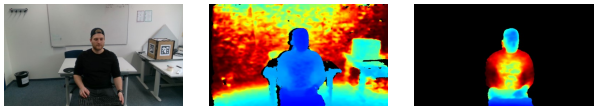
Volumetric Capture

Marcel Bruckner
email@tum.de

Kevin Bein
email@tum.de

Moiz Sajid
moiz.sajid@tum.de

Technical University of Munich



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

Figure 1

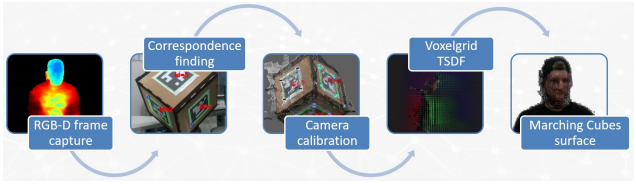


Figure 2: Reconstruction pipeline used in this project: RGB-D Frame Capture, Correspondence Finding, Camera Calibration, Voxelgrid TSDF, Marching Cubes

Abstract

In this project, our goal is to get realtime 3D mesh reconstruction using a muti-view camera capture setup. We used Intel RealSense cameras for data capture followed by calibration using the marker correspondences. Data is fused into the voxelgrid TSDF followed by Marching cubes for the surface extraction.

1. Introduction

Volumetric Capture has been a extensively researched topic [1].

2. Reconstruction Pipeline

2.1. RGB-D Frame Capture

We used three D415 Intel RealSense cameras for our setup. The reason for using Intel RealSense cameras is good API support that Intel provides for its RealSense cameras. Intel Realsense cameras work on the concept of Active Stereo. The frames from the cameras were time synchronized inorder to get better results for the calibration as well the final mesh reconstruction. For depth preprocessing, we used Threshold Filtering in which the depth of the objects that are certain distance away from the camera get discarded.

For further filtering, we used Hole Filing for ending gaps in the depth images, Spatial Filtering for smoothing the edges and Edge Enhancement Filter. All of these previous filters did not led to any significant improvements. 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. We also had to limit the color and depth frame resolution which in our case was 848 by 480 for the color stream and 848 by 480 for the depth stream.

2.2. Correspondence Finding

For correspondences, we used ArUco markers with each having a unique identifier. The markers were easily detected because we had significant camera view overlap. In the end, we just opted for one marker per side. Once we have the markers, we backproject the correspondences into the pointcloud. We also tried ChArUco markers which is basically a chesboard with arUco markers. But using the ChArUco markers

2.3. Camera Calibration

Once we have the marker locations, we backproject them into the 3D point clouds so that we can estimate the 3D pose between the cameras.

2.4. Optimization

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

In the Procrustes algorithm, we compute the vector between the center of gravity for getting the translation component and the Singular Value Decomposition of the known

correspondences for getting the rotation component. The alignment we get from Procrustes is not robust enough and hence has a high Mean Squared Error (MSE). 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:

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

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. 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 as in Equation (1). For the truncated signed-distance function (tsdf) values we did the weighted averaging using the equation. The tsdf values were truncated between -1 and 1 with values having a negative value inside the surface and positive values outside the surface.

$$tsdf = z_{voxel} - z_{depth} \tag{1}$$

$$tsdf_{i+1} = \frac{tsdf_i * weight}{weight + 1} \tag{2}$$

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 calculation 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 resolution.

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

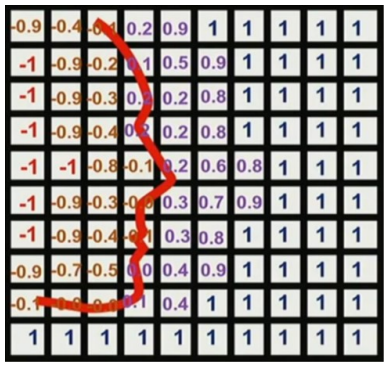


Figure 3: TSDF Represenation

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 result was really good with one cameras. But we had some artefacts when we used multiple cameras. Some artefacts were visible around the edges.

4. Future Work

Since we optimized for pose parameters for a limited number of markers, one of the further improvements that we

Figure 4: Result with using only one camera

Figure 5: Triangulation result with using only one camera

Figure 6: Result with using three cameras

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).

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

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.

5. Group Member Contributions

References

[1] B. Curless and M. Levoy. A volumetric method for building complex models from range images. 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.