LSD-SLAM on a Mobile Device

Guanhang Wu
MSCV Program
Carnegie Mellon University
guanhanw@andrew.cmu.edu

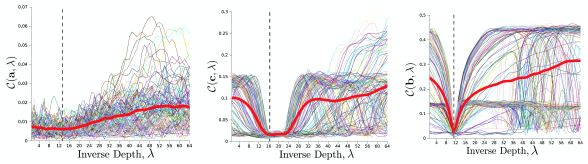
Jennifer Lake
MSCV Program
Carnegie Mellon University
jelake@andrew.cmu.edu

Abstract

In this project, an iOS app was created which uses the monocular camera to render an inverse depth map of the current scene and track the camera pose. This app, Mobile LSD, achieves an average frame rate of 31.2 frames per second (fps) on an iPad 2, 29.3 fps on an iPhone 6, and 15.4 fps on an iPhone 5. Mobile LSD uses the Visual Odometry subset of the Large-Scale Direct Monocular SLAM (LSD SLAM) algorithm to calculate the inverse depth map [3]. LSD SLAM is a direct method and creates a semi-dense inverse depth map as shown in figures 7, 5, and 6.

1. Introduction

LSD SLAM is a direct method, which does not detect features or calculate feature descriptors. Instead, it utilizes the raw image pixel intensities between the current frame and the keyframe to identify correspondences. These correspondences can be noisy, however a consensus can be formed from multiple frames to get a good estimation for the inverse depth of each point, as shown in figure 1.



(a) (b) (c)

Figure 1: Examples of the effect of texture in direct methods. Sub-figure (a) is region with little texture. Sub-figure (b) is region with edge- like texture. Sub-figure (c) is region with corner-like texture. Image credit:[9]

Feature detection and description can be costly algorithms and by omitting them, a more computationally efficient solution is found. This is a very important property for mobile algorithms, in which CPU usage and battery power are constrained. The direct benefit of of this efficiency is that a semi-dense map can be calculated in real-time on a mobile device. This semi-dense map can be used to render a 3D mesh of the scene, over which virtual objects can be placed.

1.1. Algorithm

The Visual Odometry algorithm contains two main parts: tracking (pose estimation) and inverse depth estimation, as shown in figure 2.

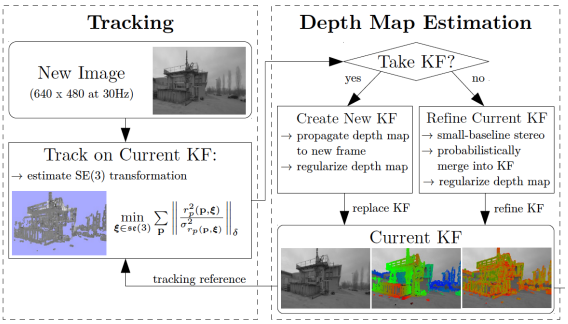


Figure 2: LSD Visual Odometry. Image credit: [3]

In pose estimation, the objective is to minimize the photometric error E between two frames, which are associated by a warp function, given by equation 1. In this equation, E represents the photometric error, I_{ref} represents the reference frame, and I represents the current frame. ω is the warp function which is calculated using equation 2. This equation takes p_i, the image point, D_{ref}, the inverse depth, and ξ, the camera matrix, as input and produces the warped image coordinates as the output.

The photometric error E is minimized using Gaussian-Newton algorithm.

$$E(\xi) = \sum_i (I_{ref}(p_i) - I(\omega(p_i, D_{ref}(p_i), \xi)))^2 \qquad (1)$$

$$\omega(p, d, \xi) = (\frac{x}{z}, \frac{y}{z}, \frac{1}{z})^T$$
$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = exp_{se(3)}(\xi) \begin{bmatrix} p_x/d \\ p_y/d \\ 1/d \\ 1 \end{bmatrix} \quad (2)$$

After the pose is estimated, the inverse depth is calculated using equation 3, where $\lambda(p)$ is the inverse depth of point p. $\lambda*$ is all the possible choices of the the inverse depth of p. It minimizes the reprojection photometric error though the epipolar line. The details of solving this equation is as mentioned in [9] and [4]

$$\lambda(p) = argminC(p, \lambda*)$$
$$C(p, \lambda*) = ||I_{ref}(p) - I(\omega(p_i, \lambda * (p_i), \xi))||_1 \quad (3)$$

The results of the current frame will be used to refine the inverse depth of the keyframe. The inverse depth map of the key frame will continue to be refined until a new keyframe in chosen. When the current frame is far from the keyframe, it will replace the old one and become the new keyframe. The distance between a frame and a keyframe is calculated based on the weighted combination of the translation and rotation matrix, as shown in equation 4, where ξ is the camera matrix and W is the weight matrix. A threshold is introduced to decide whether to generate a new keyframe or not. After the new keyframe is generated, the depth map will be initialized by transferring the previous keyframe's inverse depth map.

$$dist(\xi_{ji}) = \xi_{ji}^T W \xi_{ji} \quad (4)$$

The pose estimation and the inverse depth estimation are calculated in two different threads, which is illustrated in figure 3. Once a new frame is captured by the camera, the pose tracking thread will keep updating the pose using the current keyframe. The inverse depth estimation thread will decide whether to generate a new keyframe or to further refine the inverse depth of the keyframe.

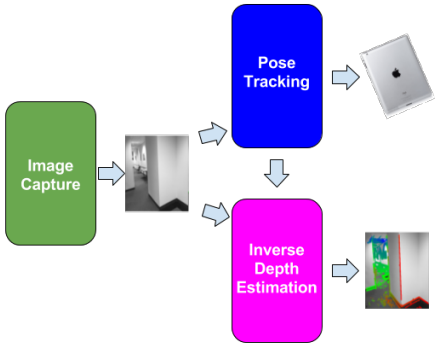


Figure 3: LSD SLAM Thread Diagram

2. Background

Our method is based on LSD-SLAM, which has previously been successfully ported to Android [11]. This implementation was performed on a older Android phone and was able to achieve 30 fps using NEON intrinsics. Unfortunately this code is not publicly available, however the desktop version is available at `https://github.com/tum-vision/lsd_slam` [3].

The LSD SLAM desktop version has the full SLAM pipeline and uses the Robot Operating System (ROS) for its input and output. ROS is a set of libraries which help to enable robot system development and was previously used for input and output to the system[5]. All input and output was replaced by OpenCV, the C++ standard library and built-in functionality in Objective-C [2]. This was a considerable effort due to the fact that the input and output was so closely tied to ROS and OpenCV lacks many of the rendering capabilities that ROS possesses, such as being able to render 3D graphs of the camera pose.

In addition to OpenCV, several other libraries were used to compute the inverse depth map. The Eigen library was used for efficient linear algebra computation [6]. This is a robust library, which is optimized for use with the xCode LLVM/Clang compiler. In addition to Eigen, Sophus was used in the implementation of the Lie Group algebra [12]. Sophus is built on top of Eigen and thereforefore it also allows for good optimization by the xCode compiler.

The LSD SLAM algorithm lends itself well to a multi-threaded approach as previously discussed. To allow for efficient multi-threading, the Boost library was used, which is an efficient and widely used C++ library [1]. Boost was able to be used in iOS by compiling the library using a script that was developed in the Open Source community [10]. After compilation, the generated framework and header files could be imported into the xCode project and be used once the correct project settings were applied.

3. Approach & Challenges

3.1. ROS and Pose Graph Removal

This project was started by reviewing the LSD SLAM papers and desktop code base [11] [3]. After reviewing the code, the large extent to which the code was dependent on ROS was apparent. ROS handled all input and output, as well as how to build the code, and software configuration, such as passing optional camera correction models to the application. To address this, the entirety of the input and output for the application was replaced by OpenCV. In addition, ROS Catkin, which was used to build the program, was replaced by CMake and the optional camera arguments passed to the application were removed completely. This was a considerable effort to remove, due to the dependencies on ROS throughout the code base. Once ROS was removed, the two versions of LSD-SLAM were compared side-by-side using a previously recorded video clip. The results of the two code bases were very similar to each other.

Once ROS was removed, the portions of the code base which were unnecessary for Visual Odometry were removed. Specifically, the pose graph optimization portion was removed. After this step, the code was reviewed once more to ensure that any classes that were orphaned by the ROS or the pose graph optimization removal were no longer present. Any unneeded libraries were also removed.

3.2. Boost and xCode Integration

In the next phase of development, toy examples of Boost code, such as creating Boost threads and mutexes, were created in xCode in an empty Single View Application. Using this toy example, the Boost library was attempted to be integrated into this project. This was not a straightforward or simple process, since it was unlike any other library that had been used previously in xCode. After several unsuccessful attempts, the open source project, Open Frameworks, was found and it allowed for the Boost library to be built in such a way in which it could be used in xCode [10]. This open source project is highly recommended and is easy to use once the correct xCode project settings are found.

Once the Boost toy example was able to be successfully compiled and run, a new Single View Application $MobileLSD$ was created. The LSD code, which had the pose graph estimation and ROS dependencies removed, was added to this project, as well as all required libraries. A simple UI consisting of only the current frame rate, as well as the inverse depth map, was created and additional frameworks needed for video acquisition were added. Additionally, the iPad camera was calibrated using an open source app and the camera intrinsics were updated using this improved intrinsic matrix [7]. At this point, the application was extremely slow and only achieved a frame rate at about .6 fps, however it was verfied that the output was correct.

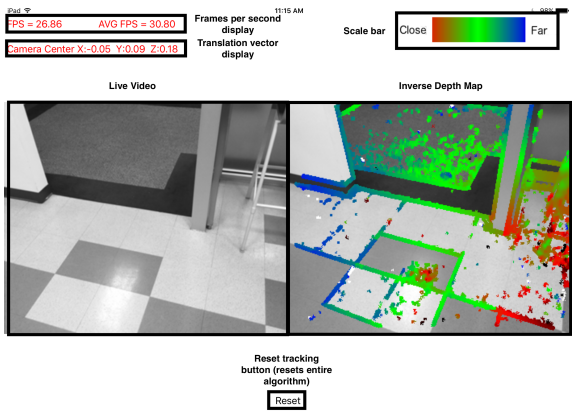


Figure 4: Overview of Mobile LSD's User Interface

3.3. Refinements

To address the slow frame rate of the application, the app was profiled using the Instruments profiler. In doing so, the profiler builds the application in release mode, which by default contains optimizations. This release version achieved an average frame rate of 25 - 30 fps. This prompted further experimentation with optimization flags, which can be found in section 4.2.

At this point in development, Mobile LSD nondeterministically crashed and was unable to recover if the correspondences between the keyframe and the current frame were too low. This required the app to be restarted frequently, which was cumbersome and a nuisance. Debug statements and breakpoints were put in place to identify the cause of these errors. It was determined that the root cause of the crashes was due to poor work synchronization between the threads. This issue was resolved and algorithm reinitialization code was added to the system to allow for a more stable app.

Simultaneously as the bug fixes and robustness improvements were ongoing, a new, more informative user interface was created that allows for better testing and nicer user experience, as shown in figure 4. An icon was added as well to give the app a more complete and polished look.

4. Results

The final user interface features the live video and the inverse depth map side-by-side as well as the translation vector (in meters), frame rate information, and the color scale, as shown in 4.

When the app is launched or if the percentage of correspondences between the current frame and the keyframe is too low, the application will automatically reinitialize and the inverse depth map will initially be all green as shown in figure 5. Once sufficient movement occurs, the map will

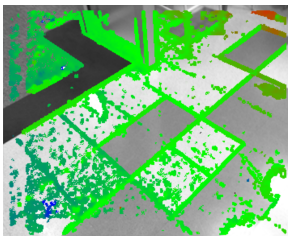


Figure 5: Example of inverse depth map during initialization

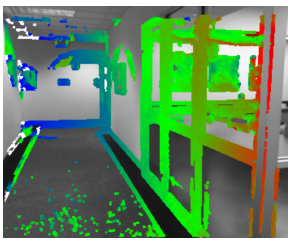
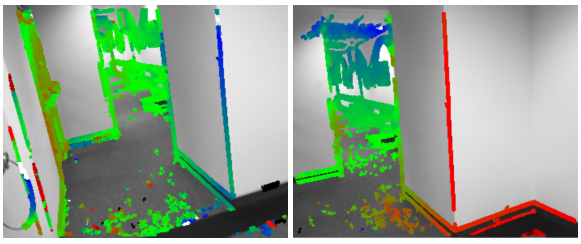


Figure 6: Example of successful inverse depth map generation

self correct and produce an appropriate inverse depth map as shown in 6.

4.1. Robustness

Mobile LSD is robust against propagating erroneous inverse depth maps and is able to self-correct after an incorrect map is generated as shown in figure 7. Typically erroneous estimates occur in situations where there is little translation that is perpendicular to the scene. Without translation, the parallax effect is not achieved and a correct inverse depth map cannot be created. However, once the user moves an appropriate distance, the system is able to calculate and render a correct inverse depth map quickly due to the frequent updates to the keyframe's inverse depth map.



(a) Erroneous inverse depth map (b) Corrected inverse depth map

Figure 7: Example of system recovery after an erroneous inverse depth map generation. Image occurred less than 3 seconds after image a.

4.2. Performance

One of the goals of this project is to create an app with a high frame rate. As shown in [11], as the frame rate increases, error contributed by the rolling shutter of the camera is minimized. Once frame rates are high enough, the effects of the rolling shutter can be ignored. Furthermore, a high frame rate allows for more updates to the inverse depth map for a given key frame, which allows for further refinement of the inverse depth map and greater accuracy.

In an effort to achieve a high frame rate, optimization flags offered by the compiler were experimented upon. The xCode LLVM/Clang compiler offers five levels of optimization, ranging from no optimization to aggressive optimization. One of the levels of optimization attempts to not only to optimize the performance, but to optimize the size of the app itself. The average frames per second on an iPad 2 over 15 seconds with each of the five possible levels of optimization can be seen in table 1.

Optimization Level	Flag	FPS
Fastest, aggressive optimizations	-Ofast	31.20
Fastest, smallest	-Os	27.32
Fastest	-O3	30.62
Faster	-O2	30.61
Fast	-O,-O1	2.23
None	-O0	0.63

Table 1: Optimization level vs. average frames per second over 15 seconds in a controlled setting on an iPad 2

In table 1 it can be seen that highest average frame rate of 31.20 is achieved by using the $-Ofast$ compiler flag. It is unsurprising that the next highest level of optimization, $-Os$, achieves a slightly lower average frame rate of 27.32 fps, however it was surprising that this frame rate was lower than the next two levels of optimization, $-O3$ and $-O2$. An explanation for this behavior was found on the Apple Developer site, where it states that the $-Os$ level of optimization performs only all the optimization which do not increase the size of the app itself [8]. It therefore stands to reason that optimization levels $-O3$ and $-O2$ result in larger app sizes, but do contain more optimized code. Finally, the lowest two levels of optimization achieve abysmal frame rates of 2.23 and 0.63 fps respectively. This is due to the fact that optimization level $-O1$ only contains simple optimizations and that level $-O0$ contains no optimizations at all.

5. List of Work

Equal work was performed by both project members.

6. GitHub Page

Full instructions on how to build MobileSLAM can be found at our GitHub page at https://github.com/xorthat/MobileSLAM. Since the project checkpoint, we have updated our Github main page with how to build the project from scratch and added have over additional 35 commits to this project.

References

[1] D. A. Beman Dawes et al. Boost c++ libraries. http://www.boost.org, 2001.

[2] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[3] J. Engel, T. Schöps, and D. Cremers. LSD-SLAM: Large-scale direct monocular SLAM. In *European Conference on Computer Vision (ECCV)*, September 2014.

[4] J. Engel, J. Sturm, and D. Cremers. Semi-dense visual odometry for a monocular camera. In *IEEE International Conference on Computer Vision (ICCV)*, Sydney, Australia, December 2013.

[5] O. S. R. Foundation. Robot Operating System. www.ros.org, 2016. [Online; accessed 08-Dec-2016].

[6] G. Guennebaud, B. Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[7] T. Horikawa. Camera calibration ios. https://github.com/thorikawa/camera-calibration-ios, 2016.

[8] A. Inc. Tuning for performance and responsiveness. https://developer.apple.com/library/content/documentation/General/Conceptual/MOSXAppProgrammingGuide/Performance/Performance.html, 2015.

[9] R. A. Newcombe, S. J. Lovegrove, and A. J. Davison. Dtam: Dense tracking and mapping in real-time. In *Proceedings of the 2011 International Conference on Computer Vision*, ICCV '11, pages 2320–2327, Washington, DC, USA, 2011. IEEE Computer Society.

[10] D. Rosser. ofxiosboost. https://github.com/danoli3/ofxiOSBoost, 2016.

[11] T. Schöps, J. Engel, and D. Cremers. Semi-dense visual odometry for AR on a smartphone. In *International Symposium on Mixed and Augmented Reality*, September 2014.

[12] H. Strasdat. Sophus. https://github.com/strasdat/Sophus, 2016.