Open3DGen: Open-Source Software for Reconstructing Textured 3D Models from RGB-D Images

Teo T. Niemirepo, Marko Viitanen, and Jarno Vanne
Ultra Video Group, Tampere University, Tampere, Finland
{teo.niemirepo, marko.viitanen, jarno.vanne}@tuni.fi

ABSTRACT

This paper presents the first entirely open-source and cross-platform software called Open3DGen for reconstructing photorealistic textured 3D models from RGB-D images. The proposed software pipeline consists of nine main stages: 1) RGB-D acquisition; 2) 2D feature extraction; 3) camera pose estimation; 4) point cloud generation; 5) coarse mesh reconstruction; 6) optional loop closure; 7) fine mesh reconstruction; 8) UV unwrapping; and 9) texture projection. This end-to-end scheme combines multiple state-of-the-art techniques and provides an easy-to-use software package for real-time 3D model reconstruction and offline texture mapping. The main innovation lies in various Structure-from-Motion (SfM) techniques that are used with additional depth data to yield high-quality 3D models in real-time and at low cost. The functionality of Open3DGen has been validated on AMD Ryzen 3900X CPU and Nvidia GTX1080 GPU. This proof-of-concept setup attains an average processing speed of 15 fps for 720p (1280×720) RGB-D input without the offline backend. Our solution is shown to provide competitive 3D mesh quality and execution performance with the state-of-the-art commercial and academic solutions.

CCS CONCEPTS

• Software and its engineering → Open-source model • Computing methodologies → Mesh models

KEYWORDS

3D model reconstruction, texture mapping, RGB-D acquisition, feature extraction, camera pose estimation, point cloud generation, mesh reconstruction

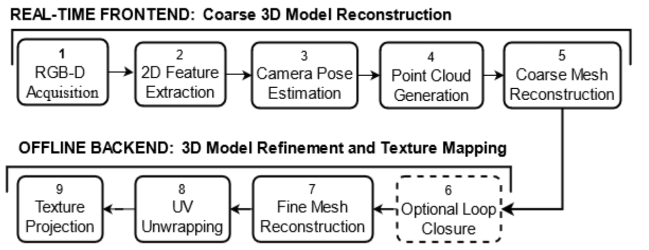
REAL-TIME FRONTEND: Coarse 3D Model Reconstruction
1 RGB-D Acquisition
2 2D Feature Extraction
3 Camera Pose Estimation
4 Point Cloud Generation
5 Coarse Mesh Reconstruction
OFFLINE BACKEND: 3D Model Refinement and Texture Mapping
6 Optional Loop Closure
7 Fine Mesh Reconstruction
8 UV Unwrapping
9 Texture Projection

Figure 1: Main processing stages of the Open3DGen pipeline.

ACM Reference format:

Teo T. Niemirepo, Marko Viitanen, and Jarno Vanne. Open3DGen: Open-Source Software for Reconstructing Textured 3D Models from RGB-D Images. In *Proceedings of ACM Multimedia Systems Conference (MMSys'21). ACM, September 28-October 1, 2021, Istanbul, Turkey.* https://doi.org/10.1145/3458305.3463374

1 Introduction

The recent advances in computer vision techniques and graphics hardware technology have led to a proliferation of high-fidelity photorealistic 3D models in various computer graphics applications. In photorealistic 3D imaging, a 3D surface of interest is translated into a precise textured 3D digital twin that can represent, e.g., lifelike avatars, realistic-looking objects, or immersive digital environments.

3D photorealism is increasingly gaining ground across a broad range of industries and businesses including video games, visualization, rendering, 3D printing, medical imaging, computer-aided design, architectural planning, history preservation, and marketing. Furthermore, real-time photorealistic reconstruction opens up various opportunities to exploit immersive *Extended Reality* (*XR*) technologies in interactive communication and collaboration platforms, live broadcasting, online social media, autonomous vehicles, robotics, and smart manufacturing.

Photogrammetry is a well-known technique for 3D reconstruction. The state-of-the-art photogrammetry applications like commercial Agisoft Metashape [1] and open-source AliceVision Meshroom [2] use various *Structure-from-Motion* (*SfM*) algorithms [3] to generate a textured 3D mesh from a set of RGB images. This type of 3D reconstruction software has the

Table 1: The main characteristics of prior art and our proposal.

Solution	Type	Input	Output	Texture proj.	Performance	Accuracy	Year	Open	License	Cross-Platform
Metashape [1]	Photogrammetry	RGB	3D Model	Offline	n/a	-	n/a	No	Proprietary	Yes
Meshroom [2]	Photogrammetry	RGB	3D Model	Offline	n/a	-	n/a	Yes	MPL2	Yes
LSD-SLAM [9]	SLAM	RGB	Sparse map	No	"real-time"	5cm	2014	Yes	GPL 3.0	Yes
ORB-SLAM [10]	SLAM	RGB	Sparse map	No	4 fps	2cm	2015	Yes	GPL 3.0	Yes
ORB-SLAM2 [11]	SLAM	Stereo RGB	Dense	No	6 fps	4cm	2017	Yes	GPL 3.0	Yes
ORB-SLAM3 [12]	SLAM	Stereo+Inertial	Sparse map	No	"real-time"	4cm	2020	Yes	GPL 3.0	Yes
Kimera [13]	SLAM	Stereo+Inertial	Segmented	No	20 fps	8cm	2020	Yes	BSD 2-clause	Yes
BundleFusion [14]	3D Scanning	RGB-D	3D Mesh	Yes *	"real-time"	n/a	2017	Yes	CC BY-NC-SA 4.0	No
VoxelHashing [15]	3D Scanning	RGB-D	3D Mesh	No	46 fps	n/a	2013	Yes	CC BY-NC-SA 3.0	No
Open3DGen	**Depth assisted**	**RGB-D**	**3D Model**	**Yes (Offline)**	**15 fps**	**< 1cm ****	**Current**	**Yes**	**MIT**	**Yes**

* point colors
** in typical 3D capturing scenarios

lowest barrier of entry as it only needs a normal color camera. However, photogrammetry is computationally extremely intensive and thereby not feasible for real-time or rapid-prototyping applications.

The landscape of 3D reconstruction technology also includes less-compute-intensive systems like *Simultaneous Localization and Mapping* (*SLAM*) [4]-[13] of which the fastest implementations are able to achieve real-time processing speed. However, SLAM solutions rarely provide a high-quality textured 3D model output since they are designed for robotic navigation and sparse room/area mapping.

In recent years, depth-sensing RGB-D cameras have come down to consumer-friendly price point. The high-resolution depth data can allow for more accurate and faster representation of the final 3D model or aid in the camera pose estimation. The existing RGB-D based approaches for 3D reconstruction, such as open-source BundleFusion [14] and VoxelHashing [15], are able to produce high-quality 3D meshes in real-time. However, they are both mainly designed for larger scale environmental 3D reconstruction, distributed under non-commercial licenses, and dedicated to Windows operating system only.

This paper proposes an end-to-end 3D reconstruction software called Open3DGen that adopts features from both SLAM and photogrammetry schemes in order to generate accurate photorealistic textured 3D models quickly from RGB-D images. To the best of our knowledge, it is the first fully open-source and cross-platform software implementation for 1) real-time 3D mesh reconstruction and 2) associated offline texture mapping.

Figure 1 depicts an overview of the proposed Open3DGen pipeline that incorporates all stages of the 3D capturing process in a modular and flexible way. The adopted image processing algorithms tend to be comprehensively studied and documented in the literature, but their implementations are often lacking in features, usability, or licensing. This work particularly addresses the implementation aspects of these algorithms and seeks to provide an open, accurate, and easy-to-use software package that works conveniently with consumer-grade RGB-D cameras.

The source code of this high-performance software package is available online on GitHub at

https://github.com/ultravideo/Open3DGen

It is being developed by Ultra Video Group at Tampere University. The source code is written in modern C++ and it is distributed under the permissive MIT open-source license. This cross-platform software can be run on all major platforms such as Windows and Linux, using its own *command line interface* (*CLI*).

The remainder of this paper is organized as follows. Section 2 investigates the feasibility and readiness of the existing approaches for 3D model capture, reconstruction, and texture mapping. Section 3 provides an overview of the proposed Open3DGen pipeline that is decomposed into a real-time frontend described in Section 4 and a non-real-time backend detailed in Section 5. In Section 6, the performance and quality evaluations of a proof-of-concept Open3DGen implementation are conducted, and the obtained results are benchmarked against the state-of-the-art photogrammetry software. Finally, Section 7 gives the conclusions and directions for future research.

2 Existing Software Implementations

Table 1 characterizes the most prominent software frameworks for 3D capture, reconstruction, and texture mapping. They can be classified into photogrammetry, SLAM, and RGB-D based 3D scanning approaches.

2.1 Photogrammetry Frameworks

The state-of-the-art photogrammetry approaches provide high-quality 3D meshes from a sequence of RGB images. Agisoft Metashape [1] is a commercial photogrammetry application accelerated with a *graphics processing unit* (*GPU*) for faster processing times. Its predecessor, Agisoft Photoscan, was probably the first fully-fledged photogrammetry software to reach wider adoption and it has also been considered the de-facto industry standard for years. AliceVision Meshroom [2] is a relatively new open-source photogrammetry software. It has been shown to have comparable quality with Metashape.

These frameworks only accept RGB images without additional depth data. Inferring accurate and high-resolution depth maps from image correspondences can be a very compute-intensive operation. Hence, their processing times are counted in hours or days, making their use in real-time or rapid-prototyping infeasible or even impossible with sparsely situated cameras.

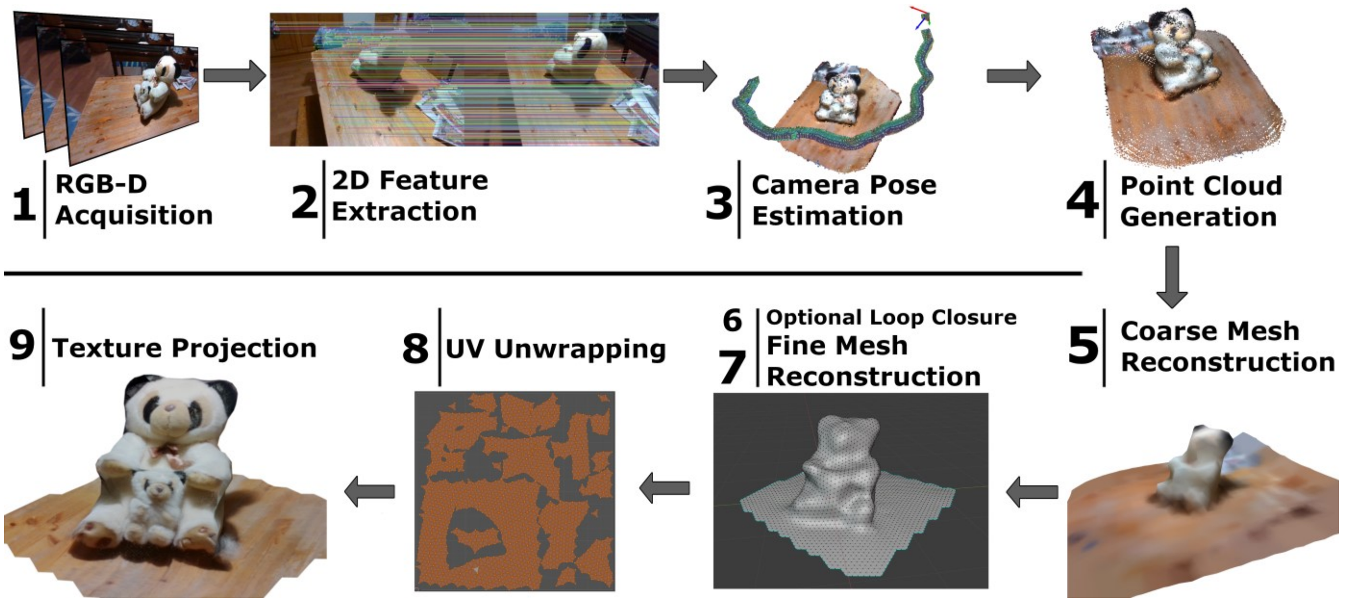

Figure 2: Visualization of the proposed Open3DGen pipeline processing with a test RGB-D image sequence.

2.2 SLAM Frameworks

A series of closed-source SLAM frameworks have been announced since early 1990s for indoor [4], [5] and outdoor mapping [6]-[8].

There are also a plenty of open-source frameworks out of which LSD-SLAM [9] is designed for large-scale mapping, ORB-SLAM [10] offers output with mono RGB input, ORB-SLAM2 [11] adds a stereo RGB and RGB-D support on top of ORB-SLAM, and ORB-SLAM3 [12] is the latest iteration of ORB-SLAM with support for additional inertial data input. They are all distributed under the GPL 3.0 license. However, none of them output actual 3D models but only location information.

BSD-licensed Kimera [13] is currently one of the few SLAM solutions with a 3D model output. However, it does not implement the texture mapping step and its reconstruction quality has not been evaluated on small scale models, where high precision is required.

2.3 RGB-D Based 3D Scanning Frameworks

Most of the existing RGB-D 3D reconstruction frameworks are commercial solutions like *Peel 3D Peel 2* [16] or *Artec Eva* [17] that tend to need a proprietary hand-held 3D scanner.

The most popular open-source solutions, BundleFusion [14] and VoxelHashing [15], are provided under non-commercial licenses and for Windows platform only. In addition, both of them are designed for larger scale and environmental 3D reconstruction rather than for high-fidelity reconstruction of very small and textured objects.

3 Open3DGen Reconstruction Pipeline

The proposed Open3DGen pipeline is dedicated for reconstructing high-quality photorealistic textured 3D models quickly and efficiently. The generated 3D models are ready to use as-is in various computer graphics and other 3D applications.

The proposed 9-stage pipeline can be divided into two main parts: 1) the real-time frontend for an RGB-D capture and a coarse 3D model reconstruction with vertex colors; and 2) the offline backend for the model refinement and texture mapping. The first five stages up to coarse mesh generation can be done in real-time to immediately provide the user with approximation of the reconstructed model. Figure 2 visualizes the operation of the pipeline with a test RGB-D image sequence of a teddy bear. The breakdown of the adopted algorithms and their implementations is listed in Table 2.

4 Open3DGen: Real-time Frontend

The real-time frontend of the pipeline consists of the following five main stages: 1) RGB-D acquisition; 2) 2D feature extraction; 3) camera pose estimation; 4) point cloud generation; and 5) coarse mesh reconstruction. These stages are detailed next.

4.1 RGB-D Acquisition

Our system uses 2D image features for camera pose estimation. Therefore, the input RGB-D images must be of high quality to be able to infer the camera movement trajectory correctly and with a needed accuracy. In practice, the images must be free of any significant motion blur and other unwanted artifacts, such as noise caused by low-light conditions.

Variance of Laplacian [18] can be used to preliminarily filter out images with high motion blur and the functions for that can be found in OpenCV [19]. However, this filtering step is not necessary if the camera is moving slowly in space and *scale-invariant feature transform* (*SIFT*) [20] features are used. Our experiments were conducted with the Intel RealSense D435 camera, which was already calibrated and undistorted.

Table 2: Adopted algorithms and their implementations.

Stage	Algorithm	Implementation
1. RGB-D Acquisition	-	RealSense SDK, Own
2. 2D Feature Extraction	SIFT [20], AKAZE [21], ORB [22]	OpenCV [19]
3. Camera Pose Estimation		
3.1. Feature - Landmark Matching	-	Own
3.2. Camera Pose Estimation	PnP [24]	OpenCV [19]
3.3. Initial Camera Pose Estimation	Essential Matrix Decomposition [25]	OpenCV [19]
3.4. Pose Verification	Linear Algebra	Own
3.5. Landmark-Candidate Tracking	-	Own
4. Point Cloud Generation	Triangulation	Own
5. Coarse Mesh Reconstruction	Poisson Surface Reconstruction [27]	Intel Open3D [30]
6. Loop Closure	Bundle Adjustment	GTSAM [29]
7. Fine Mesh Reconstruction	Poisson Surface Reconstruction [27]	Intel Open3D [30]
8. UV Unwrapping	Naïve, Mesh Parametrization	Own, Xatlas [32]
9. Texture Projection	Raycasting	Own
9.1 Texture Stacking	Averaging, Non-Zero	Own

Filtering noise from images is a less trivial task, so it advised to maintain good and uniform lighting conditions in the captured scene. This way, the end result also looks better and input images can be used without further manual work or texture adjustments.

4.2 2D Feature Extraction

For 2D feature detection, either AKAZE [21] or SIFT [20] features can be used as they proved to be robust with low-quality and noisy images. They also yielded the most stable temporal feature points with SIFT offering slightly more stable feature locations. In our testing, with better quality and higher resolution RGB images ORB features [22] were significantly faster while yielding only slightly less robust features. Features were matched with a brute force matcher with crosscheck enabled. The feature detecting and matching was done with OpenCV.

4.3 Camera Pose Estimation

Camera pose estimation in the 3D world can be further divided into the following six substages: 1) feature-landmark matching; 2) camera pose estimation; 3) initial camera pose estimation; 4) pose verification and invalid pose rejection; 5) landmark tracking; and 6) tracking new landmark-candidates.

4.3.1 Feature-Landmark Matching. The first substage is to find 3D points or landmarks in the scene, which correspond to the 2D features in the most recently acquired RGB-D image. Matching all frames pair-wise against each other is computationally intensive and cannot be implemented in real time, so a faster feature tracking method is adopted to obtain these feature-landmark pairs.

First, the system iterates over the current landmarks in the scene and collects the most recent 2D feature descriptors from them. Using these descriptors in feature matching guarantees the smallest possible difference in pixels between the frames, and thus increases the probability of good-quality matches. Secondly, these "landmarks descriptors" are matched against the feature descriptors of the new frame. From these matches, the corresponding 2D feature locations and 3D landmark points are collected for pose estimation.

The homography matrix [23] is used to filter the feature matches for outliers to guarantee the best possible camera pose accuracy and reconstruction quality. Because the homography matrix must be calculated between the current RGB-D frame and all frames corresponding to the matched landmarks, this step adds more delay than, e.g., radius outlier rejection, where matches would be rejected based on the Euclidean distance in screen coordinates. In practice, the delay introduced by the homography calculations was 1-30 ms per frame depending on the number of landmark matches. In most cases, this quality-performance tradeoff resulted in considerably more robust camera poses.

4.3.2 Camera Pose Estimation. The second substage is to estimate the camera pose using *Efficient Perspective-n-Point* (*EPnP*) [24] *Random sample consensus* (*RANSAC*) with the previously obtained 2D features and 3D points. This stage is standard for most *SfM* applications.

The recovered camera position and rotation are used to construct the camera projection matrix **P** as

$$\mathbf{P} = \mathbf{k}\ [\mathbf{R}^{\mathrm{T}} | (-\mathbf{R}^{\mathrm{T}}\ \mathbf{t})] \tag{1}$$

where **k** is the camera intrinsic matrix, **R** the rotation matrix, and **t** the position vector.

However, PnP cannot be used for the first two frames due to the absence of landmarks. Therefore, initial camera pose estimation is computed with the essential matrix [25]

$$\mathbf{E} = \mathbf{R} \times \mathbf{t} \tag{2}$$

It can be decomposed using *singular value decomposition* (*SVD*) to yield a relative rotation matrix and a relative translation vector with unit length between the two views. The essential matrix can be approximated using the 5-point algorithm detailed in [25].

After the camera translation and rotation have been resolved, the 2D features can be triangulated into the first landmarks.

4.3.3 Initial Camera Pose Estimation. The selected frames must be far enough apart in 3D space to achieve successful and accurate triangulation; if the Euclidean distance between the features is too small, the triangulation would produce an inaccurate point. On the other hand, the frames cannot be too far apart either since the number of trackable features decreases as the distance between the frames increases.

4.3.4 Pose Verification and Invalid Pose Rejection. The pose obtained using EPnP can sometimes be invalid or inaccurate, if the number of feature-landmark matches is not significant or if the distribution of the 3D landmarks is cluttered around a small area. Rejecting these poor-quality poses is important to maintain a high accuracy in the resulting 3D model and texture projection. Additionally, low-quality poses affect the subsequent camera poses and further decrease the quality of the end result.

A small time step between the frames can be assumed if the input is a real-time video feed with small camera movement and a reasonably high frame rate. This allows the next pose to be approximated by using velocity and acceleration obtained from the previous successful frames. The position and rotation of the next frame is approximated as

$$\mathbf{t}_{\text{approx}} = \mathbf{t}_{\text{prev}} + \mathbf{v}_{\text{prev}}\,\boldsymbol{\Delta t} + \frac{\mathbf{1}}{\mathbf{2}}\mathbf{a}_{\text{prev}}\boldsymbol{\Delta t}^2 \quad (3)$$

$$\mathbf{R}_{\text{approx}} = \mathbf{R}_{\text{prev}} + \boldsymbol{\omega}_{prev}\,\boldsymbol{\Delta t} + \frac{\mathbf{1}}{\mathbf{2}}\boldsymbol{\alpha}_{\text{prev}}\boldsymbol{\Delta t}^2 \quad (4)$$

where $\mathbf{t}_{\text{approx}}$ and $\mathbf{R}_{\text{approx}}$ are the approximated location and rotation of the next frame, $\mathbf{v}_{\text{prev}}$ and $\boldsymbol{\omega}_{prev}$ are the linear and angular velocity of the previous frame, $\mathbf{a}_{\text{prev}}$ and $\boldsymbol{\alpha}_{\text{prev}}$ are the linear and angular acceleration of the previous frame and Δt is the timestep between the frames. This approximation is extremely lightweight and reasonably accurate if Δt is small.

The obtained approximation is then used to calculate the Euclidean distance between the PnP location and approximated location, and the angular distance between the rotations. These distances are used to assign a quality score to every PnP pose, and any pose below a threshold is rejected.

4.3.5 Landmark Tracking. If the pose was verified to be of good quality, the inlier features and landmarks are collected from the result of the PnP algorithm. The inlier features are then appended to the corresponding landmarks.

4.3.6 Tracking New Landmark-Candidates. For longer sequences, new landmarks must be created continuously. The features that cannot be matched against landmarks are collected and matched against "candidate-landmarks", which have yet to be seen from the viewport of enough cameras in order to obtain a robust triangulation. All successful feature candidate matches are appended into the candidate track. The left-over features that were not able to be matched against either landmarks or candidates are converted into new one-frame-long candidates.

After every frame, the candidates are iterated over and all tracks meeting a certain length requirement are triangulated into landmarks. The number of candidates are kept low for higher performance by removing candidates that have lost tracking and are not able to be triangulated.

4.4 Point Cloud Generation

The camera intrinsic parameters are used to project point clouds pixel by pixel from the depth images to 3D points $[x_{local}, y_{local}, z_{local}]$ in the local space as

$$x_{local} = (u - c_x) \times \frac{z_{local}}{f_x} \quad (5)$$

$$y_{local} = (v - c_y) \times \frac{z_{local}}{f_y} \quad (6)$$

where c_x and c_y are the camera principal points, f_x and f_y are the camera focal lengths, u and v are the image 2D coordinates, and z_{local} is the depth value [26].

The resulting point cloud is origin-centered with identity-rotation. It is then transformed into the correct world position by matrix multiplication as

$$\mathbf{p} = \mathbf{T}\ \mathbf{p}_{local} \quad (7)$$

where $\mathbf{p}_{local} = [x_{local}, y_{local}, z_{local}, 1]$ is a point from the point cloud and $\mathbf{p} = [x_{world}, y_{world}, z_{world}, 1]$ is the newly transformed world point. $\mathbf{T}$ is the corresponding camera's transformation matrix $\mathbf{T} = \mathbf{R}|\mathbf{t}$, where $\mathbf{R}$ is the camera's rotation matrix and $\mathbf{t}$ is the camera's 3D position in world coordinates.

4.5 Coarse Mesh Reconstruction

A coarse and low-detail version of the mesh can be reconstructed with only vertex colors serving as texture data. The reconstruction is implemented with the Poisson algorithm [27] with a relatively low k-d search tree depth of 4 or 5 to save on performance. However, using the point cloud as feedback is often enough, and this step can be omitted.

5 Open3DGen: Offline Backend

The offline backend of the pipeline is made up of four stages: 1) loop closure and camera pose refinement; 2) mesh reconstruction; 3) UV unwrapping; and 4) texture projection. However, loop closure and camera pose refinement is an optional stage which was omitted in our testing and thereby excluded, e.g., from Figure 2 and Figure 4.

Between these four stages, the generated point cloud or 3D mesh can be exported for manual processing. Optimizing the mesh topology manually or using automated solutions, such as

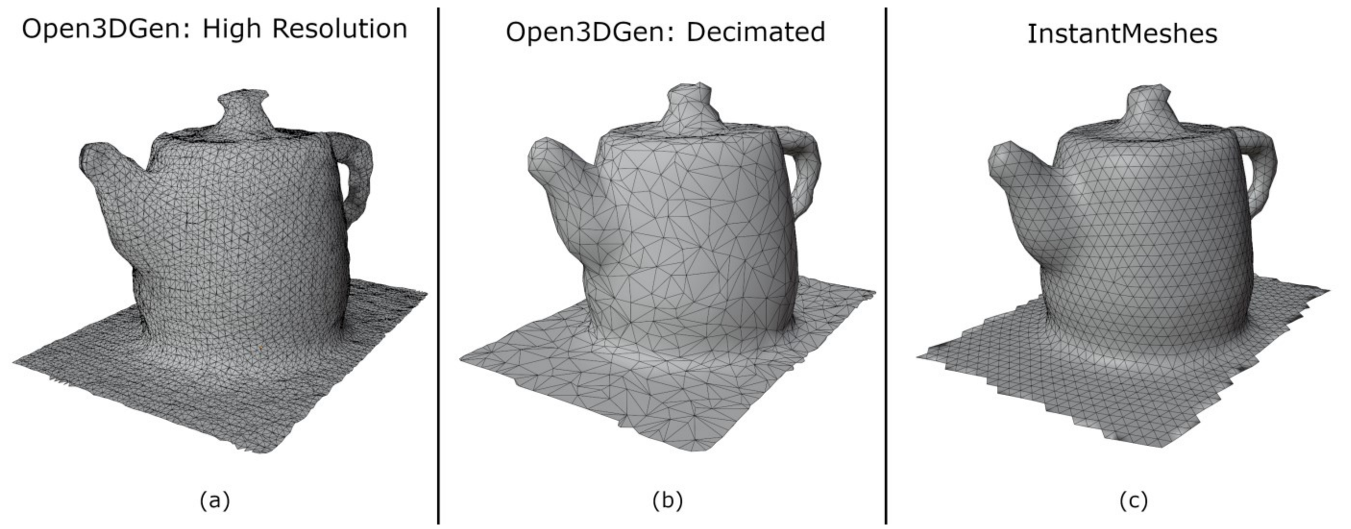

Figure 3: The mesh triangle topology of a teapot. (a) Before decimation. (b) After decimating. (c) After optimizing with InstantMeshes.

InstantMeshes [28], can lead to better results. The edited mesh can then be re-imported back into the pipeline for texture projection.

Editing the raw mesh or point cloud data before texture projection is also beneficial if the depth data is particularly noisy and has a lot of outliers. This is typical with reflective or highly light-absorbent surfaces.

5.1 Loop Closure & Camera Pose Refinement

Detecting and solving for loop closure is often a mandatory step in traditional SLAM applications but it was omitted from the real-time frontend of our system for better performance. Instead, our system can optionally refine the entire graph before mesh generation if the scene requires it. The real-time camera track is used to visualize the already reconstructed 3D scene to the user, and give feedback about what parts of the 3D model might still need to be scanned. In addition, the RGB-D sequences are usually quite short so the error cannot often accumulate enough to become noticeable.

The full graph optimization with bundle adjustment can be the most time-consuming step, especially if the camera track is long and noisy, but it is only necessary when dealing with very long capture sequences or particularly difficult indoor scenes with a small number of trackable features. As is, in typical 3D capturing scenarios the workflow includes orbiting the camera around the object-of-interest. Usually, this is not enough for the error to accumulate enough to be noticeable. Additionally, since the movement is uniform, it can easily be compensated for.

If the later frames do not contribute to the pose of earlier frames, the camera poses can start to drift. Bundle adjustment can combat this to a certain degree, but re-computing the camera poses pair-wise against all other cameras can give substantial accuracy improvements. This is especially accentuated in longer sequences, where the camera movement is not orbital around the object of interest. On average, this step takes as long as the real-time part of the system, but it is optional and can be omitted in most cases. Re-computing the camera poses is often preferred over running bundle adjustment, as it can be computationally lighter and correctly parametrizing the bundle adjustment can be a non-trivial task.

Georgia Tech Smoothing and Mapping Library (*GTSAM*) [29] was used to solve for loop closure and bundle adjustment.

5.2 Fine Mesh Reconstruction

The final high-detail 3D mesh is reconstructed using the Poisson algorithm [27] obtained from Intel Open3D [30]. This algorithm was chosen over others, such as ball pivoting [31], because of its speed and robustness when using point clouds with variable density of points. The Poisson surface reconstruction algorithm also yielded more smooth and organic looking meshes, resulting in considerably better texture projection results.

In order to capture small details of the 3D model, the mesh has to have a very high resolution. A typical output mesh can be seen in Figure 3 (a), consisting of 23972 individual triangles. To speed up the UV unwrapping and texture projection stages, the mesh can be decimated and brought down to a lower resolution, as in Figure 3 (b). The decimated resolution is 2126 triangles, and this step was done automatically. It can be desired to optimize the topology at this point of the reconstruction pipeline in order to create a better-looking 3D mesh and make texture projection results be more uniform. To achieve this, the high-resolution mesh was exported and optimized using InstantMeshes [28]. The result can be seen in Figure 3 (c), consisting of 4432 triangles.

Most automatic triangle count reducing algorithms are based on decimating or collapsing the mesh vertices. These are easy to implement and in nearly all cases the result is sufficient, but the output might suffer in its quality and smaller details can be lost in the process of decimation. In the case of geometry with complex holes and cavities, exporting the high-resolution mesh and manually retopologizing the result can often produce better results than relying on automatic solutions.

5.3 UV Unwrapping

The texture coordiantes of the reconstructed mesh, a.k.a. the UV - coordinates, can be unwrapped using two different implementations: a naive brute force approach where all of the

UVs are packed individually into a tight grid, or a smarter unwrapping algorithm using *Xatlas* [32].

5.4 Texture Projection

The textures are raytraced from the viewpoint of every camera using custom compute shaders. To avoid overlap, all views are projected onto separate textures. Afterwards, all textures are blended together to avoid invalid pixels and distortion due to parallel perspectives.

OpenGL compute shaders were chosen for projecting the textures due to their excellent performance in highly parallelized tasks and their cross-platform usability.

5.4.1 Texture Projecting Algorithm. The texture projection compute shader is dispatched per-pixel of the depth image. The shader uses the depth image and the UV -unwrapped 3D mesh for the projection. Additionally, the shader requires the camera intrinsic and extrinsic matrices as well.

Due to the difficult nature of the texture projection, the pseudocode for the algorithms can be found in **Algorithm 1**, where **CX, CY, FX, FY** are the camera intrinsic parameters, **VP** is the cameras view-projection matrix, and **TRIANGLES** contains the vertices and the edges of the reconstructed 3D mesh. The function *project_pixel* will be called for every pixel in the RGB image, in the OpenGL shader.

First, the shader verifies the depth pixel is valid. All invalid depth pixels have a value of zero. Next, the XY -coordinates of the depth pixel are used with the depth value to project the point into 3D space as point **p**. The principle is the same as was used in point cloud projection in section **4.4**. Then, every triangle of the 3D mesh is multiplied with the camera's inverse transform matrix, which brings the triangle into the projection camera's view. The projection camera is situated in the world origin. Finally, the triangle is intersected with a ray which goes from the world origin, as the camera uses pinhole camera model, into the previously obtained point **p**. If the intersection was successful and closer than any previous intersection of this triangle, the point intersection point can be set to be the closest.

After the triangles have been iterated over, the texture space UV -coordinate corresponding to the intersected triangle and the intersection point can be obtained using the distances from the triangle's corner vertices, projected on the triangle's normal. As the normal vectors of the mesh are not passed on to the shader, this step is done on the CPU after the projection is completed.

5.4.2 Texture Stacking. At this point, there are *n*-number of individual textures, corresponding to every frame's projection in 3D space. These textures must then be stacked properly. The two methods producing the best results are per-pixel average stacking and non-zero stacking.

In per-pixel average stacking, all valid pixels in the individual projection textures are averaged to form a coherent texture. Usually, this gave overall the best-looking results, as the effect of imprecisions in the camera positions was negated due to the averaging. The downside of averaging with inaccuracies is the slight blurriness of the texture.

In non-zero stacking, the projection textures' pixels are stacked only if the pixel in the destination texture is not yet valid. All subsequent pixels to that position are rejected. This yields the sharpest looking textures, but even the slightest amount of

Algorithm 1 Texture Projection Compute Shader

```
global CX, CY, FX, FY, VP
global TRIANGLES, ORIGIN, INVALID, LARGE_NUM
function pixel_to_world(coord, depth)
    return (
        (coord.x – CX) * depth / FX,
        (coord.y – CY) * depth / FY,
        depth)
end function
function intersect(origin, dir, v0, v1, v2)
    // triangle normal tr_n
    tr_n = normalize(cross(v1 - v0, v2 - v0))
    t = (dot(tr_n, origin) + dot(tr_n, v0)) / dot(tr_n, dir)
    if t < 0 then
        return INVALID
    end if
    p = ORIGIN + (t * dir)
    perpendicular = cross(v1 - v0, p - v0)
    if dot(tr_n, perpendicular) < 0 then
        return INVALID
    end if
    perpendicular = cross(v2 - v1, p - v1)
    if dot(tr_n, perpendicular) < 0 then
        return INVALID
    end if
    perpendicular = cross(v0 - v2, p - v2)
    if dot(tr_n, perpendicular) < 0 then
        return INVALID
    end if
    return p
end function
function project_pixel(i_id, RGBD)
    // shader invocation index i_id
    xy = (i_id % WIDTH, i_id / WIDTH)
    // world point world_p
    world_p = pixel_to_world(xy, RGBD[xy].depth)
    // view-projection inverse matrix vpi
    vpi = inverse(VP)
    ray_dir = normalize(world_p – ORIGIN)
    point = (0, 0, 0)
    length = LARGE_NUM
    for tr in TRIANGLES
        // triangle vertices v0, v1, v2
        v0 = vpi · tr.v0
        v1 = vpi · tr.v1
        v2 = vpi · tr.v2
        // hit point p
        p = intersect(ORIGIN, ray_dir, v0, v1, v2)
        if p != INVALID and len(p) < length then
            point = p
            length = len(p)
        end if
    end for
    return point
end function
```

fluctuation in the pose or the reconstructed 3D surface will result in un-usable textures.

The best-looking results were achieved by overlaying the non-zero texture over the per-pixel average texture and using a sharpening filter on the averaged texture.

6 Performance and Quality Analysis

In our experiments, the execution speed and camera pose accuracy of the proposed Open3DGen software was benchmarked with the first half (1500 frames) of the *Vicon Room 2* “easy” [33] and *Machine Hall* MH01-03 [33] datasets. These datasets are composed of 1280×720 black-and-white stereo images. Furthermore, the subjective quality of the resulting textured 3D model was evaluated with our own RGB-D test sequences.

The execution speed and reconstruction quality of Open3DGen were also compared with those of Metashape [1] and Meshroom [2] photogrammetry software. Instead, BundleFusion [14] and VoxelHashing [15] frameworks were excluded from the evaluation as compiling their source code was found unsuccessful due to unresolved CUDA compatibility issues with newer Nvidia graphics cards and the latest installation of Windows 10. All SLAM implementations were also omitted because none of them [6]-[13] is able to output textured 3D models.

6.1 Experimental Setup

The input RGB-D images for our experiments were taken with the Intel RealSense D435 camera. The benchmarking was conducted on a desktop workstation equipped with AMD Ryzen 3900X processor and Nvidia GTX1080 graphics card. The operating system was Ubuntu Linux 20.04.

6.2 Execution Speed Evaluation

Processing the first half of the *“Vicon Room 2”* and *“Machine Hall”* datasets with Open3DGen took 89 ms per frame on average, resulting in a frame rate of 11 *frames per second* (*fps*). These test runs were conducted with unlimited number of feature points, so the frame times were not consistent but content dependent.

In most cases, detecting 1000 features per frame guarantees a good camera pose. Setting the number of features per frame accordingly to 1000 resulted in the consistent frame time of 67 ms, i.e., the frame rate increased to $15\ fps$. However, keeping the number of features unlimited results in a more robust pose due to the increased number of points for the PnP algorithm.

6.3 Camera Pose Accuracy Assessment

With these same two datasets, the per-frame average camera pose estimation accuracies were $\approx 17\ cm$ and $\approx 24\ cm$, respectively. These results were achieved without camera pose refinement, loop closure, and bundle adjustment.

It is worthwhile noting that Open3DGen is not designed for large-scale reconstruction as these two datasets but shorter sequences with usually uniform movement around the object of interest. In these cases, the error is usually less than 1 cm. For example, the models depicted in Figure 2 and Figure 4 had an average error of 4 mm.

6.4 Subjective Quality Assessment

The subjective quality of Open3DGen was illustrated with the three test scenes depicted in Figure 4. They are called *Mossy rock* (249 frames), *Living room* (193 frames), and *Teapot* (261 frames). *Mossy rock* is a typical outdoor scene. It took 87 s to reconstruct, consisting of 24 s for the real-time part and 63 s for the texture projection. *Living room* is an indoor scene whose reconstruction took 88 s (15 s + 73 s). *Teapot* is a 3D model with a slightly reflective surface and a handle. The respective 3D mesh topology of the teapot can be seen in Figure 3 (b). Reconstruction time of a Teapot was 96 s (29 s + 67 s). The inconsistencies in the recorded times are due to the varying number of detected features in the real-time part and the triangle count of the mesh in the texture projection phase.

6.5 Comparison with Prior-Art

Figure 4 also illustrates 3D reconstruction quality obtained with Metashape and Meshroom frameworks that were adjusted for the highest available quality settings. Table 3 tabulates the respective processing times of these three benchmarked software and speedup of our Open3DGen over Metashape and Meshroom.

With *Mossy rock* and *Living room* scenes, the output quality of Open3DGen is equal to those of Metashape and Meshroom. The *teapot* scene illustrates well the shortcomings of traditional *SfM* methods; the models are not reconstructed correctly where the physical object had highly specular and reflective surfaces. Meshroom, and to a lesser degree Metashape, have a hole on the right side of the teapot. Meshroom also struggled with the floor in the *living room* scene.

Using calibrated depth data allows us not only to average the per-view point clouds, thus overcoming the issue of reflective surfaces, but to automatically compute the correct real-life scale of the 3D model as well. The photogrammetry software require trackable points to be able to compute a depth map; if the surface is reflective the reflection moves with the camera and therefore making depth map creation difficult. They cannot compute the scale automatically either, Metashape only allowing the user to manually set the size of an object and scale the output accordingly.

Neither Metashape nor Meshroom allow the direct editing of the results between all the sub-stages, supporting only removing unwanted points from the sparse and dense point clouds before generating the 3D mesh. Ours supports exporting, editing and re-import the edited results between all stages, after the real-time part of the system.

7 Conclusions and Future Work

This paper presented a fully open-source and cross-platform software called Open3DGen for reconstructing high-quality textured 3D models from RGB-D images. The proposed nine-stage Open3DGen pipeline is composed of a real-time frontend

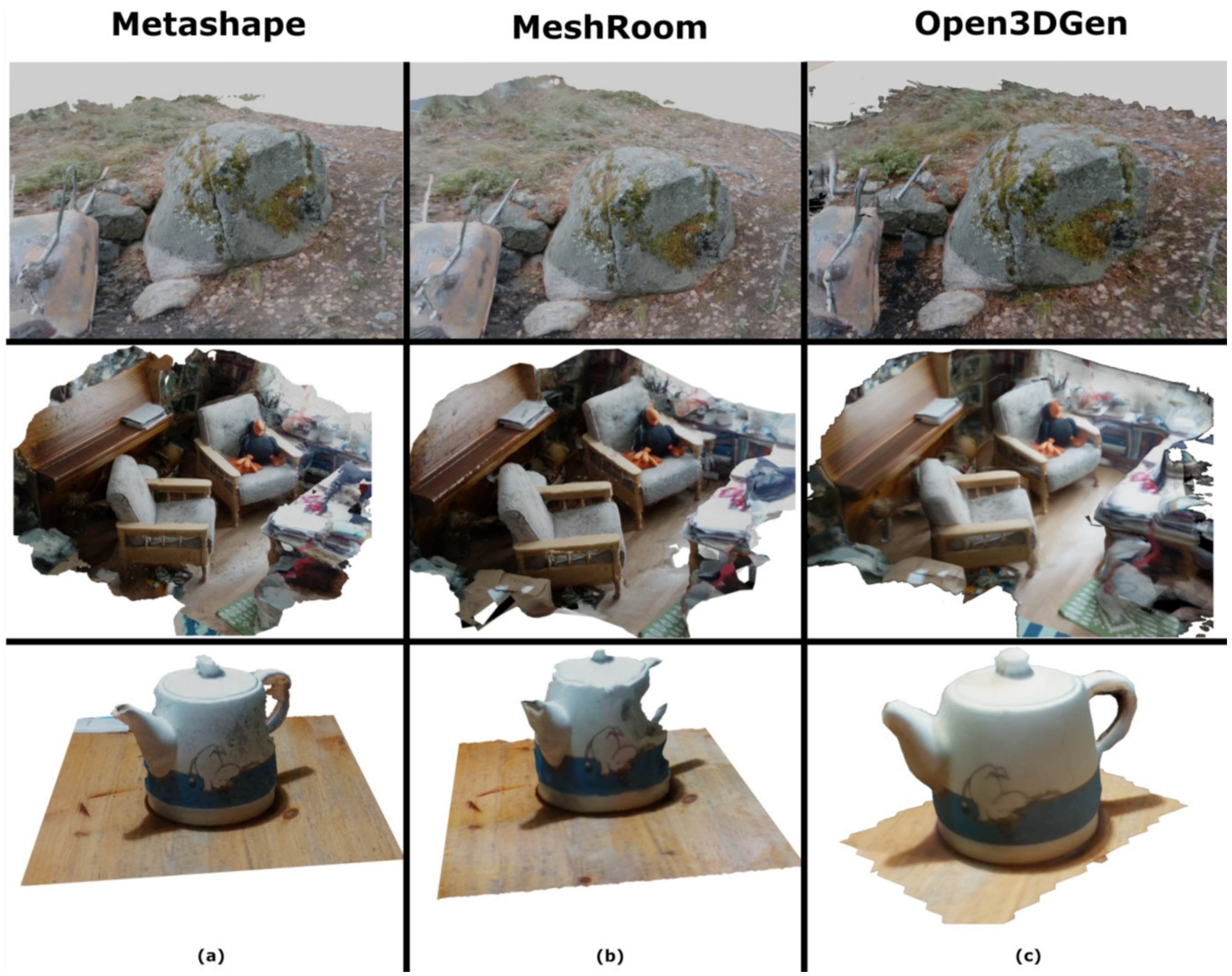

Figure 4: A comparison of reconstruction quality with three test scenes. (a) Metashape. (b) Meshroom. (c) Open3DGen.

Table 3: Performance comparison between Metashape, Meshroom, and Open3DGen.

Scene	Metashape [1]	Meshroom [2]	Open3DGen	Speedup
Outdoor (249 frames)	5530 s	3796 s	114 s	33× - 49×
Living Room (193 frames)	985 s	1220 s	88 s	11× - 14×
Teapot (261 frames)	2158 s	2630 s	96 s	23× - 27×

for a coarse 3D model reconstruction and an offline backend for model refinement and texture mapping. The system is shown to be robust with multifaceted inputs and in different operating conditions. The generated 3D models are ready to use as-is in various computer graphics and other 3D applications.

The proof-of-concept setup of Open3DGen was able to reconstruct a coarse 3D model from 720p RGB-D input at an average processing speed of 15 fps on AMD Ryzen 3900X CPU and Nvidia GTX1080 GPU. It was shown to achieve quality either comparable to or exceeding the state-of-the-art photogrammetry software in a fraction of the processing time.

In the future, the Open3DGen CLI will be upgraded to an intuitive *graphical user interface* (*GUI*) and the backend of the Open3DGen pipeline will be optimized for real-time processing. A completely real-time Open3DGen software could be used to take the user experience of the next-generation interactive and live applications to the next level of immersion.

ACKNOWLEDGMENTS

This paper is part of the NEWCONTROL project that has received funding within the ECSEL JU in collaboration with the European Union's H2020 Framework Programme (H2020/2014-2020) and National Authorities, under grant agreement 826653.

REFERENCES

[1] Agisoft Metashape. [Online]. Available: https://www.agisoft.com/.

[2] AliceVision Meshroom. [Online]. Available: https://alicevision.org#meshroom.

[3] S. Ullman, "The interpretation of structure from motion," *in Proc. R. Soc. Lond., B203*, Jan. 1979.

[4] P. Henry, M. Krainin, E. Herbst, X. Ren, and D. Fox, "RGB-D mapping: using Kinect-style depth cameras for dense 3D modeling of indoor environments," *in Proc. Int. Symp. on Experimental Robot.*, Dec. 2010, New Delhi and Agra, India.

[5] D. R. dos Santos, M. A. Basso, K. Khoshelham, E. de Oliveira, N. L. Pavan, and G. Vosselman, "Mapping indoor spaces by adaptive coarse-to-fine registration of RGB-D data," *IEEE Geosci. Remote Sens. Lett.*, vol. 13, no. 2, Feb. 2016, pp. 262–266.

[6] A. Nüchter, K. Lingemann, J. Hertzberg, and H. Surmann, "6D SLAM—3D mapping outdoor environments," *J. Field Robot.*, vol. 24, no. 8–9, Aug. 2007, pp. 699–722.

[7] P. Newman, D. Cole, and K. Ho, "Outdoor SLAM using visual appearance and laser ranging," *in Proc. IEEE Int. Conf. Robot. Autom.*, May 2006, Orlando, Florida, USA.

[8] D. M. Cole and P. M. Newman, "Using laser range data for 3D SLAM in outdoor environments," *in Proc. IEEE Int. Conf. Robot. Autom.*, May 2006, Orlando, Florida, USA.

[9] J. Engel, T. Schöps, and D. Cremers, "LSD-SLAM: large-scale direct monocular SLAM," *in Proc. European Conf. Comp. Vision*, Sept. 2014, Zürich, Switzerland.

[10] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardós, "ORB-SLAM: a versatile and accurate monocular SLAM system," *IEEE Trans. on Robotics*, vol. 31, no. 5, Oct. 2015, pp. 1147–1163.

[11] R. Mur-Artal and J. D. Tardós. "ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras," *IEEE Trans. on Robotics*, vol. 33, no. 5, Oct. 2017, pp. 1255–1262.

[12] C. Campos, R. Elvira, J. J. Gómez, J. M. M. Montiel, and J. D. Tardós, "ORB-SLAM3: an accurate open-source library for visual, visual-inertial and multi-map SLAM," *arXiv preprint arXiv:2007.11898*, July 2020.

[13] A. Rosinol, M. Abate, Y. Chang, and L. Carlone, "Kimera: an open-source library for real-time metric-semantic localization and mapping," *in Proc. IEEE Int. Conf. Robot. Autom.*, Aug. 2020, Paris, France.

[14] A. Dai, M. Niessner, M. Zollöfer, S. Izadi, and C. Theobalt, "BundleFusion: real-time globally consistent 3D reconstruction using on-the-fly surface re-integration," *ACM Trans. Graph.*, vol. 36, no. 3, June 2017, pp. 24:1–24:18.

[15] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger, "Real-time 3D reconstruction at scale using voxel hashing," *ACM Trans. Graph.*, vol. 32, no. 6, Nov. 2013, pp. 169:1–169:11.

[16] peel 2 3D scanner. [Online]. available: https://peel-3d.com/products/peel-2?variant=33046758522903.

[17] Artec Eva. [Online]. Available: https://www.artec3d.com/portable-3d-scanners/artec-eva-v2.

[18] S. Pertuz and D. Puig, "Analysis of focus measure operators for shape-from-focus," Pattern Recognition, vol. 46, no. 5, May 2013, pp. 1415–1432.

[19] OpenCV: Open Computer Vision Library. [Online]. Available: https://opencv.org/.

[20] D. Lowe, "Distinctive image features from scale invariant keypoints," *Int. J. Comput. Vis.*, vol. 60, no. 2, Nov. 2004, pp. 91–110.

[21] P. F. Alcantarilla, J. Nuevo, and A. Bartoli, "Fast explicit diffusion for accelerated features in nonlinear scale spaces," *in Proc. British Mach. Vis. Conf.*, Sept. 2013, Bristol, United Kingdom.

[22] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: an efficient alternative to SIFT or SURF," *in Proc. IEEE Int. Conf. Comp. Vis.*, Mar. 2011, Barcelona, Spain.

[23] O. Chum, T. Pajdla, and P. Sturm, "The geometric error for homographies," *Comput. Vis. Image Understanding*, vol. 97, no. 1, Jan. 2005, pp. 86–102.

[24] V. Lepetit, M. Moreno-Noguer, and P. Fua, "EPnP: an accurate O(n) solution to the PnP problem," *Int. J. Comput. Vis.*, vol. 81, no. 2, Feb. 2009, pp. 155–166.

[25] D. Nister, "An efficient solution to the five-point relative pose problem," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 6, June 2004, pp. 756–777.

[26] H. Aghajan and A. Cavallaro, "Multi-Camera Networks: Principles and Applications," *Academic Press*, Apr. 2009, Orlando, Florida, USA.

[27] M. Kazhdan, M. Bolitho, and H. Hoppe, "Poisson surface reconstruction," *in Proc. Eurographics Symp. on Geometry Process.*, June 2006, Cagliari, Sardinia, Italy.

[28] J. Wenzel, M. Tarini, D. Panozzo, and O. Sorkine-Hornung, "Instant field-aligned meshes," *ACM Trans. Graph.*, vol. 34, no. 6, Oct. 2015, pp. 189:1–189:15.

[29] F. Dellaert, "Factor graphs and GTSAM: A hands-on introduction," *Georgia Institute of Technology*, Sept. 2012.

[30] Q. Zhou, J. Park, and V. Koltun, "Open3D: a modern library for 3D data processing," *arxiv.org/abs/1801.09847*, Jan. 2018.

[31] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, and G. Taubin, "The ball-pivoting algorithm for surface reconstruction," *IEEE Trans. Vis. Comput. Graph.*, vol. 5, no. 4, Nov. 1999, pp. 349–359.

[32] Xatlas. [Online]. Available: https://github.com/jpcy/xatlas.

[33] EuRoC MAV Dataset. [Online]. Available: https://projects.asl.ethz.ch/datasets/doku.php?id=kmavvisualinertialdatasets.