3D Scene Reconstruction From Multiple Views

Chenhao Wang
Department of Computing Science
Simon Fraser University
cwa217@sfu.ca

Luciano Oliveira
Department of Computing Science
Simon Fraser University
loliveir@sfu.ca

Oleksandr Volkanov
Department of Computing Science
Simon Fraser University
avolkano@sfu.ca

Sara Pilehroudi
Department of Computing Science
Simon Fraser University
spilehro@sfu.ca

ABSTRACT

In this project, we explore existing techniques to perform a multi-view 3D scene reconstruction from a continuous sequence of images, and propose a novel way to perform 3D scene reconstruction from a limited number of disconnected input frames. Further, we build a proof of concept Matlab application that produces a dense 3D point cloud of the captured scene, using a variable number of RGB-D (RGB-depth) image references as its input.

ACM Reference Format:
Chenhao Wang, Luciano Oliveira, Oleksandr Volkanov, and Sara Pilehroudi. 2019. 3D Scene Reconstruction From Multiple Views. In *Proceedings of CMPT 742.* ACM, New York, NY, USA, 6 pages.

1 INTRODUCTION

The 3D reconstruction of a scene or an object is a classic ill-posed problem in computer vision. Given a sequence of image references, there may be several 3D model solutions that satisfy the input sequence. Thus, the goal of multi-view 3D reconstruction is to find a 3D view representation that minimizes the error caused by the input noise and inherent uncertainty in the input image sequence.

In recent decades, 3D modelling has found many applications, and it follows that multi-view 3D reconstruction techniques can assist in obtaining full-scale 3D models. For instance, building interiors and exteriors can be reconstructed for architecture planning. 3D models are more realistic and easier to understand than 2D sketches, so designs become more vivid, and provide a better visualization of the project, which can help with marketing aspect as well. Additionally, 3D designs can be easier to re-model and make corrections in an interactive manner, thus increasing their appeal in a professional setting.

3D reconstruction has also found various applications in video game, animated movie, and virtual/augmented reality industries, all of which often require modelling of real world objects. Manually designing and creating real world objects which have many details can be a very time-consuming process. Alternatively, the object can be scanned to create an initial model, and can be modified afterwards by a graphic designer according to their needs.

To simplify the multi-view alignment problem, many of the 3D reconstruction techniques use video streams as input, where hundreds of frames can be obtained from slightly different angles of a scene. Such approach makes the alignment process more trivial at the cost of requiring more computational resources, and can produce accurate model representations. In this project, however, we explore a scenario when using a video stream is not feasible, and, as a result, one would need to reconstruct a 3D scene from a limited number of disconnected camera frames.

CMPT 742, Final Project Report, Dec 2019

2 RELATED WORK

In the recent years, consumer-grade depth cameras have become widely available. The introduction of Microsoft Kinect in 2011 made it possible to obtain real-time depth data of a given scene. The follow-up KinectFusion work [1] discusses in detail the process of generating a reconstructed 3D model from the sensor's depth data. Some of the further works explore improving the KinectFusion results by applying texture mapping to the reconstructed depth data ([2], [3]), while others attempt to increase the output quality by using various up-sampling methods on low resolution depth streams ([4], [5]).

Alternatively, a convolutional neural network (CNN) based approach for spatial reconstruction from RGB images [6] have produced impressive results, given that no depth data were used. However, it suffered from noise and artifacts produced by CNN in its results, thus not achieving the same level of fidelity as the traditional depth camera based methods. As a result, for the past few years, the focus has shifted on improving the 3D model texture and model detail accuracy by optimizing the photometric consistency error [7] from the matching input image data.

It is important to notice that the key feature of the current multi-view 3D scene reconstruction is using an input video stream of both color and depth data. The goal of our approach, however, is to obtain an accurate ground truth object reconstruction from a limited subset of input image data, with no guarantee of continuity in the given sequence.

3 METHODS

For the purpose of this project, we used an RGB-D Intel RealSense D415 camera. The camera can simultaneously capture two streams of data: RGB color stream, and depth stream containing distance values with respect to the camera origin. The software bundled with the camera can also generate a point cloud representation of the scene using data from both RGB and depth streams, and save it in a "*.ply" file format. We have explored using different techniques based on the input data type to produce the final 3D reconstruction of the scene.

We consider three distinct methods to reconstruct a 3D scene in this report: (1) reconstruction from RGB-D input image data, (2) reconstruction from calibrated RGB images, and (3) reconstruction from texture-mapped 3D point cloud input data. For each method,

we showcase the results, analyze the method's advantages, as well as its limitations and constraints.

3.1 RGB-D Images

In our initial approach, we have tried to reconstruct a 3D scene of a single object. In order to do this, we take RGB-D snapshots of the target object using the D415 camera as an input, then segment the desired object in each image, and generate 3D point clouds of the segmented regions. The final step is to align all the point clouds in a common 3D world space. Figure 1 has a diagram that summarizes the steps of this approach.

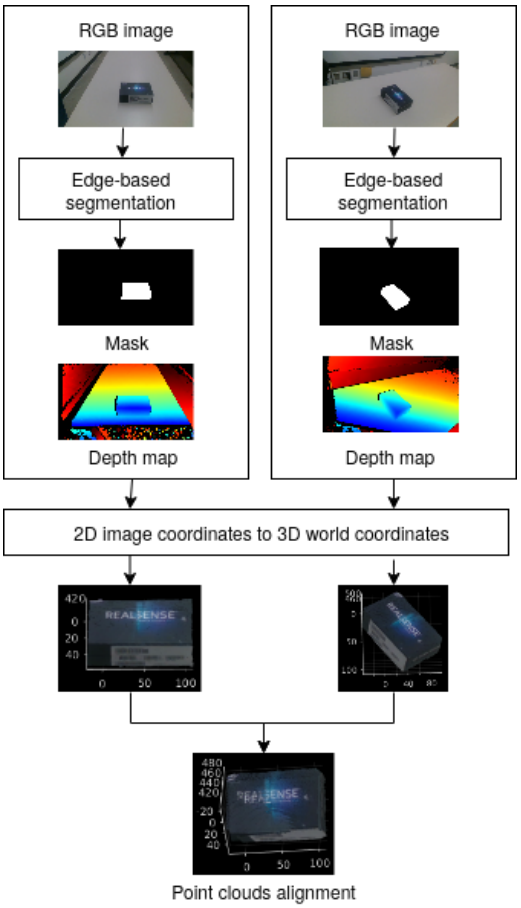

Figure 1: Process of reconstruction from RGB-D images.

3.1.1 Input. For this approach, the input data consisted of a set of 2D RGB images, their corresponding depth maps, and the camera intrinsic parameters metadata file. Each set had six to ten pictures, each capturing the scene from different angles around the object of interest. The depth maps were saved a "*.raw" file format that contains the corresponding depth value of every pixel in the matching RGB image. Some examples are shown in figure 2. The metadata file contains information about the picture resolution and the camera intrinsic parameters, such as focal length, center of projection, and the distortion model type (Brown-Conrady) that was used to calibrate the camera. However, the distortion coefficients are not provided in the metadata file, and could only be obtained by using direct Intel RealSense API queries for a continuous input video stream.

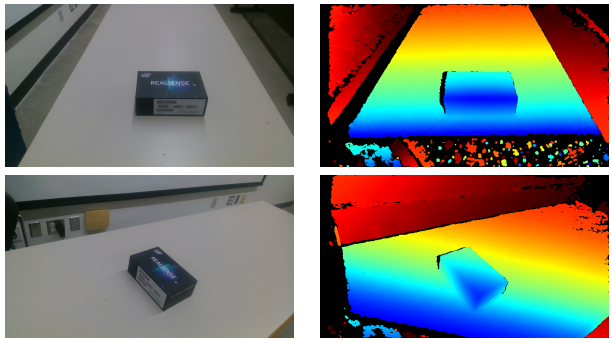

Figure 2: Sample RGB-D image pairs taken by the Intel D415 camera. RGB channel images are on the left, and their corresponding depth map representations are on the right.

3.1.2 Object segmentation. Once we have obtained all the input images from different angles around an object, we had to then remove the background to make sure we reconstruct only the desired object. An image segmentation technique had to be used to create a mask of the object for every picture. We have tried the active contours and graph cut algorithms found in the Matlab Image Processing Toolbox apps, but they did not provide optimal results. Moreover, both algorithms require manual user input. For the active contours, the user needs to provide an initial mask around the object; for the graph cut, the user needs to provide lines that correspond to both foreground and background colours. Since we need to segment multiple images, an algorithm that does not require manual user input was necessary.

3.1.3 Edge-based segmentation. We have found an algorithm can detect and segment the target object without requiring the user to provide any related input. The algorithm in question (edge-based segmentation) consists of the following steps:

(1) Obtain the image gradients using a Sobel operator.
(2) Dilate the edges to make them thicker.
(3) Fill the regions formed by the edges.
(4) Remove all edges that are directly or indirectly connected to one the four borders of the image.
(5) Apply image erosion to remove some pixels on the boundaries of the edge.
(6) Select only the largest regions among the all remaining edges.

These steps are represented in figure 3. The final output should be the mask of the object, which will then be used to selectively create the point clouds. The algorithm, however, has some constraints. First, the desired object should not be connected to any of the borders of the image or to any other edges that are connected to the borders, otherwise it would be removed during step 4. Additionally, one needs to make sure that the desired object will form edges

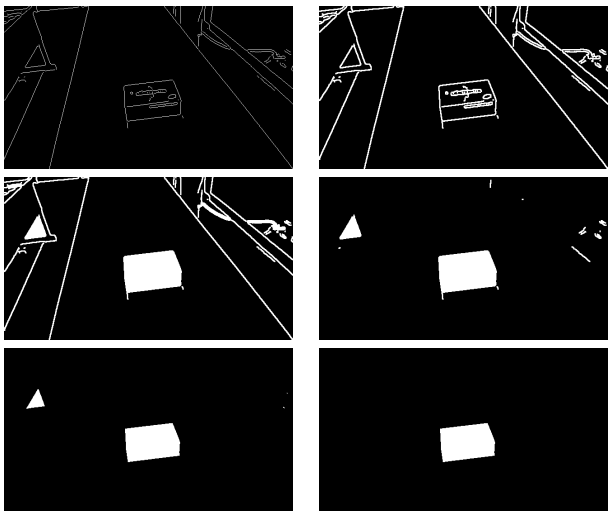

Figure 3: These images present an example of the edge-based segmentation algorithm applied to an picture of the same object box used before. Top row corresponds to steps 1 and 2, followed by steps 3 and 4 (middle row), and steps 5 and 6 (bottom row)

.

with the largest region among the regions obtained before the final step, otherwise the step 6 would also remove it. Nonetheless, these constraints only affect us during the time the photos are being taken, and the rest of the process is fully automated.

3.1.4 3D point clouds. After the final mask of the segmented object is obtained, that mask is then applied to the depth values obtained from the depth raw file in order to access only the distances related to the segmented object. All the depth values whose masks are zero are ignored and, therefore, not used to create the point clouds. At this point, all the image coordinates (**u**,**v**) should be transformed to world coordinates (**X**,**Y**,**Z**) because image coordinates only map the pixels in an image, and we need the actual location of the pixels. For that transformation, we needed to apply the intrinsic parameters of the camera: focal length ($\mathbf{f}_x$, $\mathbf{f}_y$) and centre of projection ($\mathbf{c}_x$, $\mathbf{c}_y$). The metadata file from the depth camera does not provide any information about the extrinsic parameters, so we can only transform the coordinates from different perspectives, and apply rotation and translation later in order to align them. The formula for the transformation without using the extrinsic parameters is shown below.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} * depth$$

Where $\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$ is the world coordinates, $\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}^{-1}$ is the inverse of the intrinsic matrix, $\begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$ is the image coordinates , and **depth** is the distance value from the depth raw file. So we could transform from image coordinates to world coordinates using this transformation matrix.

Another important point is how to obtain the depth used in the transformation formula. The camera sensors estimate depth of each point by its distance related to the camera origin. However, the depth used in the formula is supposed to be the distance of each point related to the camera plane (i.e. a distance that is perpendicular to the camera plane). Fortunately, the camera was pre-calibrated and already gives us the recalculated depth values as we can observe by analyzing the depth maps provided by the camera in figure 2, since every line that is parallel to the camera plane has the same depth value (same colour mapping). Therefore, we can just apply them directly into the formula. Some examples of point clouds created this way are shown in figure 4.

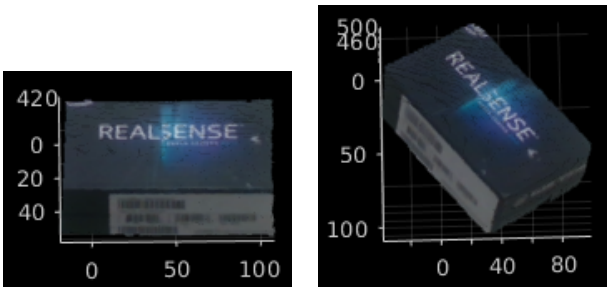

Figure 4: Sample of two point clouds created from both pictures of figure 2.

3.1.5 Alignment. Finally, we needed to align all point clouds created this way. First, we used a built-in Matlab function that implements the iterative closest point (ICP) algorithm, which is capable of registering two point clouds. We have tweaked the outliers ratio and tolerance error function parameters to increase the algorithm's accuracy; however, it was not possible to obtain good results due to the lens distortion found in the input data. And since the metadata file created by the Intel RealSense D415 camera does not provide the coefficients for the distortion, it was not possible to apply an algorithm to un-distort the images either. An example of an ICP registration is shown in figure 5. It is possible to see that the ICP algorithm was not able to successfully find a good match between the point clouds.

Moreover, we also tried to register the point clouds by manually selecting matching points between them, and then applying Horn's method [8] to find the transformation matrix between these set of points. This matrix can then be directly applied to align the corresponding point clouds. As a proof of concept, we manually selected points on the top surface of the box from our point clouds to find a good alignment on that side, as shown in left image of figure 6. However, when the top surface is aligned, we can see a large disparity on the other sides of the box, as shown in the right image of figure 6. These artifacts are due to the lens distortions in the 2D images taken by our camera.

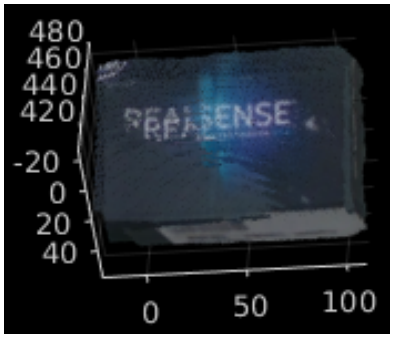

Figure 5: Point clouds alignment using ICP.

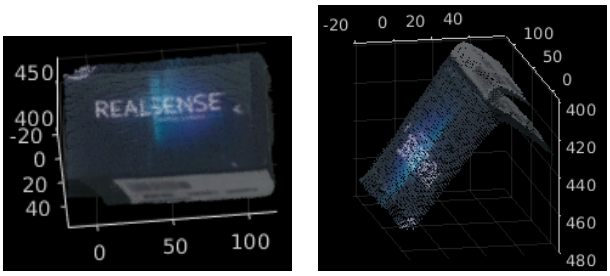

Figure 6: Point clouds alignment by selecting matching points.

3.2 Calibrated RGB Images

As an alternative approach to our problem, we have explored using the Matlab Camera Calibrator app which, given a checkerboard pattern in a 2D image, can estimate the desired intrinsic and extrinsic camera parameters. To accommodate this requirement, we have taken a new set of images that have a checkerboard pattern in each of them. The results from the Camera Calibrator app can be seen in figure 7.

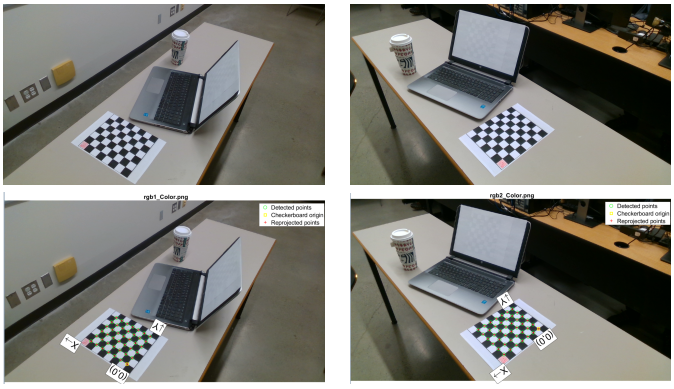

Figure 7: Sample RGB images taken by the Intel D415 camera (top row). RGB images after calibration (bottom row).

3.2.1 Algorithm. Following is a description of the camera motion estimation algorithm, which can be found in details here [9].

(1) Run the Camera Calibrator app to estimate both intrinsic and extrinsic camera parameters from the input images.
(2) Remove the lens distortion from all the input images, using the camera intrinsic parameters provided by the Camera Calibrator app.
(3) For each consecutive image pair, detect the points of interest using any of the feature detection algorithms (e.g. SURF, SIFT, or Harris Corner Detection), and match them between the image pairs.
(4) Estimate the relative camera poses using the matched point pairs and the intrinsic camera parameters.
(5) Set the first camera view as the world reference view.
(6) For each following camera view, transform its relative pose into the coordinate system of the first view.
(7) Store the current view attributes and inlier matches between the current and the previous views.
(8) Find matched points across all the processed views to get point tracks.
(9) Triangulate the point tracks to compute their corresponding initial 3D location.
(10) Refine the camera poses and the 3D points to minimize reprojection errors.

3.2.2 Result. We have found that the Camera Calibrator app requires at least 20 images to produce any results. In order to accommodate this shortcoming, we capture images of the scene in 10-15 degree intervals. The calibrator app, however, was still unable to detect a consistent checkerboard pattern across the input image data set, and rejected any images that produced a different number of checkerboard feature points. It follows that the algorithm requires a significant number of pictures taken in small degree intervals, with each image capturing each consequent between-the-frame movement. Due to these limitations, we have stopped pursuing this approach shortly after.

3.3 Texture-Mapped 3D Point Clouds

Using the Intel RealSense API, we were able to directly compute a target 3D point cloud for any given view, along with the corresponding RGB texture mapping. fig. 8 shows the point cloud in two angels.

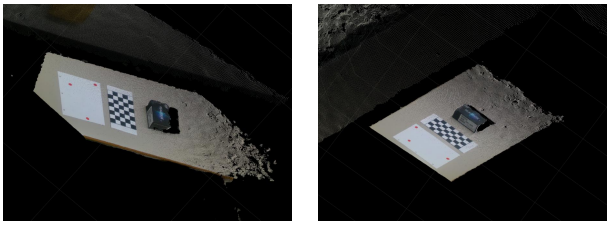

Figure 8: Sample point clouds taken by the Intel D415 camera

Having to operate on the input point clouds directly, we have decided to explore different ways we can correctly align them,

considering the limitations that make ICP algorithm not converge. Following are the constraints that we were required to keep under consideration:

- There is a limited number of point clouds that describe the whole scene. Each point cloud captures the scene from a different angle, with a total number of point clouds per scene averaging 8 point clouds in 45 degree intervals.
- Each input point cloud has a significant amount of noise, as well as a different number of total points. There is no guarantee that points from consecutive point clouds can be directly matched.

With these limitations in mind, we have decided to address the following problems at first:

- What is the minimum number of matching point pairs required to align two point clouds?
- Can the matching point pairs be used to find the affine transformation matrix between the given point clouds?

We found that we need at least three distinguishable point pairs that form a triangle pair in 3D space. Given such point pairs, we can use an absolute orientation algorithm, such as Horn's method [8], to find the target transformation matrix. Then, having such matrix, we can transform any moving point clouds into the fixed frame reference point cloud space.

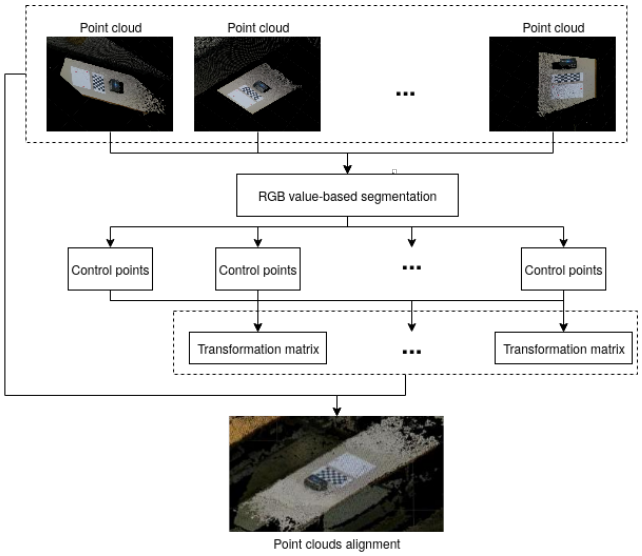

Figure 9: Reconstruction process from texture-mapped 3D point clouds.

3.3.1 Algorithm. Since the traditional segmentation and feature extraction algorithms do not work in 3D space, we have derived a custom algorithm to find and match control points between the consequent point clouds, (fig. 9):

(1) Ensure that the scene contains three separate control feature points forming a scalene triangle. The points must have a uniform coloring not found anywhere else in the scene (e.g. bright red).
(2) For each camera view point cloud, obtain the RGB color values for all points, and convert them into an HSV format.
(3) Using the HSV format properties[1], select all points that have a color range similar to the control points, and remove any outliers if any.
(4) Segment the selected points into three point cloud formation groups based on the distance threshold, and then find a centroid point for each group.
(5) Sort the centroids in a specific order based on their relative distance. In our implementation, we consider the point furthest away from the other two being an anchor point, with the other two points sorted based on their distance to the anchor point.[2]
(6) Save the sorted centroids for each point cloud, thus forming the matching point pairs between the camera view point clouds.
(7) Transform the moving point clouds into the fixed frame reference point cloud world space.

3.3.2 Result. Having found the matching point pairs, we can select one camera view point cloud to be a fixed frame reference world space, and use the absolute orientation algorithm to find the transformation between our fixed frame reference and the moving point clouds. Then, we transform the moving point clouds into the fixed frame reference point cloud world space, and obtain the final reconstructed scene in a 3D point cloud format. The final results obtained from the custom algorithm can be found in figure 10.

Figure 10: Custom algorithm final result from two different angles.

[1]Hue (H) can be used to determine the color range, while Saturation (S) and Value (V) can be used to determine the color intensity.
[2]This step ensures that the points are sorted in a consistent way for all camera view point clouds, and can be later used for alignment.

4 CONCLUSION

In order to reconstruct a 3D scene from multiple camera views, one usually require a considerable number of input frames to guarantee enough matching points for algorithms like ICP to converge. Therefore, most of the applications in this area utilize video data streams, and process them to prepare input frames with maximum overlap, which usually is a computationally expensive process.

Our approach determines the matching points by detecting a unique texture pattern in each input camera view. The detected pattern allows one to have a limited number of input frames capturing the scene in varying degree intervals.

5 FUTURE WORK

To extend this work, we can explore the following ideas:

- Update the reconstructed scene in real time by using Intel RealSense API to work on a video stream of input frames instead of separate camera view snapshots.
- Instead of using a predefined feature point color scheme, dynamically determine matching point pairs for all point clouds based on the overlapping colors in their texture mapping.
- Instead of using a single fixed frame reference for point cloud alignment, find the overlapping matching points for each consecutive point cloud pair. This approach would allow us to scan bigger environments, and remove the requirement of having the exact same matching feature points in all input point clouds.
- Alternatively, segment the target object from the scene, thus simplifying the alignment process and getting rid of any background noise.
- Generate a mesh with texture mapping from the reconstructed 3D point cloud.
- Build a Unity VR app to allow direct interaction with a reconstructed object or scene.

REFERENCES

[1] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohli, Jamie Shotton, Steve Hodges, and Andrew W. Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *10th IEEE International Symposium on Mixed and Augmented Reality, ISMAR 2011, Basel, Switzerland, October 26-29, 2011*, pages 127–136. IEEE Computer Society, 2011.

[2] Qian-Yi Zhou and Vladlen Koltun. Color map optimization for 3d reconstruction with consumer depth cameras. *ACM Trans. Graph.*, 33(4):155:1–155:10, 2014.

[3] Junho Jeon, Yeongyu Jung, Haejoon Kim, and Seungyong Lee. Texture map generation for 3d reconstructed scenes. *The Visual Computer*, 32(6-8):955–965, 2016.

[4] Rajesh Narasimha, Karthik Raghuram, Jesse Villarreal, and Joel Pacheco. Method for enhancing low quality depth maps for 3d reconstruction on a embedded platform. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*, pages 1538–1542, 2013.

[5] You Yang, Huiping Deng, Jin Wu, and Li Yu. Depth map reconstruction and rectification through coding parameters for mobile 3d video system. *Neurocomputing*, 151:663–673, 2015.

[6] Sriharsha Koundinya, Himanshu Sharma, Manoj Sharma, Avinash Upadhyay, Raunak Manekar, Rudrabha Mukhopadhyay, Abhijit Karmakar, and Santanu Chaudhury. 2d-3d CNN based architectures for spectral reconstruction from RGB images. In *2018 IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 844–851, 2018.

[7] Chen-Hsuan Lin, Oliver Wang, Bryan C. Russell, Eli Shechtman, Vladimir G. Kim, Matthew Fisher, and Simon Lucey. Photometric mesh optimization for video-aligned 3d object reconstruction. *CoRR*, abs/1903.08642, 2019.

[8] Berthold K. P. Horn. Closed-form solution of absolute orientation using unit quaternions. *Journal of the Optical Society of America A*, 4(4):629, Jan 1987.

[9] https://www.mathworks.com/help/vision/examples/structure-from-motion-from-multiple-views.html.