Lightweight Low-Poly Photogrammetry with Object Classification Validation Using OpenCV and PointNet

Tim Nguyen

Massachusetts Institute of Technology

May 5, 2023

Author Note

The author would like to give thanks to Jingnan Shi, Prof. Luca Carlone, Scott Balicki, Dr. Tina Kapur, and Dr. Alex Golby for their insight and mentoring. The author would also like to acknowledge Isa Gonzalez, Caleb Kohn, Ben Jacobson, Kimberly Nguyen, Sidney Trzepacz, Parker Hastings, Zoe Colimon, and Quyen Vo for their help with obtaining data for this project. This work was partially funded by the NSF CAREER award "Certifiable Perception for Autonomous Cyber-Physical Systems".

Abstract

Images in their purest form can only provide information in a two-dimensional space, which lacks the information that can be found in the three-dimensional space of the real world. Information such as the volumetric properties of the object is lost, and these factors are often a method used to help classify complex objects. To leverage the versatility that images provide and transform them into a medium with more detail, photogrammetry is a popular method to do the task. With programs like COLMAP(Schönberger & Frahm, 2016) and Meshroom(Griwodz et al., 2021), the user simply puts in the images of an object, and the program attempts to produce a 3D version of the object using keypoint detection. When paired up with 3D object classification, images can be transformed into 3D objects that can be used for robust classification. While both programs produce 3D objects with color, they require specialized hardware such as a dedicated GPU to run on them. Even without the need for dedicated GPUs, these programs still take a long time to perform reconstruction, These factors alone put a limit on the practicality of photogrammetry in places where image recognition is not adequate. I created a lightweight photogrammetry package that can run on a wide range of form factors, from laptops to Raspberry Pis, and can perform up to 26 times faster than COLMAP while producing up to 21 times more vertices. When leveraging the object classification network to validate the quality of the mesh, we can get an accuracy rate of up to 27% higher than COLMAP. This highlights that a high-quality and lightweight photogrammetry package is possible in the age of edge computing.

Keywords: Photogrammetry, Photo reconstruction, Optimization, Computer Vision, 3D Objects, Depth Perception, SIFT, ORB, PointNet, Object Classification, ML, Tensorflow

Lightweight Low-Poly Photogrammetry with Object Classification Validation Using OpenCV and PointNet

Contents

Introduction **4**

Methods **4**

Design Overview 5

Arduino Rigs 5

Turntable 5

Motorized Rail 6

Python Automated Reconstruction (PAR) 7

Camera Calibration 7

Keypoint Detection and Descriptor Matching 8

Essential Matrix and Recovering Pose 11

Bundle Adjustment and Surface Reconstruction 12

PointNet 13

Material 14

Data **15**

Results **15**

Data Analysis **18**

Conclusion **21**

Discussion 21

Future Direction 22

Introduction

From cancer research to archaeology, image recognition can be used to classify objects providing another dimension of information that cannot be conveyed efficiently using words. However, image recognition alone may not be powerful in certain use cases. For example, when classifying dinosaur fossils, image recognition may provide some level of detail that can differentiate species apart, but only full reconstruction can provide volumetric data for more advanced classification. Object reconstruction recovers missing depth information by generating 3D models. There are many techniques for reconstruction, including using LiDAR, photogrammetry, and ultrasonic scanning. All of these methods have their own benefits and constraints. Still, in order to create a lightweight program that takes full advantage of the reconstruction process, photogrammetry is the preferred method. Photogrammetry strictly depends on images as its input and does not require any special hardware besides a camera to perform the reconstruction. With the rise of mobile and edge computing, the need for fast and efficient programs that run advanced algorithms on relatively low-powered devices has increased. Mobile and edge computing plays a crucial role in expanding access to demanding computations. Existing software that performs photogrammetry like COLMAP(Schönberger & Frahm, 2016) and Meshroom(Griwodz et al., 2021) can perform robust object reconstruction while preserving texture and color. However, both programs often produce high-poly objects, which are hard to work with and require dedicated GPUs to perform the task. Utilizing the new and efficient methods for keypoint detection and bundle adjustment, it is possible to bring the computation capacity down to a much more accessible scale that is able to perform well in tight conditions.

Methods

This project contains three major components: building the photogrammetry package, creating tools to assist with data collection, and using a neural network model to verify the results of the object reconstruction.

Design Overview

The first major design component of this project are the camera rigs. For the project, two rigs are built: a turntable and a linear rail. Each of these rigs is powered by an Arduino and they accelerate the data collection as it automates parts of the process. The turntable allows a quick and easy way to scan small objects with a DSLR, while the linear rail is designed for larger objects. Both of these rigs can then be automated to run with the photogrammetry package for a seamless experience.

The next major component is called Python Automated Reconstruction (PAR), which is the photogrammetry package. PAR takes a sequential set of images to perform calculations to reconstruct a 3D object. The reconstruction process entails 5 major steps. The functionality of PAR will be more deeply explored further on. PAR first calibrates the camera, extracts key points from each image pair, recovers the pose of each camera and position of the points in the 3D space, performs bundle adjustment on the initial set of points, and finally performs surface reconstruction. These steps allow PAR the ability to create a 3D mesh from a set of images in a quick and efficient manner.

Lastly, in order to compare the results of the photogrammetry package with COLMAP, PointNet is used. PointNet is a neural network model that takes in a 3D point cloud as its input and can run either object classification or segmentation. For the purpose of this project, object classification is used as it compares the reconstructed object with the ModelNet40 dataset to see which object it is the most similar with. PointNet's prediction will be the main metric used to determine the mesh quality after reconstruction.

Arduino Rigs

Turntable

The first rig built was the turntable. The turntable contains a plate on which a small object can be placed, as seen in figure 1. A tripod is then mounted away from the turntable and is used to take pictures of the objects from all angles. The components and

parts for the rail are based on a project on Hackaday (Brocken, 2019). The turntable consists of an Arduino Uno that is connected to a stepper motor, joystick, and LCD. The stepper motor is passed through a DC step-down, as it gets the raw 12V power from the Arduino, and converts it to 5V for the stepper motor to operate on. The stepper motor is connected to a gear which allows it to control the direction and speed at which the plate is revolving. The LCD and joystick enable the option to go through the different modes available. The turntable can be user configured to have the plate revolve independently, or it could sync with a DSLR. These options allow increased flexibility for the user to gather data needed to perform reconstruction based on their operation constraints. The turntable can be configured to plug in directly with PAR or COLMAP using a simple Python script.

Motorized Rail

The motorized rail operates in a similar manner as the turntable. It also relies on the same Python scripts to control it. This motorized rail is based on a project featured in Adafruit. As shown in figure (Ruiz, 2015), the motorized rail consists of an Arduino Uno that is connected to a motor shield, which drives a stepper motor. The stepper motor is connected to a belt with a camera mount secured to it, as shown in 2. The camera mount sits on top of a metal rail, and the mount can slide across it. This design allows the camera to travel linearly over the whole length of the rail, which is approximately one meter long. This allows it to scan parts of larger objects, like sofas, one at a time. The rail can then be moved to the next part, and the camera can continue to scan, and so forth. The motorized rail can be controlled via serial communication through a laptop. Serial communication allows the ability for the user to specify the amount of stops the rail should make during its travel, and also allows it the ability to sync with the rotating plate to capture objects from various angles. Like the turntable, the rail can be configured to work with PAR or COLMAP using a Python script.

Figure 1

Image showing the Arduino turntable

Figure 2

Image showing the Arduino rig

Python Automated Reconstruction (PAR)

Camera Calibration

Calibrating the camera requires taking at least 10 photos from different angles of the checkerboard. These images will not be used for reconstruction, and this step only needs to be applied once for each camera. Since the shape and dimensions of a checkerboard are known, these known values can be plugged into OpenCV to help it produce the camera intrinsic matrix and distortion coefficients. The intrinsic matrix contains information regarding the camera focal length and offset. The camera focal length (f_x, f_y) determines the distance between the sensor and lens, and the camera offset (c_x, c_y) establishes the distance between the lens center from the sensor's top left corner(Simek, 2013).

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

Camera intrinsic matrix

Instead of a matrix, the distortion coefficients return a list of values describing the

lens distortion of the camera. There are two types of lens distortion that exist: radial distortion and tangential distortion. Radial distortion is the most common type of distortion found. This is when an image produced by a camera makes known straight lines look curved. Positive radial distortion makes an image have a "fish-eye lens" effect, while negative radial distortion has the opposite effect. Tangential distortion occurs when the camera sensor is not parallel to the lens. This creates a distortion effect where some parts of the image look further than they really are(Oršolić, 2020). Using this information, the list that OpenCV returns for the distortion coefficients takes into account all of these values. OpenCV returns the following list of values:

$$(k_1, k_2, p_1, p_2, k_3) \tag{2}$$

Distortion Coefficients

Where k_1, k_2, and k_3 describes the camera radial distortion, and p_1 and p_2 describes the tangential distortion. These values are important for the later steps as it allows PAR to properly map pixel coordinates of features onto real-world coordinates with minimal distortion(Gábor, 2022).

Keypoint Detection and Descriptor Matching

After taking in a sequence of photos of an object, keypoint detection, and descriptor matching are the next. These steps extract out features required to create a complete mesh. PAR grabs images in pairs in order to extract key points and descriptors. By taking in images in pairs, PAR can focus on finding and generating points between two images, and then can later put them all together to build a complete mesh. PAR can be configured to use Oriented FAST and Rotated BRIEF (ORB)("Orb (oriented fast and rotated brief)," n.d.), use Scale Invariant Feature Transform (SIFT)(Lowe, 1999) or BRISK (Leutenegger et al., 2011).

When comparing the speed of keypoint extraction, ORB and BRISK are significantly faster than SIFT. This can be attributed to ORB and BRISK's methods of

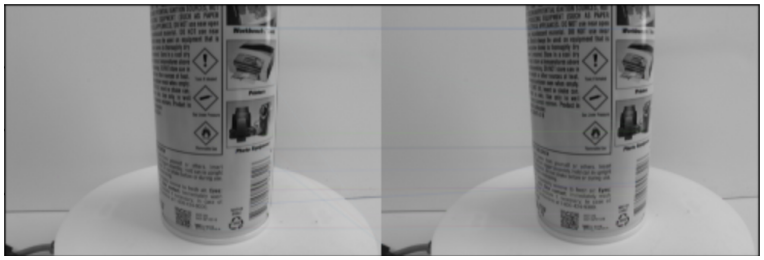

Figure 3

Image showing keypoint matches using ORB across two images

Figure 4

Image showing keypoint matches using BRISK across two images

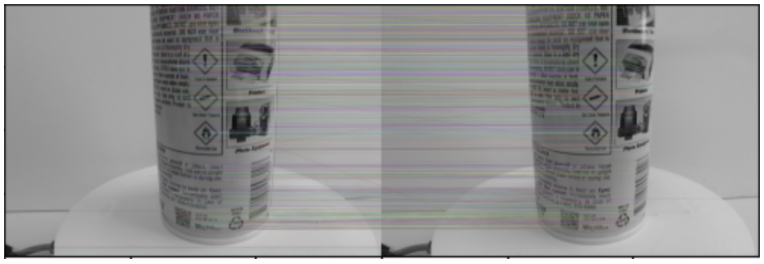

Figure 5

Image showing keypoint matches using SIFT across two images

keypoint extraction. ORB relies on FAST and BRIEF to perform keypoint and descriptor detection on both images. Image key points contain the pixel coordinates of a feature, while image descriptors are a vector containing the feature and the orientation of a point of interest. FAST first find pixels of interest and compares these pixels with their 16 neighbors("Fast algorithm for corner detection," n.d.). Once these neighboring pixels are found, FAST creates a color chart and looks at the local maxima and minima, which are then used to determine key points. These key points are then passed through BRIEF,

which recovers the orientation of the points and produces descriptors.

BRISK also relies on a similar method to perform keypoint extraction. It searches for keypoints by using a pattern of concentric rings that vary in size. In each ring, Gaussian smoothing and filtering are applied to filter out noise and this leaves with high-quality keypoints. The orientation of each keypoint is then found by extracting the angle from the sum of the gradients in the x and y direction. Once the orientation is found, BRISK can use it to create descriptors(Levi, 2013). When PAR is configured to use

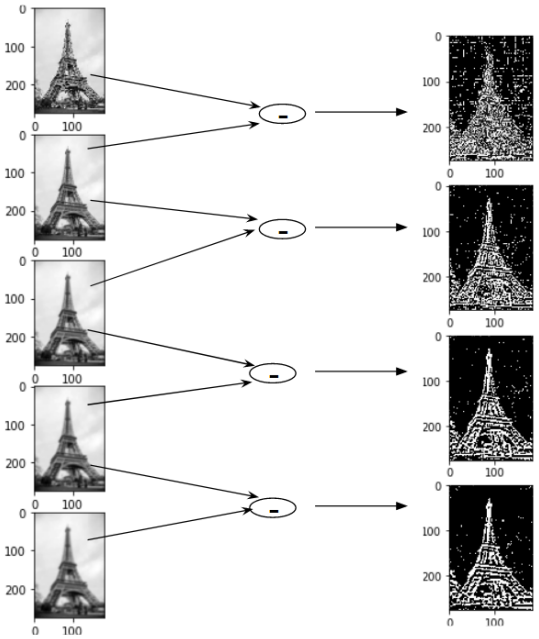

Figure 6

Image showing how SIFT detection uses Gaussian blurring to extract high-quality points(Singh, 2020a).

SIFT, it relies on a different process. SIFT also performs keypoint and descriptor detection on both images but instead utilizes image resizing and Gaussian blurring(Singh, 2020b). By blurring and resizing the image, SIFT is able to find important high-quality details that are not susceptible to noise. It then checks these points against the same images but with different blurring to determine the best points. Once these points are deemed to be of great significance are found, SIFT then checks the points one more time

and removes any points that are sensitive to noise or low contrast. These points are now keypoints. Like ORB, SIFT then determines the orientation of the keypoints and returns a list of descriptors.

In order to match the keypoints and descriptors from both images, PAR uses brute force to match those found through ORB, or FLANN when passed through BRISK and SIFT. Since ORB produces fewer keypoints, there might not be enough to be passed through FLANN. When using brute force, the descriptors from the pair's second image are compared with all of the descriptors of the first image to find matches. Matches done through FLANN rely on K-Nearest Neighbors (KNN). KNN creates branches of the descriptors of the first image and tries to find matches. Once the matches from both methods are found, they are passed through Lowe's ratio test (Lowe, 2004). In the ratio test, weak matches are removed and the corresponding keypoints are removed. By removing weak matches, the remaining high-quality keypoints and descriptors will then be passed on to the essential matrix to determine points and camera positioning.

Essential Matrix and Recovering Pose

High-quality keypoints from both images, intrinsic camera matrix and distortion coefficients are then used to form the essential matrix. The essential matrix is a 3 by 3 matrix that can be used to determine the positioning of points and the camera of two given images (Kitani, n.d.). It can be used to encode epipolar lines from the second camera view by multiplying the matrix with any given shared point. Epipolar lines are theoretical lines that connect points to the camera. Using the OpenCV command `recoverPose()`, the essential matrix and the keypoints can be passed through to determine the camera rotation matrix, translation vector, and triangulated points. The rotation matrix is again a 3 by 3 matrix that describes the second camera's rotation with respect to the first one. The translation vector is a vector that determines the second camera's position with respect to the first one. And finally, the triangulated points are a bunch of points with the keypoints translated into 3D world coordinates. The rotation

matrix, translation vector, and triangulated points can be used to create an initial estimate of the mesh by using the equation below.

$$...(R_{n-3}(R_{n-2}(R_{n-1} \times x_n + t_{n-1}) + t_{n-2} + t_{n-3})..$$ (3)

Using the previous set of rotation matrices and translation vectors to create an initial guess of the mesh

With R being the rotation matrix, t being the translation vector, and x being the triangulated points at a specific iteration, this transformation allows all of the points to be in relation with the previous points, which are known. This transformation allows the triangulated points to be transformed similarly to epipolar lines and allows all of these points to be based on the first image pair (K. Wu, 2017).

Bundle Adjustment and Surface Reconstruction

Bundle adjustment and surface reconstruction correct for camera position and the 3D points, and surface reconstruction connects the points to create faces. To perform bundle adjustment, each 3D point from the point cloud is reprojected back to the 2D space by using the intrinsic matrix, distortion coefficients, rotation matrix, and translation vector. This step is essentially performing camera pose recovery backward. The residuals between the reprojected 2D points and the corresponding coordinate of the matching keypoints are calculated. Besides calculating projection error, the residuals between the same 3D point across different vantage points are also calculated. All of these residuals are used to optimize all of the camera matrices, translation vectors, and 3D points by using the Scipy least square optimizer (Virtanen et al., 2020). Because calculating the Jacobian of the errors at the scale of bundle adjustment will take a lot of resources, Jacobian sparsity is used instead (Mayorov, 2016). Using all of these, the optimizer is able to find the best camera matrices, translation vectors, and 3D points that minimize the error. Because this process can go on forever, an error threshold is used, and for this project, it was set to 1^{-8}. Once bundle adjustment is completed and the optimized 3D points are

found, outlier rejection is then applied. Points that are determined to not be too far outside the cluster of points are removed. Lastly, after all of the high-quality points are extracted, surface reconstruction is performed. Using Open3D, PAR is able to take advantage of 2 different processes (Zhou et al., 2018). PAR can be configured to run alpha or Poisson surface reconstruction. Alpha surface reconstruction looks at each point cloud and does its best attempt to form faces with neighboring points. This method is quick and lightweight but can be susceptible to noise and often leaves the mesh with rough sides. Poisson reconstruction solves some of the pitfalls of alpha surface reconstruction. Poisson is named after the equation it uses, which is found with various camera processing features like high dynamic range and image editing (Kazhdan & Hoppe, 2013). Since Poisson doesn't use the radial method like alpha, it is able to make watertight faces of the mesh and it is more robust against noise. Poisson is able to create a mesh with smoother faces than alpha but at the slight expense of time. Both options are very powerful and are suitable for the varying parameters needed when it comes to filling in faces.

PointNet

One way to compare the quality of the mesh PAR produces with mainstream packages like COLMAP is with point cloud classification. Point cloud classification consists of a neural network that is capable of taking in 3D mesh and classifying it. Other methods of classifying 3D objects either convert the point cloud into a set of images or convert it into 3D voxels, which are then used to be classified (Qi et al., 2016). Breaking down a point cloud into images and converting them into 3D voxels are not good methods of classification as volumetric data is lost. PointNet solves this problem as it is a neural network architecture that can perform classification and segmentation with point clouds. It takes the point cloud and performs input and feature transformations. This standardizes the data and then allows it to run feature extraction on the mesh. Feature extraction allows us to differentiate between objects, and this can be taken a step further for segmentation. In segmentation, instead of a region of points classified, each point is

evaluated. For this project, PointNet was configured classify objects using the TensorFlow platform and is trained with the ModelNet40 database (Z. Wu et al., 2015) and LiDAR scans using PolyCam (Polycam Inc., 2022). ModelNet40 is a dataset that contains over twelve thousand 3D meshes that are split into 40 different classes. These classes range from airplanes to cups and are already split into training and validation sets. With the LiDAR scans, a group of volunteers used devices with a built-in LiDAR scanner and scanned various objects for training. During the training processing, a small subset of ModelNet40 and LiDAR data were used. This mixture in training allowed PointNet to train with super clean mesh from ModelNet40 and also real-world noisy data with LiDAR. Once PointNet is trained, this trained model is then used to predict the mesh that PAR produces. Comparing PointNet's prediction with the actual object label will be used as a means to quantify the quality of the mesh produced.

Material

For building the program, three different computers were used. The first one was a productivity laptop. Most of the program and testing were done on this laptop. This laptop ran on an ARM-based chip that had 12-core CPU, 30-core GPU, 32GB of RAM, and 1TB of storage. The next laptop used was a gaming laptop. This was used to run COLMAP dense reconstruction. It featured a 6-core CPU with a high-performance gaming graphics card. Lastly, a Raspberry Pi 4B was used. The Raspberry Pi featured a quad-core ARM CPU and 8GB of RAM (Pi, n.d.). It was used to prove that the PAR is able to run on limited power hardware.

The Arduino rigs have a lot of similar and different components. Both consist of a 3D-printed housing and are powered by Arduino Unos. Here is a list of the major components in each that differ.

Table 1: Arudino Rig Materials Comparison

Turntable	Motorized Rail
Small Stepper Motor	Large Stepper Motor
12v Motor Driver	Motor Shield
Bearings	Linear Rail
LCD Display	Camera Mount
Joy Stick	Timing Pulley

Data

A group of volunteers and I took collected in the form of pictures, video scans, and LiDAR scans. To ensure that the variable camera focal lengths are addressed, pictures were taken in fixed focus by either using the ultra-wide angle camera on a smartphone with macro focus disabled, using the selfie camera of older smartphones, or setting the focus of DSLR at infinity. Each person would first take a set of pictures of the camera calibration board. These images are then used to calibrate the camera. Once the camera calibration is performed, pictures and videos were taken of various objects that were in one of the ModelNet40 classes. LiDAR scans were also performed to create a wider range of noisy and real-world data. The LiDAR data would be cleaned up and then incorporated with a small subset of ModelNet40 data, and this was then passed through PointNet for training.

Results

In order to compare the quality of the objects being reproduced with PAR, scans of objects that are in ModelNet40 were used. These datasets went through PAR and were configured to use BRISK, ORB, and SIFT. A benchmark for these tests was COLMAP Sparse reconstruction as it had the closest process and time when it came to reconstruction. Both programs were tested on an ARM-based laptop that had 12-core CPU, 30-core GPU, and 32GB of RAM. The datasets tested varied in the amounts of photos. PAR completed an entire run without any modifications, while COLMAP was

modified to run surface reconstruction at the end. This allowed COLMAP to have the same level of detail and also allowed it to run through PointNet.

Table 2: Average Frame Processed Per Second During Reconstruction

Object (images)	COLMAP	PAR (BRISK)	PAR (ORB)	PAR(SIFT)
Bottle	0.57	1.64	1.08	1.06
Bowl	0.65	2.78	2.88	1.44
Cups	0.72	5.47	5.53	2.1
Toy Car	1.06	4.07	4.22	2.11

Table 3: Average Amount of Vertices Produced in Each Image

Object (images)	COLMAP	PAR (BRISK)	PAR (ORB)	PAR(SIFT)
Bottle	16.1	61.5	43.3	115.8
Bowl	17.7	70.1	51.4	76.5
Cups	11.8	0	19.5	113.0
Toy Car	3.9	34.4	33.1	59.1

Table 4: Mesh Matching Rate using PointNet Predictions

% of LiDAR in Training Set	COLMAP	PAR (BRISK)	PAR (ORB)	PAR(SIFT)
100%	31.7%	15.7%	46.9%	12.5%
77%	9.7%	21.1%	46.9%	14.3%
64%	0%	26.3%	28.6%	10.4%
54%	24.4%	52.6%	46.9%	31.25%
47%	31.7%	26.3%	40.8%	16.7%

For the following test, a random dataset is chosen and run both on a Raspberry Pi 4 and an ARM-based laptop. This is to compare the performance of the package on different hardware.

Table 5: Comparing Time to Reconstruct Between a Raspberry Pi and Laptop

	PAR (BRISK)	PAR (ORB)	PAR(SIFT)
Laptop	0:01:09	0:03:52	0:17:10
Raspberry Pi	0:14:39	0:28:09	2:53:53

Figure 7

Sample dataset image

Figure 8

PAR using ORB's result

Figure 9

PAR using SIFT's result

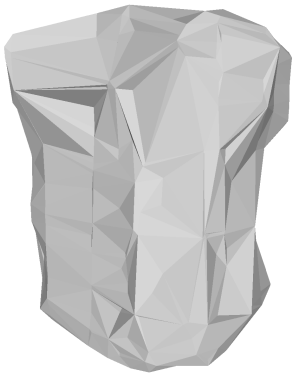

Figure 10

PAR using BRISK's result

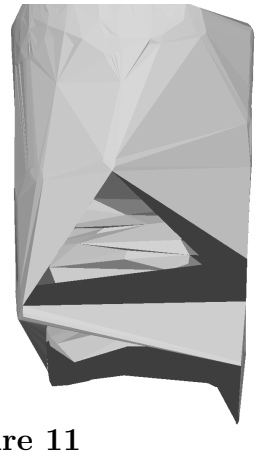

Figure 11

COLMAP Sparse result

Figure 12

COLMAP Dense result

Figure 13

These images show the results of running object reconstruction on a compressed air can with 30 sequential images.

Data Analysis

Looking at Table 2, PAR does much better across the board compared to COLMAP when it comes to speed. No matter what keypoint detector is configured, PAR does faster than COLMAP and is up to 7.7 times faster when comparing PAR (ORB)

with COLMAP. During the test, COLMAP also used up to 25GB of RAM which led the laptop to freeze for brief moments when performing reconstruction. Looking at PAR's results, BRISK and ORB either perform about the same as SIFT or significantly faster. This can be attributed to BRISK and ORB having similar methods of keypoint extraction, and this method is clearly faster than SIFT's Gaussian Blur. From the first test, PAR proves that it is possible to have speedy photogrammetry without the need for a lot of resources.

Next, looking at Table 3, PAR outperforms COLMAP when it comes to the number of vertices produced in each image, PAR is able to produce more vertices than COLMAP. SIFT leads when it comes to the amount vertices produced in each image. This again can be contributed to how Gaussian Blur is able to constantly and reliably produce high-quality key points in each dataset. ORB and BRISK perform similarly again, and this can be attributed to how both keypoint extraction programs are similar. When it comes to the cups that are reconstructed with BRISK, it is surprising how BRISK failed to reconstruct a single cup. This could be attributed to the way the cups are taken in the images. PAR is able to extract up to 15 more vertices per image when compared to COLMAP, ultimately highlighting the efficiencies of PAR.

Third, when looking at Table 4, the results highlight how sensitive neural networks are to noise and unknown data. LiDAR data is a great baseline to compare with as it is the same exact objects being reconstructed, and at some level comes with noise. Adding noise is crucial to have the model be able to train to various conditions, and ModelNet40 data helps train the model with similar objects. Neural networks can be accurate at times, but these results highlight that although PointNet is the best tool to quantify quality right now, it is not perfect. The matching rate varied across the board, and it could show that the model is not as resistant to noise as it should've been. However, PAR still outperforms COLMAP in most of the matching tests by a huge margin. Again, it shows that there can be huge leaps in efficiency without the need to sacrifice speed or quality when it comes to

photogrammetry.

Fourth, when looking at the performance comparison of running PAR on a Raspberry Pi and a laptop in Table 5, there is a big difference in performance between both computers. PAR when configured with BRISK took a little under 14 times slower on a Raspberry Pi, was about 7 times slower when configured with ORB, and about 10 times slower when configured with SIFT. Given the Raspberry Pi's limited processing power due to its form factor and its SOC, it is still amazing to see that photogrammetry could run on a Raspberry Pi. Other photogrammetry packages like COLMAP can't run on the Pi, and proving that PAR can run on the Pi shows that it is possible for photogrammetry to run on edge computing hardware if there are enough optimizations. This is a starting point, and it would be great to see if it can be optimized further for the Raspberry Pi.

Lastly, looking at the photogrammetry results in Figure 13 show a wide range of quality. Figure 7 shows one of the many photos in a random dataset that had 30 images of a can. Figure 8 shows ORB's attempt at reconstruction. For the most part, it does look like can to a certain degree. There are some artifacts on the sides, and this could be solved with a more fine-tuned outlier rejection algorithm. Looking at SIFT's attempt in Figure 9, it does seem to do the best job out of all of them. The shape of the can is more visible than the other methods. Not only are the faces look visibly smoother, but it also has a rounder shape and doesn't seem to have as many visible artifacts on the mesh. Next, looking at BRISK's attempt in Figure 10, the shape is somewhat there but there are some gaps in the mesh. The flat sides can be attributed to the fact that BRISK did not extract as many keypoints as the other methods, leaving it to have gaps in the mesh. Surprisingly, both of COLMAP's attempts do not seem to look any better than PAR's attempts. When looking at COLAMP Spare's mesh in Figure 11, there are a lot of weird faces and warping on the mesh. In the corners, it seems like COLMAP was able to partially get the shape. What is surprising about this is that COLMAP also uses SIFT keypoint detectors and it is very interesting that PAR with SIFT had a better attempt than COLMAP did. Some of

this warping can be attributed to the lack of keypoints found in that specific region of the mesh. Lastly, it is very surprising that COLMAP Dense's result in Figure 12 does not return a mesh that resembles a can at all. To produce this image, this dataset had to be run on the gaming laptop as it had the supported dedicated GPU needed for COLMAP, and it took over 4 hours to produce the following result. It is very surprising to see that COLMAP struggles to successfully reconstruct a can, despite the number of features it has when it comes to the texts and graphics on it. Once again, PAR proves that it is possible to have high-quality photogrammetry in a fraction of the time of existing implementations.

Conclusion

PAR shows that high-quality photogrammetry does not have to be limited to high-performance hardware or time. From the time it takes to perform reconstruction, the quality of meshes produces, and the vertices produced, PAR is able to outperform COLMAP while maintaining a speed, efficiency, and form-factor advantage.

Discussion

Looking back at PAR, there are some things that I wished I had done differently if I had more time. When training PointNet with ModelNet40 and LiDAR, I wished I had time to add noise and retrain PointNet with it. It would be interesting to see if that would make a significant impact on the accuracy scores. Adding noise to ModelNet40 in theory should make the model more accepting of noise. LiDAR data are pretty noisy compared to ModelNet40 data, but it would still be very interesting if this would greatly sway the predictions. Another approach I could've tried is using 3D anomaly detection models. These models can segment parts of the model that have gaps or holes, and it could be more efficient and accurate than PointNet. As it focuses on regions of error, it could also provide more useful information about the validity of the reconstructed model. Lastly, if I had more time, I would be interested to use GTSAM's implementation for bundle adjustments. GTSAM (Dellaert & Contributors, 2022) uses factor graphs and performs bundle adjustment by focusing on landmarks, and shared points across images. GTSAM is

used for various SLAM and robotics applications, and this approach is more advanced than what is currently implemented with Scipy. It would be interesting if there are any significant costs when it comes to time, and whether or not it creates a more accurate mesh.

Future Direction

In the future, it would be very interesting to see this project be used in everyday applications. Some examples where this project could make an impact are fossil detection, identifying items like screws and Legos, and segmenting through parts of a tree. This project could also be used in the medical field and biotech. Using the program to reconstruct certain proteins for drug targeting and discovery could be a potentially very powerful use of the project, I am also looking forward to finding ways to optimize this even further to make it more efficient and have it run faster on devices like Raspberry Pi.

References

Brocken, B. (2019). Arduino controlled photogrammetry 3d-scanner. https://hackaday.io/project/168301-arduino-controlled-photogrammetry-3d-scanner

Dellaert, F., & Contributors, G. (2022). *Borglab/gtsam* (Version 4.2a8). Georgia Tech Borg Lab. https://doi.org/10.5281/zenodo.5794541

Fast algorithm for corner detection. (n.d.). https://docs.opencv.org/3.4/df/d0c/tutorial_py_fast.html

Gábor, B. (2022). Camera calibration with opencv. https://docs.opencv.org/4.6.0/d4/d94/tutorial_camera_calibration.html

Griwodz, C., Gasparini, S., Calvet, L., Gurdjos, P., Castan, F., Maujean, B., Lillo, G. D., & Lanthony, Y. (2021). Alicevision Meshroom: An open-source 3D reconstruction pipeline. *Proceedings of the 12th ACM Multimedia Systems Conference - MMSys '21*. https://doi.org/10.1145/3458305.3478443

Kazhdan, M., & Hoppe, H. (2013). Screened poisson surface reconstruction. *ACM Trans. Graph.*, *32*(3). https://doi.org/10.1145/2487228.2487237

Kitani, K. (n.d.). 12.2 essential matrix - carnegie mellon university. https://www.cs.cmu.edu/~16385/s17/Slides/12.2_Essential_Matrix.pdf

Leutenegger, S., Chli, M., & Siegwart, R. Y. (2011). Brisk: Binary robust invariant scalable keypoints. *2011 International Conference on Computer Vision*, 2548–2555. https://doi.org/10.1109/ICCV.2011.6126542

Levi, G. (2013). https://gilscvblog.com/2013/11/08/a-tutorial-on-binary-descriptors-part-4-the-brisk-descriptor

Lowe, D. G. (1999). Object recognition from local scale-invariant features. *Proceedings of the seventh IEEE international conference on computer vision*, *2*, 1150–1157.

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, *60*(2), 91–110. https://doi.org/10.1023/b:visi.0000029664.99615.94

Mayorov, N. (2016). Large-scale bundle adjustment in scipy¶. https://scipy-cookbook.readthedocs.io/items/bundle_adjustment.html

Orb (oriented fast and rotated brief). (n.d.). https://docs.opencv.org/3.4/d1/d89/tutorial_py_orb.html

Oršolić, I. (2020). *Building a self-driving rc car* (Master's thesis). UNIVERSITY OF ZAGREB.

Pi, R. (n.d.). Raspberry pi 4 model b specifications. https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/

Polycam Inc. (2022, August 8). *Polycam - LiDAR 3D Scanner* (Version 2.2.18). https://apps.apple.com/us/app/polycam-lidar-3d-scanner/id1532482376?platform=ipad

Qi, C. R., Su, H., Mo, K., & Guibas, L. J. (2016). Pointnet: Deep learning on point sets for 3d classification and segmentation [cite arxiv:1612.00593Comment: CVPR 2017]. http://arxiv.org/abs/1612.00593

Ruiz. (2015). Bluetooth controlled motorized camera slider. https://learn.adafruit.com/bluetooth-motorized-camera-slider

Schönberger, J. L., & Frahm, J.-M. (2016). Structure-from-motion revisited. *Conference on Computer Vision and Pattern Recognition (CVPR).*

Simek, K. (2013). Dissecting the camera matrix, part 3: The intrinsic matrix. https://ksimek.github.io/2013/08/13/intrinsic/

Singh, A. (2020a). https://cdn.analyticsvidhya.com/wp-content/uploads/2019/09/Screenshot-from-2019-09-25-14-18-26.png

Singh, A. (2020b). Sift: How to use sift for image matching in python. https://www.analyticsvidhya.com/blog/2019/10/detailed-guide-powerful-sift-technique-image-matching-python

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R.,

Larson, E., ... SciPy 1.0 Contributors. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, *17*, 261–272. https://doi.org/10.1038/s41592-019-0686-2

Wu, K. (2017). Estimation of fundamental matrix. https://imkaywu.github.io/blog/2017/06/fundamental-matrix/

Wu, Z., Song, S., Khosla, A., Yu, F., Zhang, L., Tang, X., & Xiao, J. (2015). 3d shapenets: A deep representation for volumetric shapes, 1912–1920. https://doi.org/10.1109/CVPR.2015.7298801

Zhou, Q.-Y., Park, J., & Koltun, V. (2018). Open3D: A modern library for 3D data processing. *arXiv:1801.09847*.