Volumetric Capture

Marcel Bruckner
marcel.bruckner@tum.de

Kevin Bein
kevin.bein@tum.de

Moiz Sajid
moiz.sajid@tum.de

Technical University of Munich

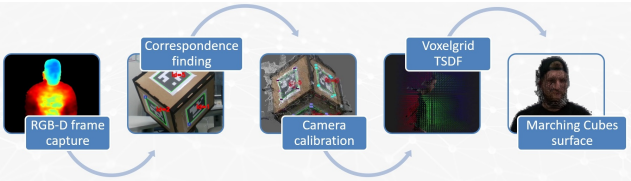


Figure 1: Reconstruction pipeline used in this project: RGB-D Frame Capture, Correspondence Finding, Camera Calibration, Voxelgrid TSDF, Marching Cubes

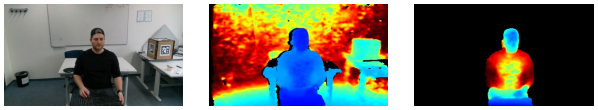
Abstract

Our goal in this project is to get a realtime 3D mesh reconstruction of humans using a muti-view camera capture setup. We used three Intel RealSense cameras for data capture rig. The cameras are calibrated using the marker correspondences. Finally, the data is fused into the single voxelgrid TSDF followed by Marching cubes for the final surface extraction. Both qualitative and quantitative results have been presented in this report.

1. Introduction

Volumetric Capture is an extensively researched topic where the ultimate goal is to get an accurate 3D reconstruction in realtime. The first work in the area was done Cureless and Levoy [1] in which they integrated data from one range image into cumulative weighted signed distance function. KinectFusion [4] was another milestone in this area which proposed a realtime mapping system of indoor scenes using a single depth camera. DynamicFusion [3] handles non-rigid scenes but our work mostly focussed on handling rigid scenes.

One of application areas of our project is Holoportation [5] where 3D recontructed models were used for interactions between two persons using Microsoft HoloLens.



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

Figure 2

2. Reconstruction Pipeline

2.1. RGB-D Frame Capture

We used three D415 Intel RealSense cameras for our capture setup. One of the benefits of using Intel RealSense cameras is that it provides good support and updates for its API. The Intel RealSense cameras work on the concept of Active Stereo in which a texture in projected for better better details. The RGB-D frames from the three cameras were time synchronized which not only helped us to get better results for the cameras calibration but also for the final mesh reconstruction. For depth image preprocessing, we used Threshold Filtering in which the depth of the objects that are a certain distance away from the camera get discarded.

For further filtering, we also tried using Hole Filing for ending gaps in the depth images, Spatial Filtering for smoothing the edges and Edge Enhancement Filter. All of these filters did not led to any significant improvements hence they were not used in the final mesh reconstruction. For captur setup, we initally wanted to use four camera but due to bandwidth limitations we were not able to use color and the depth streams from all four cameras at the same time instance. We also had to limit the color and depth frame resolution for faster processing. The resolution which we used for the color stream is 848 by 480 pixels and for the depth stream is 848 by 480 pixels.

2.2. Correspondence Finding

For correspondences, we used ArUco markers with each marker having a unique identifier. The markers were mounted on a cube box as shown in Figure 3. The markers were placed in such a way such that any one particular cam-

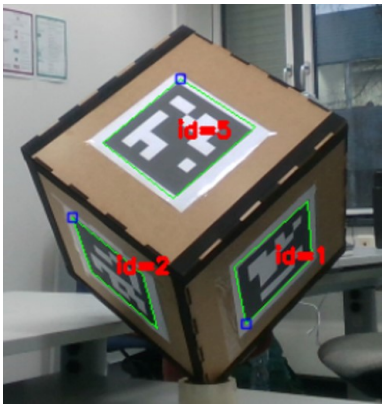


Figure 3: ArUco box cube with the detected marker identifiers. The markers were used for camera calibration

era can easily detect two out of three markers. The cameras placed with significant view overlap. In the end, we just opted for one marker per side. It is also important to mention here that we also tried the ChArUco markers which is basically a chessboard with the ArUco markers. However, the ChArUco markers did not gave us good results.

2.3. Camera Calibration

After getting the marker locations, we backprojected these correspondences into the 3D pointcloud using the depth information. The camera poses of the cameras were then estimated using OpenCV. Using these camera poses, we calculated relative transformatons between the cameras. Out of three camera, we selected the first camera's coordinate frame as the world coordinate frame and hence the poses from the other two cameras were aligned with respect to the first camera. The alignment we got from the camera calibration was not good which meant that it can be improved further.

2.4. Optimization

As the pose parameters that we got from the camera calibration were not good enough, we used two different optimizations approaches namely Procrutes and Point-to-Point correspondence error for optimizing the 3D pose parameters of the cameras even further.

In the Procrustes algorithm, we compute the vector between the center of gravity for getting the translation components and the Singular Value Decomposition of the known correspondences for getting the rotation components as shown in Equation (1) and (2). The alignment that we get from Procrustes is not robust enough and hence has a high Mean Squared Error (MSE).

$$USV^T = SVD(X^T X) \tag{1}$$

$$R = UV^T \tag{2}$$

In order to further improve the results that we get from Procrustes, we used the Point-to-Point correspondence error. For the Point-to-Point correspondence error, we optimize for the following energy term:

$$\sum_i \sum_j \sum_k \|(X_{ik} - T_j R_j * X_{jk})\|_2^2 \tag{3}$$

Further results have been sumarzied in Table 1. One of the difficulties that we faced in camera calibration was visualizing the aligned points for which we had to write our own rendering pipeline in OpenGL [6]. Another difficulty that we faced was with Ceres because we had to get ourselves acquainted with general working of the non-linear optimization framework.

2.5. Voxelgrid TSDF

Once we have these aligned pointclouds, we fuse them into the voxelgrid. We project the voxelgrid into the depth frame of the camera and we calculate the difference by subtracting the voxel depth from the actual depth value using Equation (4). For the truncated signed-distance function (tsdf) values we did the weighted averaging using the Equation (5). The tsdf values were truncated between -1 and 1 with points having a negative value inside the surface and positive value outside the surface.

$$tsdf = z_{voxel} - z_{depth} \tag{4}$$

$$tsdf_{i+1} = \frac{tsdf_i * weight}{weight + 1} \tag{5}$$

Given that we were using a large number of gridpoints, computing tsdf values on the CPU was not feasible. So we had to move our tsdf voxelgrid implementation onto the GPU inorder to achieve realtime results. Since we were already using OpenGL for the rendering pipeline, we decided to use OpenGL Compute Shaders [2] for parallelizing the tsdf voxelgrid implementation. We also had to find a tradeoff between voxel resolution and frame rate. With a lower voxel resolution, the frame rate dropped because more calculations had to be done. Furthermore, finding a good truncation distance was also very crucial for lowering the artefacts. The Table 2 summarizes the time comparisons between different voxel resolutions.

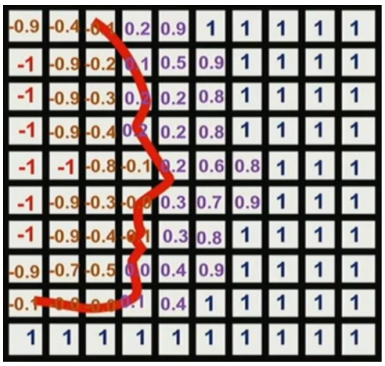


Figure 4: TSDF Represenation

2.6. Surface Extraction

For the iso-surface extraction, we used the Marching Cubes algorithm which converts the implicit surface representation to a polygonal mesh. We then iterated over the voxelgrid cells to determine the zero crossings and looked up the triangulation in the Marching Cubes table. This gives us the trianguled mesh as an final output. With a lower voxel resolution and more grid points, we were able to get more finer reconsturction. Again we had to utilize the OpenGL Compute Shader in order to achieve realtime results because the computations on the CPU were really slow.

Intially, we had voxelgrid points that were not initialized because they were outside the camera view. As a result, we had a huge number of artefacts in the final mesh reconstruction which took a lot of time to figure out. We implemented a two pass OpenGL Compute Shader. In the first pass, we counted the number of triangles that will generated and only allocated space for them. In the second pass, we generate the triangles. We do this because the upper bound for the number of generated triangle is five times the number of voxelgrid points. For example if we have 200x200x200 grid points with a 5 millimeter voxel resolution, we roughly had 40 million number of possible triangles. We only generated a fraction of these triangles after the the triangle counting OpenGL Compute Shader pass. For a voxel resolution of 5mm resolution, the Marching Cubes algorithm took 15 milliseconds which was realtime capable. For 3 millimeter resolution, the results were not realtime capable however, we were to get fine mesh reconstructions. The Table 2 summarizes the time comparisons between different voxel resolution. The Table 3 summarizes the time comparisons for running Marching Cubes between different voxel resolutions.

3. Results

The results that we get using one camea are really good as can be seen in Figure 5. Figure 6 displays the triangulation results using one camera. However, we get artefacts when we use all three camera streams. The which artefacts which are mostly visible around the edges can be seen in Figure 7. This suggests that the camera pose parameters can be improved further.

Figure 5: Result with using only one camera

Figure 6: Triangulation result with using only one camera

Algorithm	Iterations	Duration [ms]	MSE
Procrustes (A)	1	8 - 20	<0.7 - 1.73
Point-to-Point (A)	1 - 20	900 - 1000	<0.1
Procrustes (C)	1	20 - 40	0.25 - 1.29
Point-to-Point (C)	1 - 20	18000 - 22000	<0.1

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

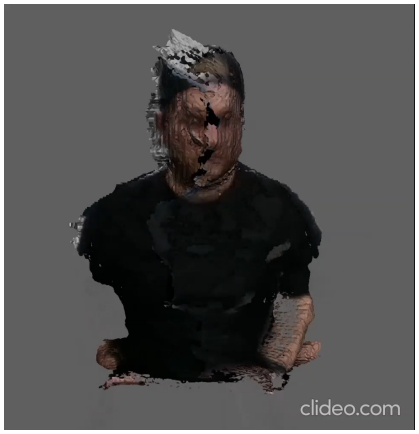


Figure 7: Result with using three cameras

Algorithm	Resolution [mm]	Gridpoints	Duration [ms]
All frames	20	125,000	<1
All frames	10	1,000,000	<1
All frames	5	8,000,000	5
All frames	4	15,625,000	20
All frames	3	37,000,000	50

Table 2: Time comparisons for calculating voxelgrid of different voxel resolutions as well as the number of gridpoints used. The voxelgrid measured 1 meter x 1 meter x 1 meter.

Resolution [mm]	Duration [ms]	Gridpoints
20	<1	7,000 - 10,000
10	<1	40,000 - 45,000
5	15	200,000 - 250,000
4	35	380,000 - 420,000
3	80	600,000 - 800,000

Table 3: Time comparisons for running the Marching Cubes on different voxel resolutions. The number of triangles generated are also displayed.

4. Future Work and Conclusion

Since we optimized for pose parameters for a limited number of markers, one of the further improvements that we can do in camera calibration is to use more correspondences which can found for example by using Nearest Neighbor Search and then use the Global Alignment techniques like Iterative Closest Points (ICP).

5. Group Member Contributions

References

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

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

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

[4] 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.

[5] 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.

[6] J. Vries. Learnopengl. `https://learnopengl.com/`, 2015. Accessed: 10.02.2020.