Interactive RGB-D SLAM on Mobile Devices

Nicholas Brunetto, Nicola Fioraio, Luigi Di Stefano

CVLab - Dept. of Computer Science and Engineering, University of Bologna
Viale Risorgimento, 2 - 40135 Bologna, Italy
nicholas.brunetto@studio.unibo.it,
{nicola.fioraio,luigi.distefano}@unibo.it

Abstract. In this paper we present a new RGB-D SLAM system specifically designed for mobile platforms. Though the basic approach has already been proposed, many relevant changes are required to suit a user-centered mobile environment. In particular, our implementation tackles the strict memory constraints and limited computational power of a typical tablet device, thus delivering interactive usability without hindering effectiveness. Real-time 3D reconstruction is achieved by projecting measurements from aligned RGB-D keyframes, so to provide the user with instant feedback. We analyze quantitatively the accuracy vs. speed trade-off of diverse variants of the proposed pipeline, we estimate the amount of memory required to run the application and we also provide qualitative results dealing with reconstructions of indoor environments.

1 Introduction

In the past decades, the problem of Simultaneous Localization And Mapping (SLAM) has been mainly addressed by either expensive 3D sensors, *e.g.* laser scanners, or monocular RGB cameras. Laser scanners feature high precision and have enabled impressive results in the SLAM realm [1,2]. However, their size and expensiveness limit the range of addressable applications. On the other hand, monocular SLAM has reached a considerable maturity [3,4] but still mandates massive parallelization by GPGPU to attain dense 3D reconstruction [5].

The availability of consumer-grade RGB-D cameras capable of delivering color and depth information in real-time, such as the Microsoft Kinect and Asus Xtion Pro Live, has fostered a new approach in solving the SLAM problem. Nowadays, a newer generation of devices is appearing, *e.g.* the Structure sensor [6], the Creative Senz3D [7] and Google's Project Tango [8], which opens up new possibilities to introduce 3D sensing into the mobile world. However, most RGB-D SLAM systems have not been conceived for mobile platforms and typically rely on massive GPGPU processing to reach real-time execution on desktop environments [9].

Very recently, though, the SlamDunk [10] RGB-D SLAM algorithm has been proposed, which features a few threads, low memory consumption and can achieve real-time operation without any GPGPU acceleration. As SlamDunk is quite accurate and robust, it qualifies itself as a suitable reference design towards the creation of an RGB-D SLAM framework for mobile platforms. Hence, rather than investigating on a possible brand-new pipeline, in this paper we focus on the adaptation and implementation of the

key steps of the SlamDunk algorithm on an Android-operated tablet, underlining the difficulties faced and the solutions adopted to overcome them. Accordingly, our contribution may pave the way for further research on the novel topic of interactive RGB-D SLAM on mobile devices, as countless exciting applications and new scenarios may be envisioned should this technology trend grow and reach maturity.

The remainder of this paper is organized as follows. The next section reviews related publications, including SlamDunk and similar approaches, so to better motivate the adoption of the former as a SLAM reference design "to go mobile" as well as to highlight the key steps that will be adapted to enable the actual mobile implementation. Then, in Sec. 3 we briefly summarize the SlamDunk pipeline and in Sec. 4 present the software architecture deployed in the final SLAM mobile application. Sec. 5 discusses the adaptations made to the original algorithm, chiefly needed to improve efficiency and better meet the computational requirement of a mobile platform. Experimental findings are reported in Sec. 6, while in Sec. 7 we draw concluding remarks and point out the major issues to be addressed by future work.

2 Related Work

In the field of RGB-D SLAM, most state-of-the-art proposals can now produce high quality meshes in real-time. However, they usually require hardware acceleration by massive GPGPU processing together with high-end desktop platforms [9, 11, 12]. As such, these systems can hardly provide reference design guidelines towards rendering interactive SLAM feasible on resource limited platforms such off-the-shelf mobile devices. Among those not relying on GPGPU acceleration, RGB-D Mapping [13] has been one of the first proposals aimed at exploiting RGB-D sensing for SLAM. Their camera tracking is based on pairwise matching of image features, the system performing also global pose graph optimization to handle camera drift. To constrain nodes and make the optimization effective, they look for possible loop closures by matching image features within a subset of previous keyframes and perform a global pose graph relaxation accordingly. Therefore, as the explored space gets wider and the graph size increases, more time is spent in finding loop closures and optimizing poses. A similar approach is deployed by RGB-D SLAM [14, 15], where, moreover, near real-time processing (less than 15 FPS) is achieved by using the GPU for extraction of SIFT visual features. Nonetheless, the speed usually drops after gathering many frames due to the increasing complexity of the global optimization routine.

SlamDunk [10] is a very recent and particularly "lightweight" approach to RGB-D SLAM. Unlike previous work, it builds a Local Map against which the camera is robustly tracked and avoids full-graph optimization to preserve real-time operation even along large trajectories. Moreover, the Local Map enables implicit loop closure handling and allows for efficient operation as the camera moves back and forth within an already mapped portion of the workspace. As SlamDunk is computationally rather inexpensive and can achieve real-time performance (*i.e.* more than 30 FPS) without any GPGPU acceleration, it appears as an interesting approach for mobile RGB-D SLAM.

As regards SLAM systems specifically proposed for a mobile platform, we point out that they usually rely on the RGB camera of the device and the internal sensors,

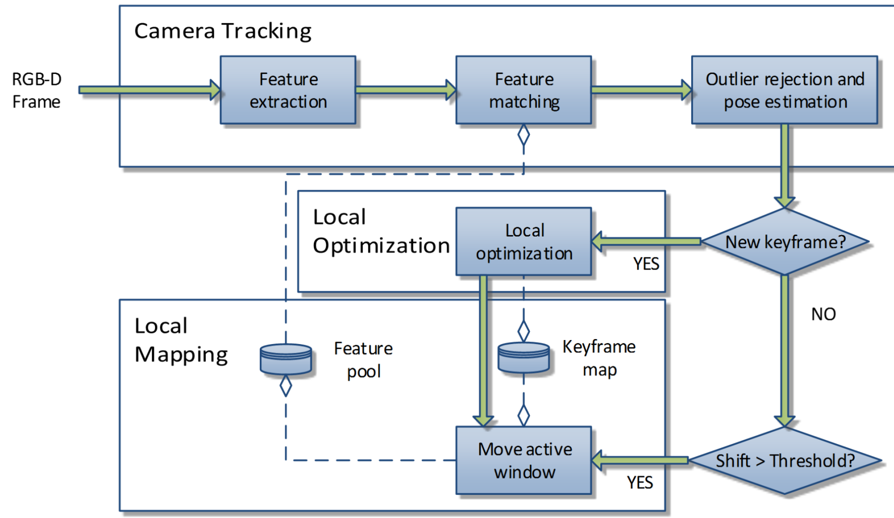

Fig. 1. The SlamDunk pipeline comprises three main modules: Local Mapping, Camera Tracking and Local Optimization.

e.g. accelerometer and gyroscope, while we target RGB-D data. Klein and Murray [16] showed how to run PTAM [4] on an Apple iPhone 3G. However, while tracking is typically carried out in 30ms, the bundle adjustment step may take seconds to complete. Lee *et al.* [17] process frames off-line on a remote server, so the computation is not performed on the device. Pan *et al.* [18] produces 3D reconstructions of very large scenes on a phone. The models are suitable for AR, but the system does not run in real-time and the user can perceive the result only after seconds of processing. More recently, Tanskanen *et al.* [19] achieved real-time tracking on a smartphone by leveraging on inertial sensing and GPU processing. Also, the system is designed for object capturing with model reconstruction running at about 0.3-0.5Hz. In this paper we demonstrate a much higher frame rate without exploiting any other sensors than the RGB-D camera.

3 SlamDunk System Overview

As depicted in Fig. 1, the SlamDunk algorithm [10] can be decoupled in three main modules: Local Mapping, Camera Tracking and Local Optimization. Local Mapping models the camera path as a collection of RGB-D keyframes and stores their poses within a quad-tree data structure. The algorithm does not consider the full path for tracking but, instead, selects a subset of spatially adjacent keyframes by retrieving an *Active Keyframe Window* from the quad-tree. Local visual features associated with such keyframes, *e.g.* SURF [20] or SIFT [21] descriptors, are gathered and stored into a local *Feature Pool*, which is a KD-Tree forest [22] enabling efficient feature matching.

The Camera Tracking module is the first to execute, taking the RGB-D frame coming from the sensor as input and returning the estimated camera pose. Purposely, visual

features are extracted from the RGB image and matched into the *feature pool* to find correspondences between the current frame and the local map. Using the associated depth measurements, matching pixels are back-projected in the 3D space leading to 3D correspondences. Accordingly, a full 6DOF pose can be robustly estimated by running a standard Absolute Orientation algorithm [23] within a RANSAC framework. Camera pose is represented as a 4×4 matrix in the following format:

$$\mathrm{T} = \begin{pmatrix} \mathrm{R} & t \\ \bar{\mathbf{0}}_3^\top & 1 \end{pmatrix} , \quad (1)$$

where R represents a 3×3 rotation matrix and t is a translation vector. Points are projected onto the image plane by means of the camera matrix

$$\mathrm{K} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} , \quad (2)$$

where f_x, f_y are the focal lengths and c_x, c_y the principal points. Then, given a pixel position $m = (u, v)$ and its associated depth measurement D, its 3D back-projection is computed as

$$p = \mathrm{K}^{-1} \cdot \begin{pmatrix} u \cdot D \\ v \cdot D \\ D \end{pmatrix} . \quad (3)$$

Then, as detailed in [10], if the currently tracked frame exhibits a limited overlap with respect to the Local Map, it is promoted as a new keyframe, thereby triggering the Local Optimization module and updating the Local Map itself, which, accordingly, gets centered around the newly spawned keyframe.

The Local Optimization module is in charge of optimizing poses across a pose graph associated with the keyframes to minimize the reconstruction error. The cost function is expressed as a sum of squared residuals, where each residual draws from a successful match:

$$r_{ij} = p_i - \mathrm{T}_i^{-1} \mathrm{T}_j q_j . \quad (4)$$

Here, (p_i, q_j) is the 3D point pair derived from a 2D feature match, while T_i and T_j are the associated camera poses. The optimization considers only the keyframe poses within a certain distance along the graph, in order to preserve local consistency while being able to scale smoothly to large workspaces. Indeed, a global pose graph optimization, though usually reaching a more accurate solution, would grow in complexity and time requirements with the number of keyframes.

4 Software Architecture

A global system overview of the proposed framework is shown in Fig. 2. The application consists of three main threads, namely, the *Grabber* (see Sec. 4.1), the *Application*

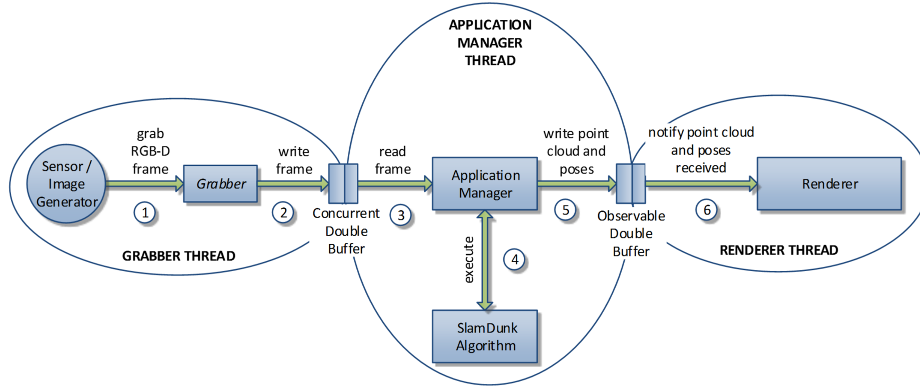

Fig. 2. The software architecture of the mobile application, divided into three main execution modules: Grabber, Application Manager and Renderer.

Manager (see Sec. 4.2) and the *Renderer* (see Sec. 4.3). The next sub-sections describe these components, focusing on their specific implementation on an Android mobile platform.

4.1 Grabber

The Grabber entity is used to abstract from the actual hardware device that would gather RGB-D frames. In this way, the other modules can execute independently and the specific kind of sensor will not influence execution of the SLAM algorithm. This entity is also used for testing purposes to emulate the behavior of an RGB-D camera. Different implementations of the actual Grabber may be easily included in the future, thus allowing for easy expansion of the range of sensors supported by the application.

The Grabber entity executes independently of the other components as a single thread which takes the data coming from the specific RGB-D sensor (step 1 in Fig. 2) and inserts them into a double buffer, called *Concurrent Double Buffer* (step 2 in Fig. 2). The choice of a double buffer is used in a multi-threaded environment to handle concurrency issues: the read operation on the buffer would block the associated thread if the data were not yet available, while the write operation frees the threads waiting for the data, if any.

4.2 Application Manager

The Application Manager is the most resource and time consuming thread, as it endlessly collects new data and runs the core SLAM algorithm, *i.e.* the adapted SlamDunk pipeline. First, RGB-D data are read from the input buffer previously filled by the Grabber (step 3 in Fig. 2). Then, the SlamDunk algorithm is executed (step 4 in Fig. 2). In case of tracking failure, the frame is discarded, otherwise the estimated camera pose is retrieved, together with updated keyframe poses had an optimization been run.

Should tracking be successful and the current frame promoted to keyframe, each pixel (u,v) on the RGB image is then expressed in 3D homogeneous coordinates and its associated depth value D is applied as:

$$\begin{pmatrix} u \cdot D \\ v \cdot D \\ D \\ 1 \end{pmatrix} , \tag{5}$$

without including the effect of the camera parameters. This is the only projection executed by the Application Manager, thus avoiding the multiplication with the inverse of the camera matrix and then with the pose estimated by the SLAM algorithm. Instead, we extend the inverse of the camera matrix to 4×4 as

$$\mathrm{K}_e^{-1} = \begin{pmatrix} \mathrm{K}^{-1} & \bar{\mathbf{0}}_3 \\ \bar{\mathbf{0}}_3^\top & 1 \end{pmatrix} \tag{6}$$

and produce the affine transformation which is given as input to the Renderer together with the partially projected points:

$$\mathrm{T}_f = \mathrm{T} \cdot \mathrm{K}_e^{-1} . \tag{7}$$

The point cloud generated by the projection, together with the model matrix and the other matrices representing updated poses, are inserted into the *Observable Double Buffer* (step 5 in Fig. 2). This buffer follows the Observer design pattern: the entity that observes the buffer, in this case the Renderer, is notified whenever the data changes. Accordingly, updates are promptly reported to the user interface as soon as new data become available.

4.3 Renderer

The final module in our mobile application is the Renderer, which manages the user interface. The Renderer is notified when a new keyframe has been detected or existing poses have been updated in the Observable Double Buffer (step 5 in Fig. 2). Then, it reads the points and the affine transformations (step 6 of Fig. 2) to show them properly on the screen of the mobile device. The Renderer draws the new point cloud as well as updates those that have their poses updated because of a local optimization.

To improve the overall efficiency, the actual multiplication between the points and the corresponding affine matrix has been delegated to the graphical pipeline. In our specific case, the rendering part has been developed using *OpenGL ES 2* [24].

5 SlamDunk on a Mobile Platform

The SlamDunk pipeline proposed in [10] has been modified to better match the capabilities of modern mobile devices. The main changes refer to the feature detection, description and matching approaches. Indeed, the visual features algorithms deployed

by the desktop version, *i.e*. SURF [20] and SIFT [21], cannot be run at interactive frame-rate on a modern mobile platform. Moreover, the descriptors generated by these algorithms are quite long (up to 128 floating-point numbers), so that a nearest neighbor search based on calculation of the Euclidean distance does not allow for sufficiently fast matching, further limiting speed on a mobile platform.

To overcome these limitations, several other feature detectors and descriptors have been considered. We list below the most promising variants and, in Sec. 6, compare them in terms of both accuracy and speed:

- ORB keypoint detector and feature descriptor [25];
- ORB keypoint detector and BRISK feature descriptor [26];
- Upright-SURF (U-SURF) optimized keypoint detector and BRISK feature descriptor.

The implementation of the above methods have been taken from the OpenCV library [27]. However, we have further optimized the U-SURF detector for a mobile platform by means of ARM NEON parallel instructions. It is noteworthy to point out that the feature descriptors chosen, namely ORB and BRISK, are binary descriptors that allow for replacing the Euclidean distance with the faster Hamming distance during nearest-neighbor search.

6 Experimental Results

In this section we present quantitative and qualitative results obtained by running the mobile version of SlamDunk with the different algorithms considered for feature detection and description. We also present an estimate of the amount of memory used by the application. Fig. 3 shows the user interface of the developed Android application. The reference mobile platform used in all the experiments is a Samsung Galaxy Tab Pro 10.1 tablet.

As for quantitative experiments, we have processed several sequences from the publicly available TUM RGB-D benchmark dataset [14], which includes color and depth streams acquired by a Microsoft Kinect and an Asus Xtion Pro Live, together with ground truth data obtained by a motion capture system tracking camera movements. Results are shown in Tab. 1. Clearly, the *U-SURF + BRISK* variant provides overall the best accuracy. We ascribe this to the quality of SURF keypoints, which, in particular, show higher repeatability than ORB features across viewpoint changes. Indeed, between the other two variants, *i.e*. *ORB + ORB* and *ORB + BRISK*, the difference is far less evident. These two variations can be considered equivalent in terms of accuracy, though they differ significantly in terms of timing performance, as discussed below.

As for the efficiency of feature algorithms, we have measured separately the mean time spent for detection and description (see Tab. 2). As expected, the ORB detector is much faster than U-SURF, due to the simpler pipeline of the algorithm. On the other hand, the BRISK feature descriptor is considerably more efficient than ORB. Indeed, it is also worth noticing that the combination *ORB + BRISK* is faster than *ORB + ORB*, the former taking an average time of about 35 milliseconds compared to the 100 milliseconds required by the latter. The combination *U-SURF + BRISK* runs at an approximate

Fig. 3. The user interface of the developed Android application.

Table 1. RMSE of the absolute trajectory error (meters) for several sequences of the TUM RGB-D benchmark.

Sequence	ORB + ORB	ORB + BRISK	U-SURF + BRISK
fr1/floor	0.058	0.055	**0.051**
fr1/desk	0.049	0.052	**0.042**
fr1/room	0.270	0.278	**0.140**
fr3/structure_texture_near	0.092	0.047	**0.025**
fr3/structure_texture_far	0.052	0.045	**0.028**
fr3/nostructure_texture_near_with_loop	0.046	0.057	**0.030**
fr3/nostructure_texture_far	0.178	0.139	**0.083**
fr3/long_office_household	0.058	0.063	**0.041**

speed of about 84 milliseconds. However, as shown above, it has the advantage of a notably higher precision. Therefore, we consider *U-SURF + BRISK* as the preferred feature detection and description approach for our current implementation of a SLAM system in a mobile environment, while *ORB + BRISK* turns out a more favorable choice than *ORB + ORB* due to faster speed and equivalent accuracy.

In Tab. 3 we highlight the efficiency of the other steps of the SlamDunk algorithm in terms of maximum and minimum execution times. Though the actual times vary depending on the observed scene, it is clear that graph optimization is by far the most time consuming step, especially when a high number of feature matches constrains the poses. However, a local optimization is run only when a new portion of environment starts being explored and a new keyframe is spawn; thus we are able to run the application at an average speed of about 10 frames per second using the *ORB + BRISK* variant, while *U-SURF + BRISK* can work at an average speed of about 6-7 frames per second.

Table 2. Average execution times (milliseconds) for the different feature combinations.

Algorithms	Detection	Description
ORB + ORB	**28**	72
ORB + BRISK	**28**	**7**
U-SURF + BRISK	70	14

Table 3. Execution times of the other SlamDunk modules.

Operation	Time of Execution
Feature Matching	20-40 milliseconds
Robust Pose Estimation	1-10 milliseconds
Local Optimization	20 milliseconds - 2 seconds or more

The final reconstructions can also be qualitatively inspected. Fig. 5 and Fig. 6 show two reconstructions from the TUM RGB-D benchmark, while Fig. 4 is an indoor environment explored by connecting an Asus Xtion Pro Live camera to our Galaxy Tab Pro 10.1 tablet. These reconstructions have been attained using the *U-SURF + BRISK* variant of the algorithm.

With regard to the memory footprint, we report now an estimate of the amount of memory required by the SlamDunk algorithm as well as the visualization module. Every keyframe stored in the map contains a list of features, each one having a length set by the algorithm used for the extraction. Also, for each feature we keep the 3D point representing the projected keypoint in camera space, expressed by 3 floating-point values, and an integer index used to find the corresponding keyframe during the feature matching operation. Considering both the ORB and BRISK algorithms, which define, respectively, a 32- and 64-byte descriptor, the memory needed by 100 keyframes is computed as follows:

$$M_{ORB} = (32\text{B} + 3 \cdot 4\text{B} + 4\text{B}) \cdot 100 \cdot 500 = 2400000\text{B} \approx 2.3\text{MB} \; , \tag{8}$$

$$M_{BRISK} = (64\text{B} + 3 \cdot 4\text{B} + 4\text{B}) \cdot 100 \cdot 500 = 4000000\text{B} \approx 3.8\text{MB} \; , \tag{9}$$

where we have considered an average number of 500 features extracted from each keyframe. It is worth to notice that the SIFT and SURF descriptors used by the desktop version have a much higher memory consumption, being the feature vector composed of 64 or 128 floating-point numbers.

As for the pose graph, for each node a quaternion and a translation vector are stored as 7 double-precision floating point numbers, coupled with a unique integer index; considering 100 keyframe, it amounts to

$$M_{poses} = (7 \cdot 8\text{B} + 4\text{B}) \cdot 100 = 6000\text{B} \approx 6\text{KB} \; . \tag{10}$$

Fig. 4. 3D reconstruction of a kitchen using the Asus Xtion Pro Live camera.

Fig. 5. 3D reconstruction of a wooden floor from dataset fr1/floor.

Fig. 6. 3D reconstruction of some objects from dataset fr3/structure_texture_far.

For each feature match a constraint is created and the corresponding 3D points are kept in double-precision format. Assuming an average number of 200 matches per keyframe during the tracking phase, we get

$$M_{matches} = 2 \cdot 3 \cdot 8\text{B} \cdot 200 \cdot 100 = 960000\text{B} \approx 938\text{KB} \ . \tag{11}$$

The optimization routine allocates space for the Hessian matrix as a collection of 6×6 double-precision blocks, one for each keyframe and one for each pair of keyframes connected by an edge in the graph. If each keyframe is connected, on average, to 10 other keyframes in the local map at the end of the tracking phase, we reach a memory usage of

$$M_{hessians} = 6 \cdot 6 \cdot 8\text{B} \cdot (1+10) \cdot 100 = 316800\text{B} \approx 310\text{KB} \ . \tag{12}$$

We are now able to estimate the memory usage of the entire SlamDunk algorithm for both ORB and BRISK features:

$$M_{SlamDunkORB} = 2.3\text{MB} + 6\text{KB} + 938\text{KB} + 310\text{KB} \approx 3.5\text{MB} \ . \tag{13}$$

$$M_{SlamDunkBRISK} = 3.8\text{MB} + 6\text{KB} + 938\text{KB} + 310\text{KB} \approx 5.0\text{MB} \ . \tag{14}$$

These values highlight the low memory requirement of the SlamDunk framework and, also, they are suitable for a mobile environment, which often shows limited memory availability. As for the storage and visualization of point clouds, for each 3D point

its position and its color have to be saved, the former as three single-precision floating-point values, the latter as three bytes accounting for the RGB components. Considering 100 keyframes at VGA resolution, we get:

$$M_{PointClouds} = 640 \cdot 480 \cdot (3 + 3 \cdot 4\text{B}) \cdot 100 = 460800000\text{B} \approx 439\text{MB} . \quad (15)$$

Although this value appears very high, usually not all the image pixels need to be projected into the 3D space. Indeed, sub-sampling the depth images by a factor of two can reduce the memory occupancy by 75%, *i.e.* ≈ 110MB.

7 Conclusions

We have presented an implementation of a state-of-the-art RGB-D SLAM algorithm on a mobile platform. The original pipeline has been modified and adapted to better cope with limited memory and hardware resources, leading to an application running at interactive frame-rate on an Android tablet. We have investigated on diverse feature detection and description approaches, thereby determining that the combination Upright-SURF and BRISK provides the preferred trade-off between reconstruction quality and responsiveness. Though our mobile SLAM application can run at about 6-7 Hz, or even faster when deploying ORB keypoints (10Hz), the timing performance of the local optimization step should be addressed in the future. The main challenge here is to keep a bounded maximum complexity, *e.g.* by marginalization of 3D point matches in terms of pose-to-pose constraints, while locally refining the trajectory and reducing the overall reconstruction error. Finally, we hope that our work may foster further research on the new topic of mobile RGB-D SLAM, so to devise soon more efficient solutions that could deliver real-time operation together with high reconstruction quality.

References

1. Borrmann, D., Elseberg, J., Lingemann, K., Nüchter, A., Hertzberg, J.: Globally consistent 3d mapping with scan matching. Journal of Robotics and Autonomous Systems **56** (2008) 130–142
2. Montemerlo, M., Thrun, S.: Large-scale robotic 3-d mapping of urban structures. In: International Symposium on Experimental Robotics (ISER), Singapore (2004)
3. Davison, A., Reid, I.D., Molton, N.D., Stasse, O.: MonoSLAM: Real-time single camera SLAM. Pattern Analysis and Machine Intelligence (PAMI), IEEE Trans. on **29** (2007) 1052–1067
4. Klein, G., Murray, D.: Parallel tracking and mapping for small ar workspaces. In: International Symposium on Mixed and Augmented Reality (ISMAR). (2007) 225 –234
5. Newcombe, R., Lovegrove, S., Davison, A.: DTAM: Dense tracking and mapping in real-time. In: International Conference on Computer Vision (ICCV). (2011) 2320–2327
6. Occipital Inc.: The Structure sensor. `http://structure.io/` (2014)
7. Creative Technology Ltd.: The Creative Senz3D sensor. `http://us.creative.com/p/web-cameras/creative-senz3d` (2014)
8. Google Inc.: Project Tango. `https://www.google.com/atap/projecttango/` (2014)

9. Newcombe, R., Izadi, S., Hilliges, O., Molyneaux, D., Kim, D., Davison, A., Kohli, P., Shotton, J., Hodges, S., Fitzgibbon, A.: KinectFusion: Real-time dense surface mapping and tracking. In: 10th IEEE International Symposium on Mixed and Augmented Reality (ISMAR), Washington, DC, USA (2011) 127–136
10. Fioraio, N., Di Stefano, L.: SlamDunk: Affordable real-time RGB-D SLAM. In: ECCV Workshop on Consumer Depth Cameras for Computer Vision, Zurich, Switzerland (2014)
11. Bylow, E., Sturm, J., Kerl, C., Kahl, F., Cremers, D.: Real-time camera tracking and 3d reconstruction using signed distance functions. In: Robotics: Science and Systems (RSS), Berlin, Germany (2013)
12. Whelan, T., Kaess, M., Leonard, J., Mcdonald, J.: Deformation-based loop closure for large scale dense RGB-D SLAM. In: International Conference on Intelligent Robot Systems (IROS), Tokyo, Japan (2013)
13. Henry, P., Krainin, M., Herbst, E., Ren, X., Fox, D.: RGB-D mapping: Using kinect-style depth cameras for dense 3D modeling of indoor environments. The International Journal of Robotics Research **31** (2012) 647–663
14. Sturm, J., Engelhard, N., Endres, F., Burgard, W., Cremers, D.: A benchmark for the evaluation of RGB-D SLAM systems. In: International Conference on Intelligent Robot Systems (IROS). (2012)
15. Endres, F., Hess, J., Sturm, J., Cremers, D., Burgard, W.: 3D mapping with an RGB-D camera. IEEE Transactions on Robotics (2013)
16. Klein, G., Murray, D.: Parallel tracking and mapping on a camera phone. In: International Symposium on Mixed and Augmented Reality (ISMAR), Orlando, Florida, USA (2009) 83–86
17. Lee, W., Kim, K., Woo, W.: Mobile phone-based 3D modeling framework for instant interaction. In: International Conference on Computer Vision Workshops (ICCV Workshops). (2009)
18. Pan, Q., Arth, C., Rosten, E., Reitmayr, G., Drummond, T.: Rapid scene reconstruction on mobile phones from panoramic images. In: International Symposium on Mixed and Augmented Reality (ISMAR). (2011)
19. Tanskanen, P., Kolev, K., Meier, L., Camposeco, F., Saurer, O., Pollefeys, M.: Live metric 3D reconstruction on mobile phones. In: International Conference on Computer Vision (ICCV), Sydney, Australia (2013)
20. Bay, H., Ess, A., Tuytelaars, T., Gool, L.V.: Speeded-up robust features (SURF). Computer Vision and Image Understanding **110** (2008) 346 – 359
21. Lowe, D.G.: Distinctive image features from scale-invariant keypoints. International Journal of Computer Vision (IJCV) **60** (2004) 91–119
22. Muja, M., Lowe, D.G.: Fast approximate nearest neighbors with automatic algorithm configuration. In: International Conference on Computer Vision Theory and Applications (VISAPP). (2009)
23. Arun, K.S., Huang, T.S., Blostein, S.D.: Least-squares fitting of two 3-d point sets. Pattern Analysis and Machine Intelligence (PAMI), IEEE Trans. on **9** (1987) 698–700
24. Segal, M., Akeley, K.: The opengl graphics system: A specification. `http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf` (2004)
25. Rublee, E., Rabaud, V., Konolige, K., Bradski, G.: ORB: An efficient alternative to SIFT or SURF. In: International Conference on Computer Vision (ICCV), Barcelona, Spain (2011)
26. Leutenegger, S., Chli, M., Siegwart, R.: BRISK: Binary robust invariant scalable keypoints. In: International Conference on Computer Vision (ICCV). (2011)
27. Bradski, G.: The OpenCV Library. Dr. Dobb's Journal of Software Tools (2000)