700HQ CPU @ 2.60GHz [4], 40GB of DDR4 RAM and a Nvidia GeForce GTX 970M [10].

2.1. RGB-D Frame Capture

We use three Intel© RealSense™ Depth Camera D415 [5] for our capture setup which work on the concept of active stereo depth. Two cameras are placed on the sides of the camera and the depth is calculated from a triangulation using the displacement. An infrared pattern is projected into the scenweree to enhance the details.
We process the depth images using a threshold filter which discards pixels that are above or below a threshold set during runtime. We also tried different hole-filling, spatial and edge enhancing filters that did not improve the results and are not used in the final reconstruction.
Figure 2 displays the raw RGB-D images and the filtered depth image.

We tried to extend the three camera setup with a fourth one but were not able to integrate it due to bandwidth limitations. Furthermore, we had to limit the color and depth resolutions as well as framerates to enable three cameras and to allow a processing of the data in real time. The resolution we used for the color and depth stream is 848x480 px at 30 frames per second. [7] gives more detail about the limitations of the cameras in a multi camera setup.

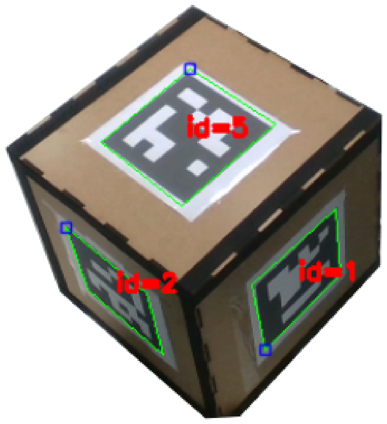


Figure 3: Marker cube with the detected markers highlighted in green and their unique identifiers (red).

The RGB-D images from the three cameras are time synchronized [7] what improves the results of the following camera calibration and the final mesh reconstruction.

Intel© provides an excellent SDK [6], good support and updates for its API which made it easy to use the cameras.

2.2. Correspondence Finding

We use ArUco markers [11] to find correspondences between the three camera streams. These markers have a unique identifier and can easily be detected in every color image using the OpenCV [13] library.
The markers are mounted on a wooden cube box as shown in Figure 3.

By placing the cameras such that each one is looking on one of the cubes corners we can assure that in every color image three markers are visible. This also ensures that between every pair of cameras we have an overlap of two markers for which the eight corner positions are detected sub pixel perfect. This gives us a total of $8 * 3 = 24$ constraints between every pair of cameras which are used in the next step to estimate the relative transformation between the markers.

It is also important to mention here that we tried ChArUco markers [12] which are a mixture of ArUco markers [11] and a chessboard pattern which allow simultaneous detection and pose estimation. However, the ChArUco markers [12] did not give us good pose results and slowed down the calculation (Table 1), so we stick to ArUco markers [11].

2.3. Camera Calibration

We use the known marker positions in the color image and combine them with the depth information to calculate the 3D position of the marker corners. As we ensure to have an overlap of two markers between every pair of cameras we now have $8 * 3 = 24$ constraints (8 markers each with a x, y and z value). The 3D positions are then fed into two alignment algorithms to find the relative transformations between the cameras and to align the pointclouds based on the results.

2.3.1 Procrustes

The Procrustes algorithm aligns two pointclouds using known correspondences by estimating the relative translation and rotation between the two sets of points.

Translation We calculate the center of gravity for every pointcloud by summing up the individual point positions and dividing by the number of points. This gives us for every set of points the mean point. The translation is then estimated by the difference between the centers of gravity.

$$T_{ij} = C_i - C_j \tag{1}$$

were

Rotation The rotation between two pointclouds can be estimated by optimizing over the unknown rotation R_{ij}. This is done by minimizing the mean squared error between the set of points X_i and the X_j.

$$\min_{R_{ij}} ||X_i - R_{ij} X_j||_2^2 \tag{2}$$

Fortunately there exists a closed form solution for the rotation R_{ij} which is based on the singular value decomposition of the matrix $X_i^T X_j$.

$$X_i^T X_j = U \Sigma V^T \tag{3}$$

$$R_{ij} = U V^T \tag{4}$$

Table 1 displays the duration and mean squared error values after the Procrustes alignment. It shows a still fairly high error, so we had to further align the pointclouds.

2.3.2 Point-to-Point Error

In order to further improve the alignment that we get from Procrustes we use a Point-to-Point correspondence error. We optimize for the relative translation T_{ij} and rotation R_{ij} over every pair of camera frames i and j and in each of these for the known correspondences k. We came up with the following energy term:

$$\min_{T_{ij}, R_{ij}} \sum_i^{frame} \sum_j^{frame} \sum_k^{corres.} ||X_{ik} - T_{ij} R_{ij} * X_{jk}||_2^2 \tag{5}$$

A comparison of the results from this optimization is given in Table 1. It shows that the mean squared error is much lower using the transformation from the Point-to-Point optimization. On the other side it displays an increase of the duration for the estimation as it is an iterative optimization algorithm, but as the pose estimation is done only once in the beginning the increase is negligible.

2.4. Voxelgrid TSDF

We calculate a truncated signed distance field (TSDF) based on a voxelgrid to represent the implicit surface of the aligned pointclouds. The voxelgrid is a three-dimensional cuboid structure with points (voxels) filling it on a grid at a given resolution. An exemplary voxelgrid is displayed in Figure 4.

We fuse the aligned pointclouds into the voxelgrid by iterating over every voxel and projecting it into the depth frame of the camera. We can now lookup the surface depth value z_{depth} at the projected position in the depth image and compare it to the depth of the voxel. This gives us the distance d_i of the voxel to the surface in the i-th camera frame:

$$d_i = z_{voxel} - z_{depth,i} \tag{6}$$

The final TSDF value is then calculated by truncating the distance with a given threshold t:

$$tsdf_i = min(max(-t, d_i), t) \tag{7}$$

We do this integration for all three sets of points coming from the cameras. To get a valid representation of the implicit surface for all pointclouds we perform a weighted averaging:

$$tsdf_{i+1} = \frac{tsdf_i * weight}{weight + 1} \tag{8}$$

As we compute the TSDF values for a large number of voxels we had to parallelize the calculations using the GPU (Section 2). Since we are using OpenGL [14] for the rendering we decided to use OpenGL [14] compute shaders [3] for hardware accelerated parallelization. Table 2 displays the duration for integrating all three camera frames into the voxelgrid at different resolutions and their respective number of voxels.

2.5. Marching Cubes Surface Extraction

We extract the implicit surface represented by the voxelgrid using the Marching Cubes algorithm [1].
We iterate over all voxelcells that are spanned up by eight adjacent voxels in the voxelgrid. Within these cells the TSDF values for all eight corners are looked up in the voxelgrid and then used to find the zero-crossings along the edges between every pair of corners. We use these zero-crossings to look up the triangulation for the voxelcell based on the Marching Cubes tables described by [1]. This gives us the trianguled mesh as a final output that approximates the implicit surface represented by the voxelgrid. Figure 5 shows a triangulation of one of our group members at a resolution of 10 mm.

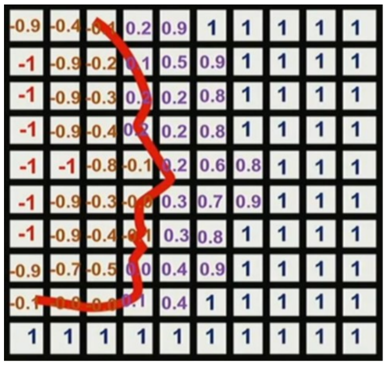


Figure 4: A two-dimensional voxelgrid with an implicit surface (red) and the respective TSDF values. The TSDF values are positive outside of the surface and negative inside. The zero-crossing represents the surface.

We implemented a two pass OpenGL Compute Shader to calculate the triangulation. The upper bound for the number of triangles that can be generated during Marching Cubes is 5 times the number of voxelcells. This gives a possible count of $(200 * 200 * 200) * 5 = 40.000.000$ triangles for e.g. a voxelgrid of 200^3 voxels. In reality the numbers of actually generated triangles is much smaller with $200.000 - 250.000$ which displays the need of a two pass shader. The first pass is only to count the number of triangles that will be generated and to allocate memory. In the second pass we then generate the triangles and fill the memory that exactly fits the needs without overhead.
Table 3 displays the number of triangles that are generated by the algorithm at different resolutions and the respective durations. It can be seen that down to a 5mm resolution we are easily real time capable at 60 frames per second.

3. Results and Conclusion

Section 7 shows a variety of reconstructed scenes using our reconstruction pipeline.

We have achieved our goal to get a real time mesh reconstruction of dynamic scenes using a multi view camera capture setup. We can calculate the relative poses between the cameras during calibration to align the pointclouds that the three cameras capture. By fusing these sets of points into a voxelgrid we can represent the implicit surface and extract it using the Marching Cubes algorithm. All these steps are performed on GPU which gives as a real time reconstruction of dynamic scenes.

4. Future Work

Since we optimized the relative transformations during camera calibration for a small number of markers and thus correspondences one of the further improvements that can be done is to implement a Nearest Neighbor Search. This would result in a much higher number of correspondences on which global alignment techniques like Iterative Closest Points (ICP) can be performed.

Furthermore, we currently see a high number of artifacts that are generated at the overlap of the three camera streams (Figure 6). To reduce these a next step is to calculate depth normal maps for every depth stream and to use these to improve the weighted averaging during TSDF calculation.

To improve the whole reconstruction a final improvement that can be done is to track the reconstructed scene over time. By using the temporal information and the deformations between the frames a canonical model can be generated that would improve the reconstruction further.

5. Group Member Contributions

Research Marcel Bruckner, Kevin Bein, Moiz Sajid

Final Presentation & Report Marcel Bruckner, Kevin Bein, Moiz Sajid

Code Marcel Bruckner

6. Tables

Algorithm	**Iter.**	**MSE**	**Time [ms]**
Procrustes (A)	1	<0.7 - 1.73	8 - 20
Point-to-Point (A)	1 - 20	<0.1	900 - 1000
Procrustes (C)	1	0.25 - 1.29	20 - 40
Point-to-Point (C)	1 - 20	<0.1	± 20.000

Table 1: Comparisons between different optimization algorithms in terms of iterations, MSE (Mean Squared Error) and time. A = ArUco, C = ChArUco

Algorithm	**Res. [mm]**	**Voxels**	**Time [ms]**
Voxelgrid	20	125.000	<1
Voxelgrid	10	1.000.000	<1
Voxelgrid	5	8.000.000	5
Voxelgrid	4	15.625.000	20
Voxelgrid	3	37.000.000	50

Table 2: Comparison of the time needed to calculate the TSDF values in a $1m^3$ voxelgrid for a multitude of voxelgrid resolution.

Algorithm	**Res. [mm]**	**Triang. [$\pm 20\%$]**	**Time [ms]**
March. Cubes	20	10.000	<1
March. Cubes	10	40.000	<1
March. Cubes	5	200.000	15
March. Cubes	4	400.000	35
March. Cubes	3	800.000	80

Table 3: Comparison of the time needed to calculate the triangulation of the implicit surface represented by a $1m^3$ voxelgrid using the Marching Cubes algorithm at varying resolutions.

7. Images

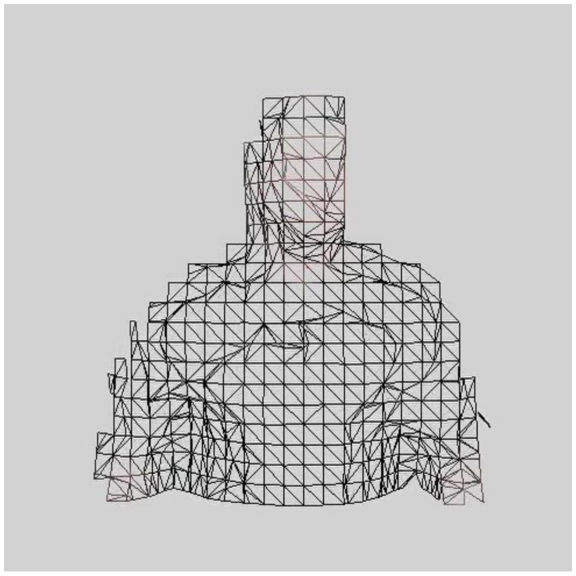


Figure 5: Triangulated mesh of one of our group members using one camera. $1m^3$ voxelgrid at 10 mm resolution.

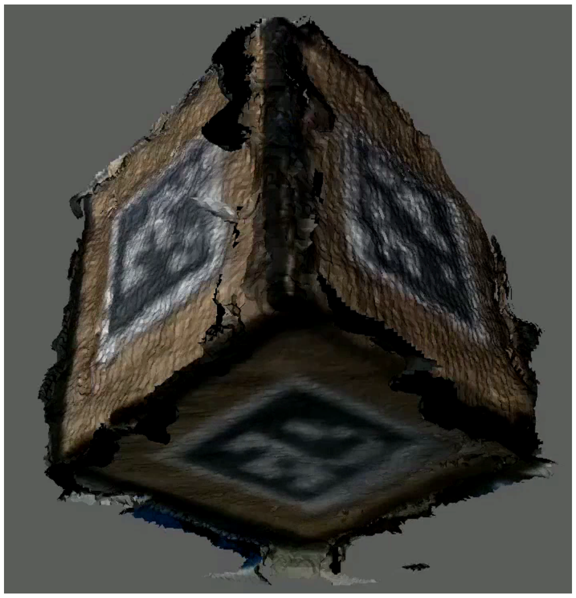


Figure 7: Reconstructed surface of the marker cube using all three cameras. The artifacts that arise in the parts where the pointclouds of the cameras overlap (cube edges) can be seen. $1m^3$ voxelgrid at 5 mm resolution.

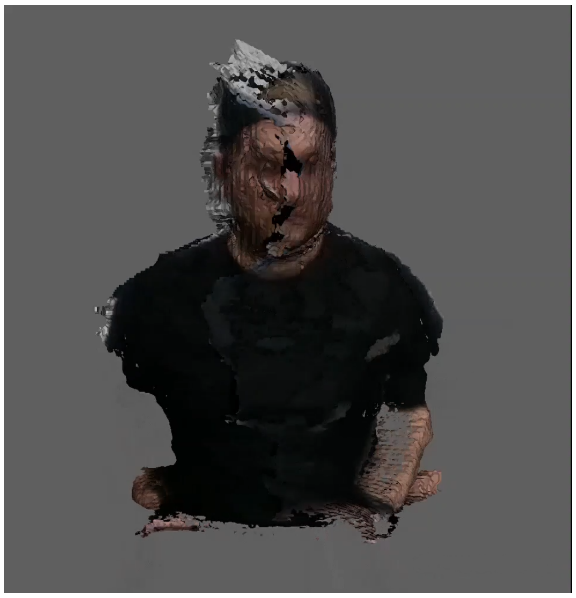


Figure 6: Reconstructed surface of one of our group members using all three cameras. The artifacts that arise in the parts where the pointclouds of the cameras overlap (face) can be seen. $1m^3$ voxelgrid at 5 mm resolution.

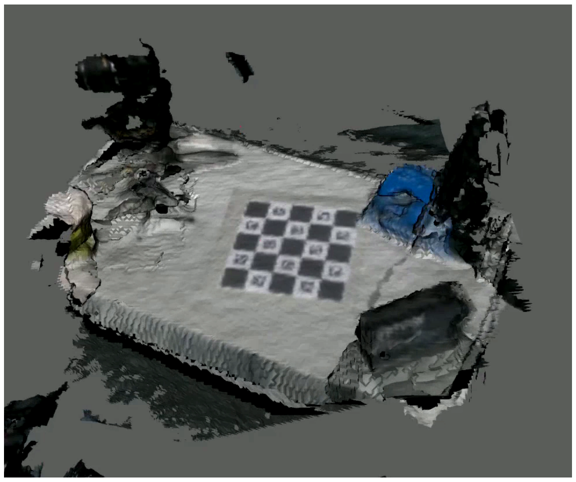


Figure 8: Reconstructed surface of an arbitrary scene using all three cameras. The ChArUco board (middle), a hole puncher (blue), a stapler (lower right) and a DSLR camera (top left) are displayed. $1m^3$ voxelgrid at 5 mm resolution.

References

[1] Paul Bourke. Polygonising a scalar field. http://paulbourke.net/geometry/polygonise/. Accessed: 19.02.2020.

[2] B. Curless and M. Levoy. A volumetric method for building complex models from range images. *ACM SIGGRAPH*, 1996.

[3] A. Gerdelan. It's more fun to compute: An introduction to compute shaders. http://antongerdelan.net/opengl/compute.html, 2016. Accessed: 10.02.2020.

[4] Intel©. Intel® core™ i7-6700HQ prozessor. https://ark.intel.com/content/www/de/de/ark/products/88967/intel-core-i7-6700hq-processor-6m-cache-up-to-3-50-ghz.html. Accessed: 19.02.2020.

[5] Intel©. Intel© realsense™ depth camera d415. https://www.intelrealsense.com/depth-camera-d415/. Accessed: 19.02.2020.

[6] Intel©. Intel© realsense™ sdk. https://www.intelrealsense.com/developers/. Accessed: 19.02.2020.

[7] Intel©. Using the intel© realsense™ depth camera d4xx in multi-camera configurations. https://www.intel.com/content/dam/support/us/en/documents/emerging-technologies/intel-realsense-technology/RealSense_Multiple_Camera_WhitePaper.pdf. Accessed: 19.02.2020.

[8] R. A. Newcombe, D. Fox, and S. M. Seitz. Dynamicfusion: Reconstruction and tracking of non-rigid scenes in real-time. *IEEE CVPR*, 2015.

[9] 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. *IEEE ISMAR*, 2011.

[10] Nvidia. Geforce gtx 970m. https://www.geforce.com/hardware/notebook-gpus/geforce-gtx-970m. Accessed: 19.02.2020.

[11] OpenCV. Detection of aruco markers. https://docs.opencv.org/trunk/d5/dae/tutorial_aruco_detection.html. Accessed: 19.02.2020.

[12] OpenCV. Detection of charuco corners. https://docs.opencv.org/3.4/df/d4a/tutorial_charuco_detection.html. Accessed: 19.02.2020.

[13] OpenCV. Opencv. https://opencv.org/. Accessed: 19.02.2020.

[14] OpenGL. Opengl. https://www.opengl.org/. Accessed: 19.02.2020.

[15] S. Orts-Escolano, C. Rhemann, S. Fanello, W.Chang, A. Kowdle, Y. Degtyarev, David Kim, P. Davidson, S. Khamis, M. Dou, V. Tankovich, C. Loop, Q. Cai, P. Chou, S. Mennicken, J. Valentin, V. Pradeep, S. Wang, S. Bing Kang, P. Kohli, Y. Lutchyn, C. Keskin, and S. Izadi. Holoportation: Virtual 3d teleportation in real-time. *ACM User Interface Software and Technology Symposium (UIST)*, 2016.