Imperial College
London

FINAL YEAR PROJECT REPORT

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

ElasticFusion on the Google Tango Tablet

Author:
Jiahao Lin (CID: 00837321)
Email: jiahao.lin13@imperial.ac.uk
Supervised by Dr Stefan Leutenegger
Second marker: Professor Andrew Davison

Date: June 19, 2017

1 Abstract

Google Tango[1] is an augmented reality computing platform on Android[1] platform by Google that give mobile devices the ability to sense 3D space information. Equipped with this platform Google's Tango tablet is an Android device that carry sensors including 4MP(Mega Pixel) color camera, Inertial Measurement Unit (IMU), and Infra-Red depth camera. it also carrys NVIDIA Tegra K1 processor which provides more powerful computational power than other mobile devices. Given this cheap commodity mobile hardware and software, this project aims to build a fully self-contained 3D dense surface reconstruction solution using state-of-the-art RGB-D(color and depth) SLAM (Simultaneous Localization And Mapping) algorithms.

In the recently published dense RGB-D SLAM algorithms, ElasticFusion[2] published by Dyson Robotics Lab[2] at Imperial College London is one that achieve state-of-the-art performance by applying local and global loop closures using a deformation graph, so that drifts raised by accumulated errors can be recovered and global consistency of the map is maintained. The fact Elastic Fusion scored high marks on benchmarks for trajectory estimation and surface estimation makes it perfect to be applied on Google Tango Tablet. Also, the current code exist for ElasticFusion running on desktop computer made heavily use of CUDA(Compute Unified Device Architecture) for GPU general data processing, which is supported by the NVIDIA Tegra K1 processer on Google Tango tablet.

This project have attempted to port the code for ElasticFusion onto Google Tango Tablet and integrated with 3D points data from Tango platform, although the result of running ElasticFusion on the tablet has shown to be not effective enough for self-contained real time application due to processing power and sensor problem on the tablet and time and effort limitation in this project, this attempt has still proved that it is possible and more convenient to record the 3D space data on a hand held portable device first, and then let the data to be processed on a more power desktop computer for high quality 3D surface reconstruction.

Also, since the Tango platform on the tablet provides pose estimation feature, this pose is simply feed into the ElasticFusion program run on the tablet, so that the ElasticFusion part doesn't need to waste time and processing power to execute the camera tracking part. However, the pose provided by Tango may not be accurate enough and during the reconstruction process ElasticFusion will alter the pose if global or local loop closure happens, in this project a simply way to correct pose input to ElasticFusion by Tango is developed to over come this problem.

[1] `https://www.android.com/`

[2] `http://www.imperial.ac.uk/dyson-robotics-lab/`

2 Acknowledgements

Firstly I would like to express my deepest appreciation to all those who provided me the possibility to complete this report. A special gratitude I give to my project supervisor, Dr. Stefan Leutenegger, for his patient overseeing and guidance throughout this project, also his time and effort invested to give useful advices.

Secondly I would also like to thank my friends for accepting nothing less than excellence from me. Last but not the least, I would like to thank my parents for supporting me spiritually throughout the progress of this project and my life in general.

Contents

3 Introduction

The main objective of this project is to develop a fully self-contained mobile dense 3D surface reconstruction solution running on Google Tango Tablet using the ElasticFusion algorithm. Ideally the 3D surface reconstructed should be rendered on screen in real time from the tablet's viewpoint when user is holding the tablet to scan the room, and after the scanning stopped the 3D map created should be able saved on device to allow for later use or shown on other devices. The motivation of behind this project is the lacking of high quality 3D surface reconstruction application on mobile device. Applications for 3D reconstruction currently exist on Tango Platform mostly use volumetric representation, and can only produce coarse model of surrounding space, a screen shot of using 3D mesh created using Google Tango's demo mesh builder application is shown in Fig 1. Also the existing applications fail to produce surface map without misalignments during the scanning process, which is often caused by loopy motion of the scanning device in the process. When doing 3D reconstruction, simple fusion of 3D point clouds would produce drifts caused by errors accumulating over time, therefore by applying ElasticFusion algorithm to the solution, even extremely loopy trajectories of the tablet and long duration of scanning process would not result in decrease of accuracy of the map. On the contrary, revisit areas that has previously been mapped will only help to maintain the global consistency of the map. The part of algorithm that made this possible in ElasticFusion is the local and global loop closure checking after camera tracking process, which will be discussed in detail later.

Previously, ElasticFusion can only be run on a desktop computer with powerful

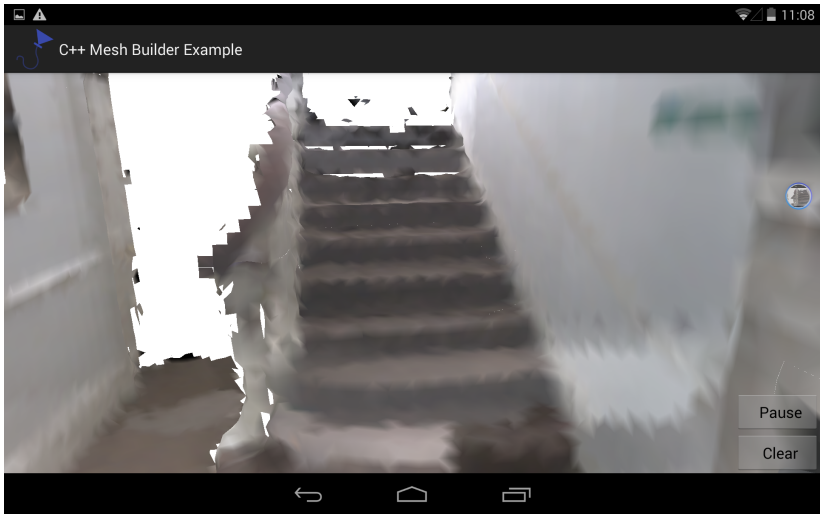

Figure 1: Demo App Mesh Builder running on tablet

graphic card, using a raw dataset that is given, or connect the computer to a RGB-D camera such as Microsoft Kinect or ASUS Xtion Pro Live or Intel RealSense 3D Camera. That makes it non convenient for a user to scan the space he or she is currently in for 3D reconstruction, therefore the Google Tango tablet with both RGB and depth camera would be the perfect mobile device to implement a portable version of Elas-

ticFusion on. The Google Tango tablet carries Android 4.2.2 OS, has 4GB of RAM, a quadcore Nvidia Tegra K1 graphics card supporting CUDA, six-axis gyroscope and accelerometer, a wide-angle 120 field of view tracking camera which refreshes at 60Hz, a projective depth sensor which refreshes at 3Hz, and a 4 megapixel color sensor which refreshes at 30Hz. The processing power on this tablet succeeds most mobile devices on the market right now, which makes running ElasticFusion on the device alone become possible. Since ElasticFusion requires both high end processor and graphic card on a desktop computer, from the read me of ElasticFusion on Github: "A very fast nVidia GPU (3.5 TFLOPS+), and a fast CPU (something like an i7)." is recommended for a desktop computer, therefore one of important problem in this project is that Tango tablet's processing power is still not strong enough to run ElasticFusion for real time application. However, the attempt make in this project have still shown the possibility of running a state of the art SLAM algorithm on mobile device.

In the original design for this project, if the development of this solution is successful, an augmented reality application could be created as demo to illustrate the interaction of virtual object with scanned 3D surface map. But after various experiments in this project, a conclusion that ElasticFusion could not be run on Tango tablet as a real time application drawn, due to processing power and depth sensor accuracy problem, although due to time and effort limitations, the ElasticFusion on Tango application produces in this project still produce incorrect reconstruction model and the program has not been optimized. Despite all the above, the contribution of this project still includes the attempt made to implement this solution, and the possibility of using Tango tablet as a pure scanning device for recording data, and let the data to be processed on other computers for doing 3D surface reconstruction.

In the solution, an Android application is developed using Android Native Development Kit (NDK)[3], which allows the use of Tango C API[4] to acquire depth camera data and 3D pose data, also makes it possible to reuse the existing code for ElasticFusion that is written in C++. Therefore most of the application is written in C++, with a few Java code for the Android application. The application uses the code for Advanced Estimation in Robotics (433H) course practical exercise by Dr Stefan Leutenegger in Department of Computing at Imperial College London as the basic frame work, firstly the code is combined with the logger for ElasticFusion[5] to write the RGB, depth and pose data in format required by ElasticFusion acquired from Tango into two files on the Tango tablet, which allows ElasticFusion to be run on desktop using these two files as raw dataset. Then the core function part of code for ElasticFusion is ported and adapted to Android with its dependencies changed or removed, so that ElasticFusion can be run on Android while its source remains in C++. One thing worth notice in here is that this project uses an internal version of ElasticFusion source code at Dyson Robotics Lab in Imperial College London, instead

[3] https://developer.android.com/ndk/index.html
[4] https://developers.google.com/tango/apis/c/
[5] https://github.com/mp3guy/Logger1

of the open source version on Github[6]. The final produced application is able to log the data from sensors on Tango into files, or run ElasticFusion from sensor data in real time, or run ElasticFusion from raw dataset files on the device with or without pose data. This provides various way to do experiments or measure efficiency during the project. However, due to time and effort limitations, no interface for showing the progress of ElasticFusion running is built, this is different from original planning but the reasons will be explain in detail later. Right now on the screen of the application, interface only shows the image that color camera is taking and a few buttons, screen shot and description will be provided in later section.

Finally, simple change is made on both desktop version and Android version of ElasticFusion to allow for pose feed into the program to be corrected overtime when global or local loop closure happens. This help maintain consistency between the pose feed into ElasticFusion and the internal pose stored in the program. Since the original ElasticFusion can only estimate pose from RGB-D data, or use an input pose as ground truth pose, or use the input pose as a bootstrap pose for optimization, the adaptation made in this project enables the pose feed into ElasticFusion to be consistent with the true pose while also not waste time to do the camera tracking process when the processing power is not so strong like on the Tango tablet.

[6] `https://github.com/mp3guy/ElasticFusion`

4 Background

This section includes two parts, the first part includes some paper related to SLAM and SLAM algorithm including ElasticFusion, the second part includes some SLAM applications developed to be run on mobile devices.

4.1 SLAM papers

4.1.1 Past, Present, and Future of Simultaneous Localization And Mapping: Towards the Robust Perception Age

This survey paper: past present and future of SLAM [3] gave a detail description about the current development state of modern SLAM, including standard formulation and models, different methods used in SLAM. Below we give a brief summary of basic knowledge about SLAM from this paper.

Simultaneous Localization and Mapping (SLAM) means constructing the model of environment (map) that the robot is in, and estimating the robot state within the environment at the same time. Usually the robot is equipped with some kind of sensor: RGB camera, depth sensor, IMU, or GPS, So that the robot is able to perceive the environment in some way from these sensor. In standard models, the robot state includes its position and orientation, velocity, sensor biases and calibration parameter. Using the robot status and data read from sensors, the environment(map) constructed could be representation in different forms. With the existence of the map, errors raised in estimating robot state could be eliminated and also doing ”loop closure” when robot revisits a place in the map allows drift to be corrected and hence improve accuracy of the estimated status of robot.

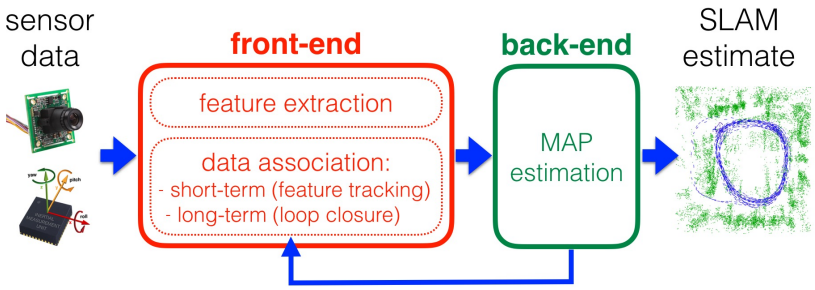

Figure 2: Front-end and back-end in a typical SLAM system[3]

The structure of SLAM system is mainly divided into two parts: front-end and back-end The front-end extract features from sensor data and associate them to the model used to make predictions, then the back-end does estimation of robot state and the map using this model produced by front-end. As you can see in the figure 2, this process forms a continuous process of tracking and update.

Classical formulations of SlAM is to take probabilistic approaches such as models based on Maximum Likelihood Estimation, Maximum-a-posteriori (MAP) estimation, EKF(Extended Kalman Filters), and particle filters.
Many of the popular SLAM algorithm models it as a maximum-a-posteriori estimation problem by taking probability approach. And usually using a factor graph to show relationships and constraints between variables. In these models, often an observation or measurement model is used to represent the probability of observe measurements given the state of robot, given some random measurement noise that usually model to be under a zero-mean Gaussian distribution. By using Bayes theorem, the MAP estimate, in which the posterior is the probability of robot state given a measurement, can be calculated given some prior belief of the robot state. In the case that there is no prior information known about robot state, the MAP estimate would simply become a maximum-likelihood-estimate. When we are given a set of independent robot states and measurements, these variable would become factors(nodes) in the factor graph, and their probabilistic relationships would become the constraints(edges) in the factor graph.

Factor graph allows us to visualize our problem in a simple way and provides an insight into how the variables are related together by their constraints. 3 from the paper have shown an example of it for a simple SLAM problem.

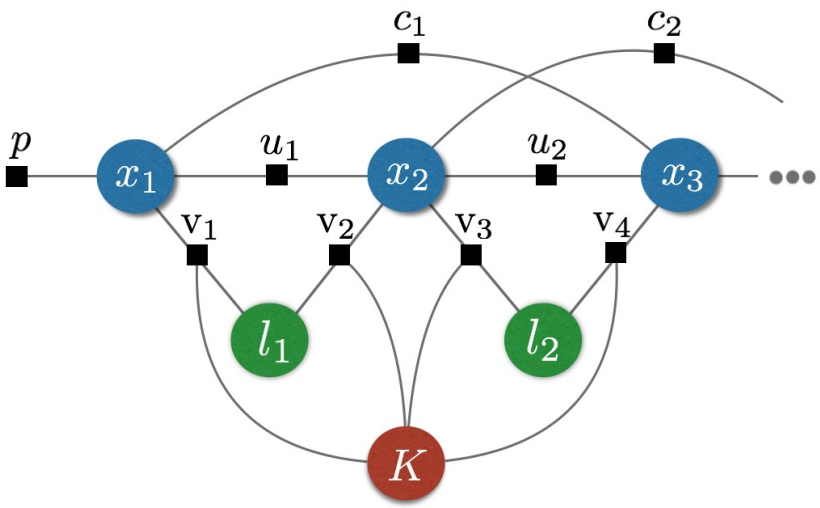

Figure 3: SLAM as a factor graph[3]

As explained in [3]: "Blue circles denote robot poses at consecutive time steps ($x1$, $x2$ ...), green circles denote landmark positions ($l1$, $l2$ ...), red circle denotes the variable associated with the intrinsic calibration parameters (K). Factors are shows are black dots: the label "u"marks factors corresponding to odometry constraints, "v "marks factors corresponding to camera observations, "c"denotes loop closures, and "p"denotes prior factors."

When calculating the MAP estimate of robot state, we can transform the problem

into minimizing the negative log of posterior, assuming the noise is zero-mean Gaussian distributed, this becomes a non-linear least squares problem, since we will be minimising the measurement error's $l2$ norm. This method is similar to the BA (Bundle Adjustment) method in computer vision area, which minimise the reprojection error. However factor graph can not only contain visual and geometry factor, but also a variety of data from different sensors. Also the MAP estimate is calculated every time a measurement is arrived, instead of performing calculation after all the data is given.

Different from factor graph optimization, a similar approach is called pose graph optimization, which estimate trajectory of robot using measurements between relative poses as constraints, and optimized these poses to achieve better accuracy trajectory.

Other than optimization-based estimation, some filtering approach using EKF model have also achieve great results, such as MSCKF(Multi-State Constraint Kalman Filter of Mourikis and Roumeliotis)[4], EKF is used for performing filtering on the linearised point of estimate, and have shown to be accurate in visual-inertial navigation.

In terms of the representation of the map, one way is to have a group of features or landmarks in 3D, these features are usually extracted from sensor images to be allow easy distinguishable from environment, such as edges and corners. The method used to extracting features from images are well-established within computer vision area, using descriptors like ROB, SIFT, and SURF. The discriminative and repetitive properties of landmarks allow the correspondence of each landmark and sensor measurement(image), and then via triangulating, the robot is able to compute the relative pose between measurements and geometric information about landmarks, therefore achieve localization and mapping.

Other than feature based approach, sometimes raw dense representation of the environment is used. different than extracting landmarks, this approach stores a dense 3D map using high resolution points, the 3D geometry of environment is described by these vast amount of points, so called point clouds. With stereo or depth cameras, lase r depth sensors, 3D information of the points can be easily obtained, therefore this method is popular in RGB-D SLAM. Sometimes, these points doesn't only stores simple 3D information, in Elasticfusion[2], each 3D point is represented by a surfel, which is a disk with certain radius centred by the 3D point and stored information such as normal vector of that point on the surface and color. These suefel disk combine together can encode the geometry of environment better than simple points.
At more a higher level than raw, unstructured points, dense related element that partitioned by space is also used to represent surface and boundaries, such as surface mesh models(connected polygons formed by points) and implicit surface representations like octree and occupancy grid. KinectFusion[5] storing surfaces using volumetric representation by uniformly subdivide a 3D physical space into a 3D grid of fixed size voxels, voxels defining surface has a zeros value of the truncated signed-

distance function (TSDF) function that stores distance information relative to the surface. We can see that represent the environment in these types of dense data requires huge amount of storage, however they give very low level geometry information which is suitable for 3D reconstruction and rendering.

Comparing feature-based and dense SLAM methods, for the purpose of this project, dense SLAM algorithm Elasticfusion[2] is used as 3D reconstruction is the main objective instead of state estimation, since Elasticfusion maintains a dense 3D surfel map that make use of all the information about the environment obtained from sensors it is more robust in localization and mapping that feature-based approaches.

4.1.2 Kinectfusion

KinectFusion[5] is a dense RGB-D SLAM method that enables high quality 3D reconstruction and interaction in real time. This method make use of low cost mobile RGB-D camera, scanning the environment in real time and acquire 3D point information about the space. During the reconstruction process, holes caused by absence of depth measurement in original 3D point cloud is filled in, and the model is refined by repetitive scanning over time.

Firstly the 3D points reading are then stored in a 3D vertex map and their normal vectors are calculated by reprojection of neighbouring vertexes. After that the camera pose is tracked by using Iterative Closest Point (ICP)[6][7] algorithm to align current 3D points with the previous frame. In ICP algorithm, first the corresponding point of each current 3D point is found by doing projective data association, which reprojects each previous point into image coordinates and use this to look up current vertexes that may be correspondence points. In the second part these correspondences are tested by calculated distance and angle between them and reject any outliers that are beyond certain threshold. Please be noted that the running of ICP in real time is because the use of GPU to parallel processing a current vertex per thread, according to the original paper.

The structure of KinecFusion is given in Fig 4 from original paper. The result of ICP is a transformation matrix from previous camera pose to the current pose, which is later used for converting the coordinates of current vertexes into the global coordinate frame. These coordinate data is then fused into the current model that uses a volumetric representation, as mentioned before. The 3D space is uniform divided into fixed size voxel grid. The geometry of surface is implicitly stores in these voxels by having zero values of Truncated Signed Distance Functions (TSDFs) at the surface, positive value in the front of surface, and negative at the back of it. Also, to improve efficiency and minimise storage usage only a truncated area around the surface is actually stored, thus the name ”truncated”.

When integrating the current vertexes into voxel grid, each voxel is perspective projected back onto the image to calculate the difference between it and the mea-

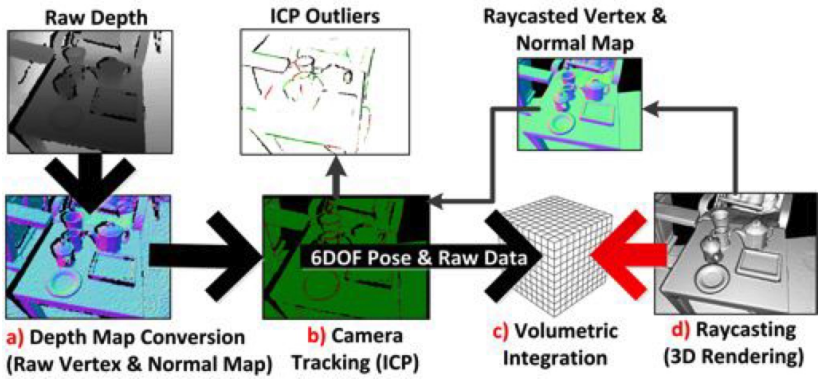

Figure 4: Overview of tracking and reconstruction pipeline from raw depth map to rendered view of 3D scenem[5]

surement depth of this coordinate. The distance difference is then normalized to maximum TSDF distance and updated using weighted average with previous value. Similar to the implementation of tracking stage, the integration and update process also make use of GPU parallel processing, each thread is assigned a slice of of grid along the $Z-axis$, and sweep through each voxel by its (x, y) position for processing, as shown in Fig 5.

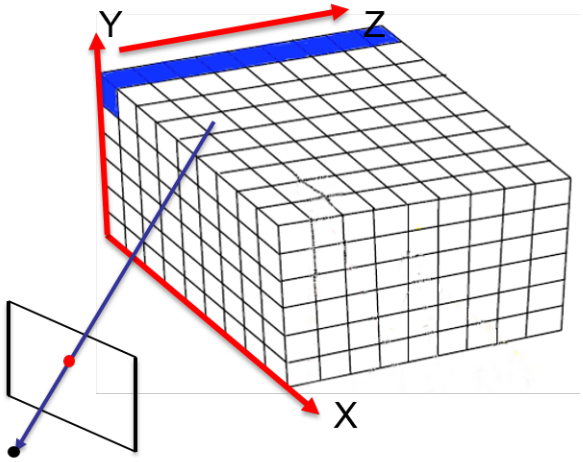

Figure 5: Volumetric Integration using GPU thread [8]

Finally, in order to render the 3D reconstructed surface raycasting is performed. This is done by having each GPU thread to follow a ray from the camera center along one pixel in the image into the voxel grid in camera coordinate space, the ray is extended until a zero-crossing voxel is met, this voxel represents the surface point observed by this pixel and the position is extracted by doing a trilinear interpolation on this voxel point. And the normal at this position is calculated by find the surface gradent of TSDF values around the position. With these information for each pixel, a live 3D reconstruction view from camera view point can be rendered using camera's pose and intrinsic information.

4.1.3 Keyframe-Based Visual-Inertial SLAM Using Nonlinear Optimization

Other than using depth sensor with RGB camera, another kind of sensor equipped on the Google Tango tablet is Inertial Measurement Unit (IMU). Okvis[9] is a sparse feature-based SLAM approach that tightly coupled IMU measurement with stereo visual measurement using nonlinear optimization.

Usually in optimization-based visual SLAM, the structure of geometry is to relate camera poses using landmarks, and optimization is performed to minimize the error of landmark reprojection in different observing frames. However, in visual inertial SLAM, the IMU measurement is used to add additional constraint between camera poses and speed and estimation biases from gyroscopes and accelerometers. The structure graph from original paper is shown in Fig 4 with detail labelling of different variables and constraints.

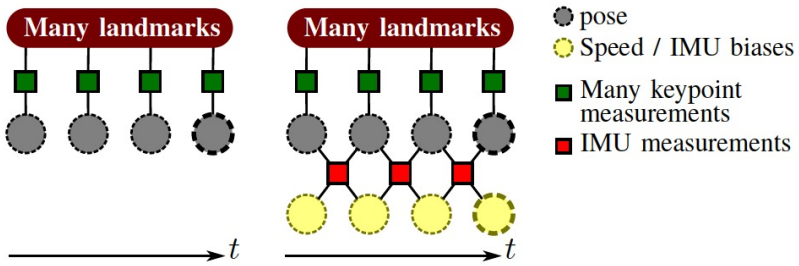

Figure 6: graph of variables and their relationships in visual SLAM on the left and visual intertial SLAM on the right [9]

In order to tightly couple the visual and inertial measurements, a joint nonlinear cost function that contains both visual reprojection error term and IMU error term is created. These both errors are eliminated by optimizing this one error function.

To reduce complexity and ensuring real time operation, a fixed size temporal window of robots poses and relative states is maintained, and old robot poses will be marginalized out as new frames are inserted.
In the visual frontend, a sparse map of frames and landmarks is stored. For each new frame, keypoints are extracted using SSE(Streaming SIMD Extensions)-optimized Harris corner detector and BRISK(Binary Robust Invariant Scalable Keypoints) descriptor, their 3D positions are then triangulated by stereo and put into the sparse map. During the process landmarks and keypoints are brute-forcely matched and outliers are rejected by using predicted pose from IMU data. In the optimization window, two kind of frames is maintained, first is a temporal window of the S most recent frames, second is N keyframes in the past. A frame is determined as keyframe if the number of matched points with previous frame is less than 50 to 60% of the number of detected keypoints.

In the marginalization process, initially the first $N + 1$ frames forms the marginal-

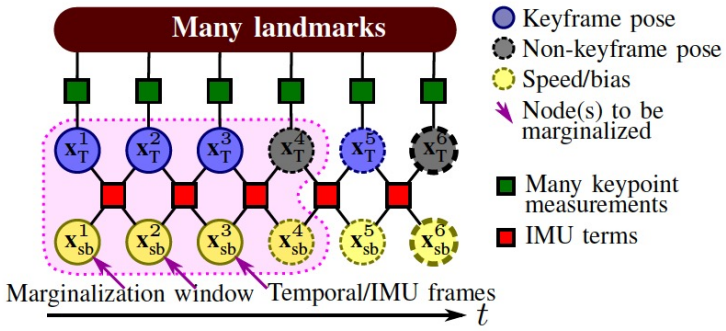

Figure 7: Graph of initial marginalization window [9]

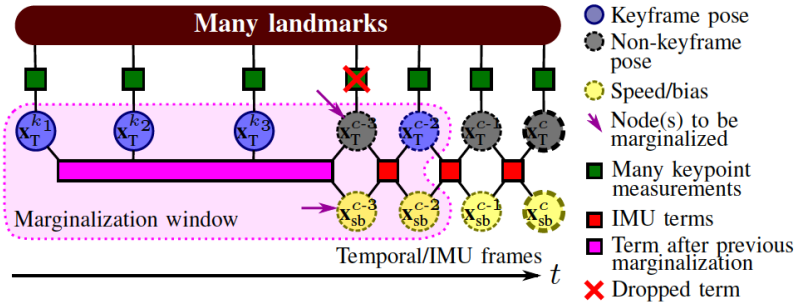

Figure 8: Graph for marginalization in the first case [9]

ization window, as shown in Fig 7. When a new frame is inserted into the marginalization window, there are two ways to marginalize old state. If the oldest frame in temporal window is not a keyframe, then the state of that frame with its measurements, speed and biases will be dropped, as shown in Fig 8. On the other hand if it is a keyframe, landmarks that are visible in first temporal frame but in most recent keyframe is also dropped, as shown in Fig 9.

4.1.4 Elasticfusion

Elasticfusion[2] is a RGB-D dense SLAM method for 3D reconstruction in room scale without using pose graph it also use surfels to represent the map instead of voxel grid such as in KinectFusion[5]. In the scanning process, it iteratively check for local and global loop closures and refine the map by using a deformation graph to apply non rigid deformation to the surface. However, while this approach has achieved great results in room scale, it may be difficult to be used on bigger or outdoor scenes due to complex light condition and size of memory needed to store the surfels.

In the surfel map, each surfel stores the information about a local surface area around the center of the surfel, the surfels are lay out in a wat that holes on the surface are minimised. The information about a surfel include its position, normal vector, color, weight and radius. The radius is larger as distance between image

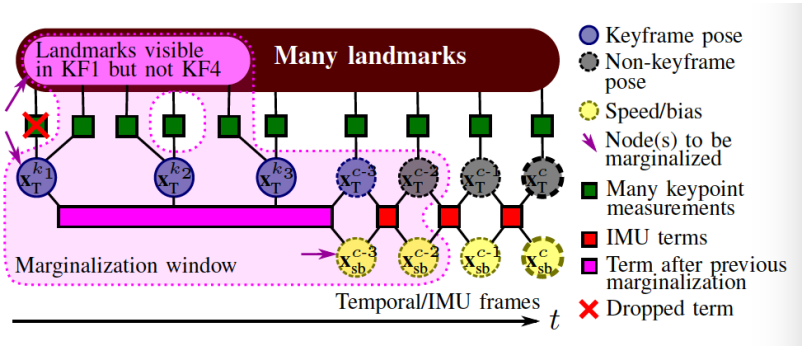

Figure 9: Graph for marginalization in the second case [9]

center and surfel is greater. The surfels are extracted from point cloud using the similar method as in [10], illustrated in Fig 10, but differently, in here the surfels are divided into active and inactive parts. Only active surfels are used for camera pose estimation and depth map fusion, while the inactive parts are the surfels that have not been observed for a given time period and can be reactivated in a local loop closure. In ElasticFusion, OpenGL shading language is used to manage the surfel map and update, fuse and render the view.

The detail structure of ElasticFusion is shown Fig 11 in the original paper. From

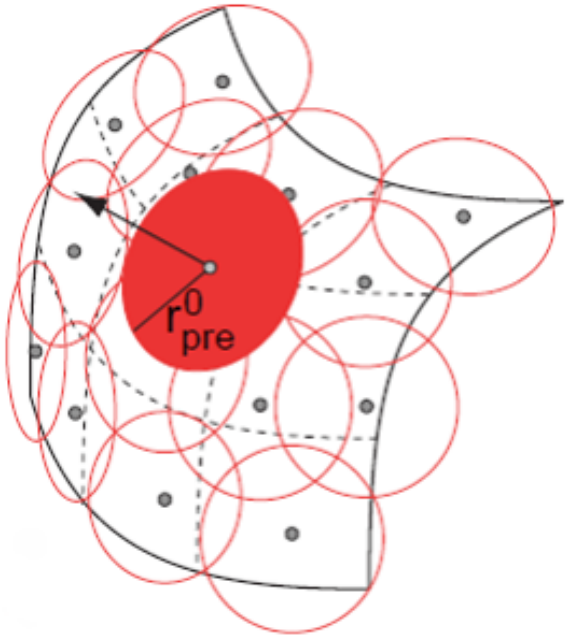

Figure 10: Illustration of sample graph nodes from surfels

the graph can be seen that firstly in the camera pose estimation step, dense frame-to-model camera tracking is used, geometric and photometric pose estimations are combined into a joint cost function to for minimization. Geometric error is calculated using frame to model projective data association and ICP algorithm as in kinecFusion[5], and photometric error is gain by calculating the intensity difference

between current RGB image and backprojected image from the model. Note that here in the tracking process only active surfels are used.

After camera pose tracking for each frame, an active predicted model is produced,

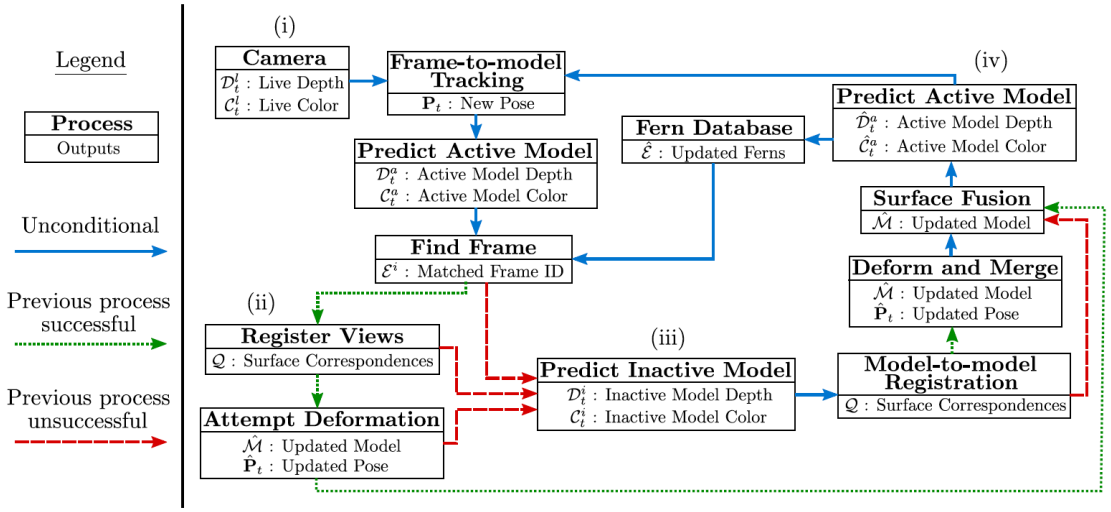

Figure 11: 3D reconstructed model using Elasticfusion

this model is then rendered into an image for global loop cloure checking, to try recover from drifts produced by accumulating error, and realign the camera pose. To enable global loop closure checking, key frame information from past is stored in a randomized fern encoding database using same technique as in [11], it allows storing and matching frames without putting redundant information into the database. If a high quality match between the current predicted view of active model and a frame in fern database is found, points are sampled from matched frame and active view to construct surface constraints that are used to register the surface together and optimize the deformation graph. Each surface constraint contains a pair of points and timestamps to represent the position of point and the destination position that it should reach the destination after deformation.

If no global loop closure is found then local model to model loop closure is checked by try to register the currently predicted inactive model with active model under the current frame, similar method is used when high quality alignment is found, constraints are extracted and fed into the deformation graph, then the inactive surfels in this alignment is reactivated to allow for fusion into the map.

The realignment of map is done by applying non rigid deformation using a deformation graph. Deformation graph is constructed each frame for efficiency, with nodes sampled uniformly from the surfels using similar method as in DynamicFusion[12], this is illustrated in Fig 12 to shown the relation between surfel and deformation graph noded. Each node have a rotation matrix and position vector that represents optimization parameter and neighbourhood of each node forms the edge of the graph. Each surfel has a set of influencing nodes that affects the deformation of

that surfel.

The optimization of deformation of the model is done by minimising five cost func-

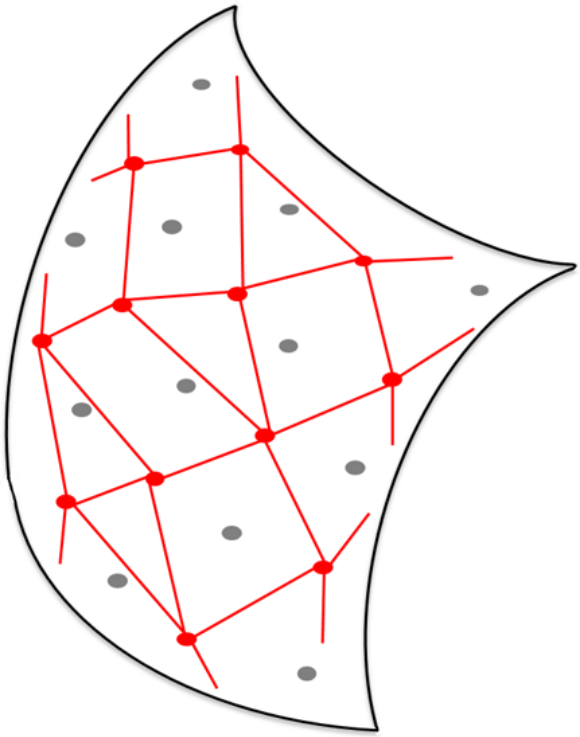

Figure 12: Illustration of surfels

tion in total using the surface constraints constructed in the steps described above. The parameters in surface constraints after optimization reflects the surface registration happened in loop closure. These five cost functions include the following functions: maximises the rigidity of deformation, guarantees the smoothness of deformation in the graph between neighbour nodes, minimises the error in position constraints, that is reduce the difference between source and destination position, also ”pins” the inactive surface of the model without moving to make sure is the active surface that deforms into the inactive surface. In the global loop closure optimization process an extra cost function is added, it introduce the set of relative constraints to prevent the previous surface registrations from being pulled apart by subsequent global loop closures. The optimization problem is solved using iterative Gauss-Newton algorithm on CPU and the result gives a relative transformation matrix that bring the surfaces registered into alignment. The graph Fig 13 from the original paper have shown the process of a scanning with loop closure happens when the scanning camera revisits area that has been scanned before, with active model coloured by surface normal overlaid on the inactive model in greyscale.

After the optimization, deformation graph is applied to the surface by using a weighted distance method to calculate the transformation of each surfel by its influencing nodes. Model is then deformed using this transformation to reflect the registration

of surfaces in loop closure. Finally, the current camera data is fused into the latest updated model to update the surfel map under current viewpoint.

Before new frame of data come in, the current active model is rendered to an image using the latest predicted pose to prepare for tracking, and also being decided whether this frame will added to the fern database.

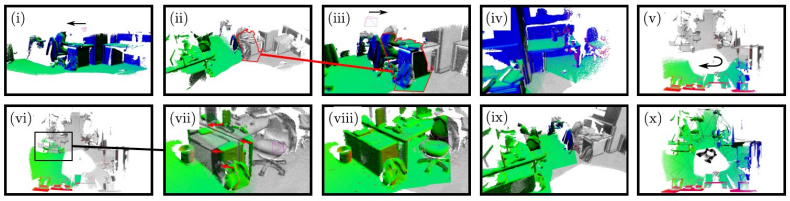

Figure 13: Illustration of loop closure process in ElasticFusion

4.2 Related Work

This section gives some SLAM systems that have been implemented on mobile devices, such as Android phone or tablet, and Apple iPhone or iPad. However, most of these systems mainly focus on the state estimation part of SLAM rather then the mapping part, which means they are not able to be used for 3D surface reconstruction purpose.

4.2.1 CHISEL

CHISEL[13] is a large scale dense 3D reconstruction solution implemented on Google Tango Devices. The graph from the original paper that shows CHISEL application running on the Googlt Tango phone can be seen at Fig 14. It uses dynamic spatially-hashed truncated signed distance field[14] to represent the map and useds the visual-inertial odometry provided by Tango as frontend for localization. To save memory and computation power, space that does not contain surfaces is culled out. By using space carving, even though Tango's depth sensor provides data with high noise, reconstruction with an accuracy of 2-3 cm is still achieved in large scenes, the Fig **??** show a picture of global map reconstructed by CHISEL form the original paper, from this picture can be seen that CHISEL is able to scan and reconstruct the area of a flat with relatively high quality detail. To generalise the solution onto other mobile devices that does not have a powerful GPU,in CHISEL no parallel processing is used to utilise the powerful general purpose GPU provided on Google Tango tablet.

In the pose estimation stage, CHISEL uses the pose estimated provided by Tango platform as a black box, by try to register each depth scan with the model, poses are optimised in the same way as using ICP algorithm, this allows small drifts between frames to be recovered, however during long process of scanning large scene, drifts will still build up overtime due to the lack of loop closure process.

Since the map is stored using spatially-hashed data structure, insert and look up is very fast. The map is divided into chunks, and each chunk is a fixed grid of voxels. These chunks are then spatially-hashed into a spatial 3D hash map. When looking for chunks that need to be updated or drawn, a camera frustum is created for culling chunks that do not intersect with the frustum or not have a depth data, the rest of chunks left is then processed for update or drawn.

For fusing depth scan data into the model, projection mapping is used to compare the depth value on image with the projected visual hull of a voxel, the result is then used to update the TSDF value and weight of that voxel.

In the rendering stage, incremental Marching Cubes is used to generate triangle meshes for the chunk when it is been updated by a depth scan, triangles are generated at the zero isosurface of the TSDF volxel grid.
CHISEL has shown a nice memory-efficient approach to scanning large scene for high quality 3D reconstruction, however it fails to maintain global consistency of the global map, therefore drifts will increase over time due to lacking detection of global loop closure. Despite that, CHISEL provides a great example for utilising the Tango platform as a dense SLAM frontend for 3D reconstruction purpose.

Figure 14: CHISEL screenshot

4.2.2 InfiniTAM

InfiniTAM[15] is an open source cross-platform real time large scale RGB-D SLAM framework publish by University of Oxford[7]. It provides both dense fixed size 3D volume like the one used in KinectFusion[5] and sparse volume using voxel block hashing method[14]. The design of the framework also enables other representations, such as octrees, to be added easily.

[7] https://github.com/victorprad/InfiniTAM

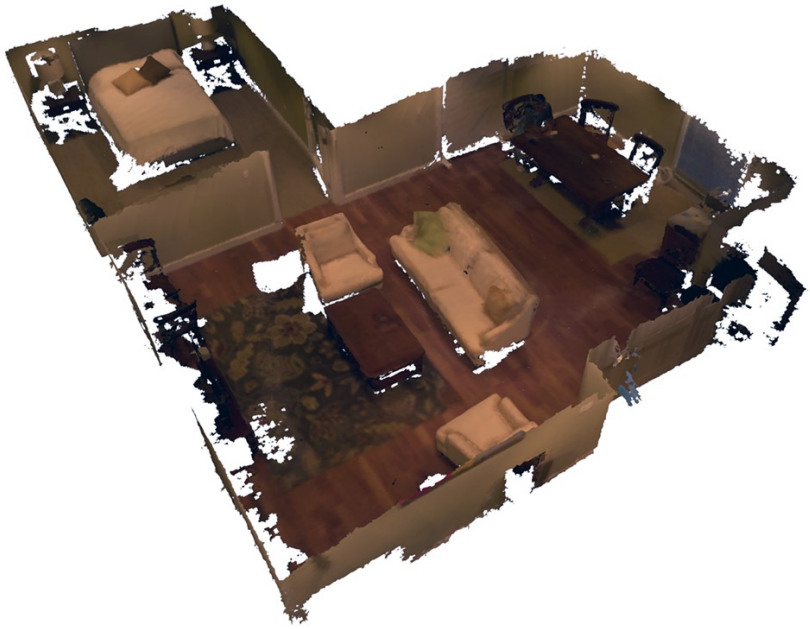

Figure 15: 3D reconstructed model by CHISEL

InfiniTAM is optimised to run at high frame rate in multiple platform, with NVIDIA Titan X graphics card on desktop computer, it can run at over 1000 fps(frame per second), on iOS platform it utilise Apple Metal Graphics API to reach over 25 fps of frame rate. Also on Android devices with NVIDIA K1 processor such as NVIDIA Shield Tablet and Google Tango Tablet, by using NVIDIA CUDA to for parallel data processing on GPU the frame rate is able to reach over 40 fps.

The architecture designed for InfiniTAM makes it easy to extend functionalities and add new features. Also the C++ source code for InfiniTAM is provided on Github for both pure CPU and GPU implementations. The Android version of InfiniTAM is similar to ElasticFusion since it also use both OpenGL and CUDA in the program, it has provided an example of dense 3D reconstruction solution on mobile device using volumetric space representation and CUDA for GPU acceleration. The sample screen shot of using InfiniTAM from the official website is given below at Fig 16.

4.2.3 ORB-SLAM on mobile devices

ORB-SLAM is a monocular real time SLAM system based on detecting feature points of input image, with loop closure detection and camera relocalization function. It's capable of being used in both indoor and outdoor scenes at various scales, and its localization stays robust even with rapid motion of camera.

ORB-SLAM2[16][8] is an open source system builds on previous work of ORB-SLAM,

[8] `https://github.com/raulmur/ORB_SLAM2`

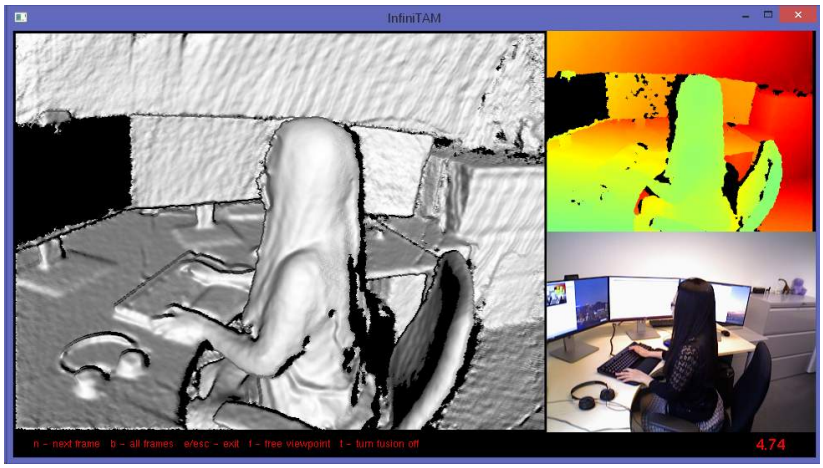

Figure 16: InfiniTAM screenshot

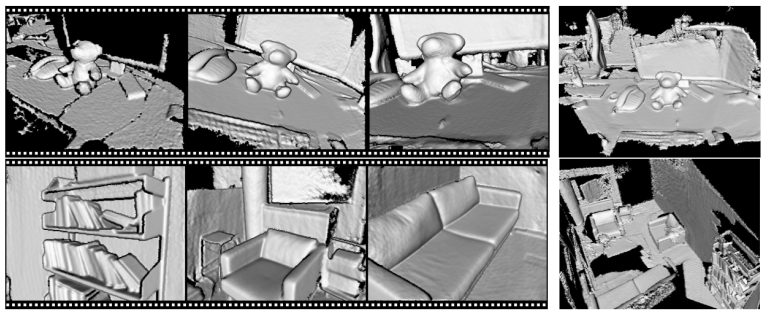

Figure 17: InfiniTAM reconstructed model

supports not only monocular but also stereo and RGB-D cameras used for computing camera trajectory and sparse 3D reconstruction with true scale.

ORB-SLAM system is divided into three thread performing different functions, the tracking thread is responsible for extracting key points from the input image, as shown in picture Fig 18 from the original paper, this is done by using ORB descriptors, which later are used for pose estimation from the last frame and then decide whether this is a new key frame. the LocalMapping thread inserts the key frame and create map points to perform a local BA(Bundle Adjustment) for optimization, and it maintains a covisibility graph for keypoint and pose tracking. Lastly the LoopClosing thread detects loop closure using Bag-of-words method[17], and perform loop correction by optimising a essential graph if loop closure is detected. With loop closure detection, as shown in the picture Fig 19 from original paper, ORB-SLAM is able to compute camera trajectory consist with the world and reconstruct a sparse map of the world in accurate alignment even in large outdoor scenes.

Since ORB-SLAM2 is a robust SLAM system with robust tracking and loop closure function, there has been many work to implement it on mobile devices using its open source code. Here a few of these work that is open sourced on Github is listed with

its dependencies used.

- `ORB_SLAM2_Android`[9] ported ORGB-SLAM2 on Android, using DBoW2[10] for loop detection, clapack[11] and eigen3[12] for math calculation, and g2o[13] for graph optimization and OpenCV for Android. Pangolin[14] was in the original open source code for visualization and user interface and is removed in this implementation.
- `ORB-SLAM-Android`[15] is also an implementation of ORB-SLAM on Android using DBoW2 and g2o.
- `ORB_SLAM_iOS`[16] is implemented on iOS platform with dependencies ported in the same way.

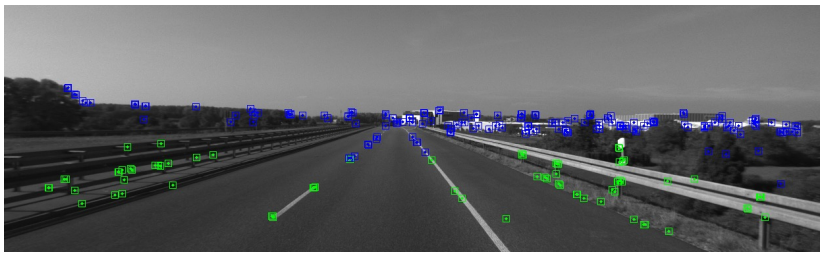

Figure 18: Illustration of feature points detected in ORB SLAM

4.2.4 LSD SLAM

LSD-SLAM(Large-Scale Direct Monocular SLAM)[18][17] is a real time direct monocular SLAM system. It uses pixel intensities of input image to track the pose directly, the alignment two images is done by minimising photometric error using Gauss-Newton algorithm. And depth map is estimated using extended Kalman filter (EKF), a $Sim(3)$ pose-graph of keyframes is maintained to allow for loop closure, using g2o library to optimize the pose graph, this corrects drift occur during the process. As the result, a semi-dense 3D map of point clouds is reconstructed in high accuracy. The structure of LSD-SLAM is shown in Fig 20 using the graph from original paper.

LSD-SLAM not only runs a desktop CPU in real time, it can also run on a modern smartphone for AR application without any GPU acceleration. In this paper [19], an AR application is built on top of direct semi-dense visual odometry using LSD

[9] `https://github.com/FangGet/ORB_SLAM2_Android`
[10] `https://github.com/dorian3d/DBoW2`
[11] `http://www.netlib.org/clapack/`
[12] `http://eigen.tuxfamily.org/index.php?title=3.0`
[13] `https://github.com/RainerKuemmerle/g2o`
[14] `https://github.com/stevenlovegrove/Pangolin`
[15] `https://github.com/castoryan/ORB-SLAM-Android`
[16] `https://github.com/egoist-sx/ORB_SLAM_iOS`
[17] `https://github.com/tum-vision/lsd_slam`

Figure 19: Trajectory reconstructed using ORB SLAM

SLAM. In this experiment the application runs on Sony Xperia Z1 smartphone with Android platform, real time performance of at least 30 fps frame rate is reached makes it capable of used as interactive application, this is due to the separation of tracking and mapping part in the application, and also using NEON optimization for computational heavy parts. The structure of application is shown in Fig 22, also Fig 21 shows the virtual object added in AR application, depth map estimated from the current image and collision mesh created for virtual object to interact with. Both pictures are from the original paper.

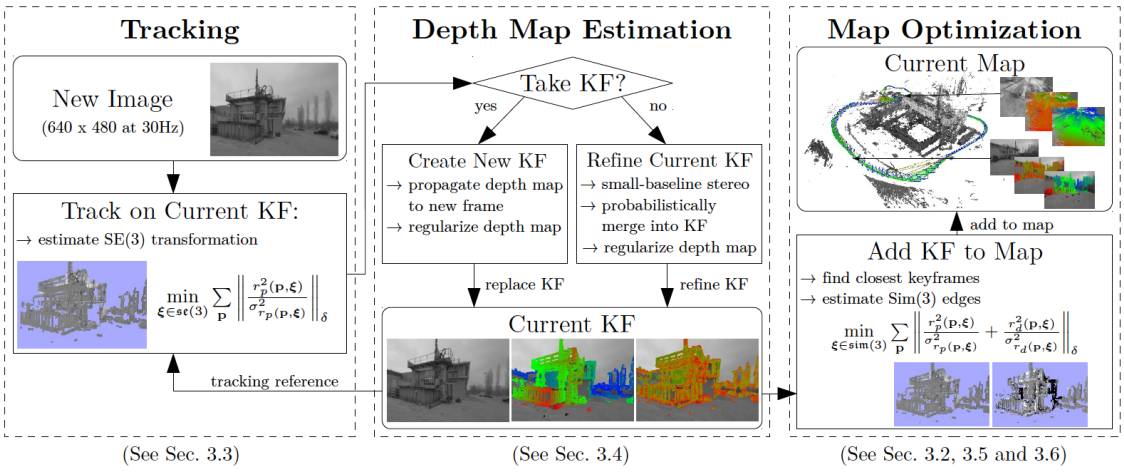

Figure 20: LSD SLAM system structure

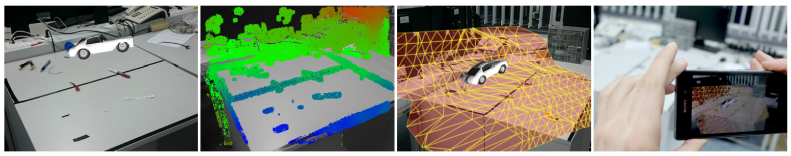

Figure 21: AR illustration of LSD SLAM

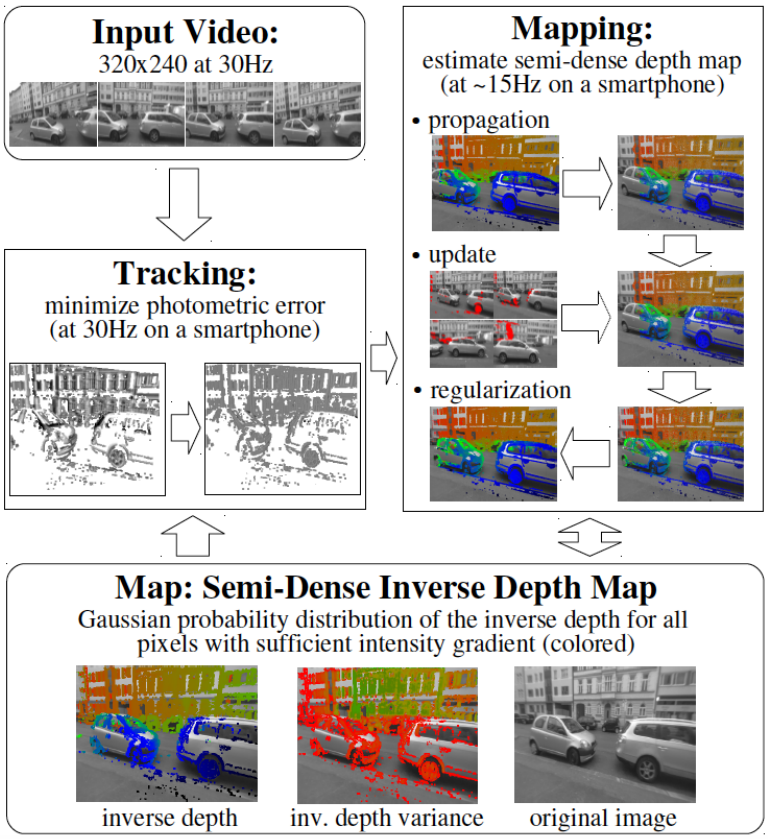

Figure 22: LSD SLAM system illustration

4.2.5 VINS-Mobile

VINS-Mobile[20] is a open source[18] real time Monocular Visual-Inertial State Estimator on mobile devices developed by members in HKUST Aerial Robotics Group. It can run on iOS devices for providing localization function in AR (augmented reality) applications and also on UAV (Unmanned aerial vehicle) for state estimation.

It provides highly accurate visual-inertial odometry by using sliding window optimization, and also enables automatic initialization, relocalisation and loop closure function. The drift error accumulated over time is corrected by optimise the global pose graph maintained. In this picture Fig 23 from the paper [19] can be seen that the virtual object added in the AR application remains in place even after the camera

[18] `https://github.com/HKUST-Aerial-Robotics/VINS-Mobile`
[19] `http://www.ece.ust.hk/~eeshaojie/ismar2017peiliang.pdf`

have travelled a long loopy trajectory.

The program uses ceres solver[21] from Google[20] for non-linear optimization in state estimation stage, the loop closure function is implemented using bag of words method DBoW2[21] similar to ORB-SLAM's loop detection function. The screenshot of AR application is shown in Fig 24 and the structure of the application is shown in Fig 25. Both pictures are from the original paper.

[20] https://github.com/ceres-solver/ceres-solver
[21] https://github.com/dorian3d/DBoW2

5 Project Method Overview

This project develops an Android application running on the Google Tango [1] tablet that can either logs RGB, depth and pose data from Tango to files, or run ElasticFusion on tablet using data from Tango platform, or run ElasticFusion using data from dataset on the tablet.

The application starts with sample Tango demo applications from Google that uses Tango C API, the code for these applications are most written in C++ under the Java NDK(Native Development Kit) on Android platform, which allows it to use the Tango Client C API, the source code for these demo applications can be found on Github[22].

One demo application that is useful for this project is the RGB depth sync application[23], it make use of both Tango's Motion Tracking and Depth Perception feature. And it calculates a depth map by use an upsample method to match the Tango Point Cloud data with each pixel in the RGB image from color camera, and it renders the RGB image on screen with an overlay of grey scale depth map that can adjust the alpha value. The screen shot of only RGB image rendered is shown in Fig 29, and the screen shot with the depth map overlay set to about 0.5 is shown in Fig 31, also screen shot of only depth map rendered(alpha value of 1) is shown in Fig 32.

The application developed in this project make use of the code for Advanced Estimation in Robotics (433H) course practical exercise as the basic frame work, it's also an Android application running on the Google Tango tablet. On top of this application, first using the Tango Client C API the RGB image data, Tango Point Cloud data, and Tango Pose data is acquired. The RGB image data from camera is gain as the RGB color value of each pixel, and the Tango Point Cloud data is in the form of an array of 3D point coordinates, where each 3D point represent a pixel in the depth image where depth sensor has a reading. And the Pose data comes in as the latest pose estimated by Tango VIO(Visual Inertial Odometry) frontend using the IMU onboard and mono image from wide angle motion tracking camera, details of technique used in the Tango VIO can be found in [4] and [22], but in this project it is treated as a black box.

After the Tango Point Cloud Data is acquired, it needs to be process in the same way as in the Tango demo application RGB depth sync, each depth value of 3D point in Tango Point Cloud is upsampled at a window around the point to produce a depth map that is matched with the RGB image.
Then according to which function the user of application choose to run, these data of a frame is stored into a circular buffer of fixed size for use by another thread. If the user choose to log these RGBD and pose data into files, the data logging thread takes the latest frame of data put that was stored in the buffer, and then the depth

[22] `https://github.com/googlesamples/tango-examples-c`
[23] `https://github.com/googlesamples/tango-examples-c/tree/master`

buffer is compressed using the *compress2* function from zlib library[24]. The RGB image data is also compressed into JPEG format using OpenCV library for Android[25]. After that the RGBD and pose data is written to two files respectively, for later use such as using as dataset for running ElaticFusion on desktop computing.

The main purpose of this project is allow ElasticFusion run on the Tango tablet for 3D reconstruction with data from Tango as input, therefore the code for ElasticFusion on desktop is ported and adapted to run on Android under NDK environment. The source code of ElasticFusion is available on Github[26], however in this project an internal version of source code is used. The source code of ElasticFusion included three parts: the Core part contains the code for ElasticFusion algorithm, and can be used as an API for other application, the GIU part is responsible the displaying the user interface of program and handles input setting from the command line, reading from files provided for the program, the there is a GPUTest program that user can test the performance of their GPU for running ElasticFusion.
In this project, only the core part of ElasticFusion is used, the original dependency of Pangolin is removed and replaced with plain OpenGL ES code that full fills the same function, other necessary change is made to enable the ElasticFusion program be run on Android platform. When the user starts ElasticFusion on the Tango tablet, a new ElasticFusion thread is started and it takes RGBD and pose data from the circular buffer previously mentioned, and use these frame data as the input to the program.
Since ElasticFusion program original required high configuration of hardware on desktop, with the processing power on Tango that is much lower than high end desktop computer, all the features of ElasticFusion that requires more processing are turned off on the Tango to make it run faster, such as disabling $\mathfrak{so}(3)$ pre-alignment, use single level tracking pyramid by enabling fast odometry, and disable loop closure detection.
After the user using the device to complete the scanning, the 3D reconstruced surface of the map can be saved into a ply file on the tango tablet, and the saved map can then be viewed using mesh viewing application on the device itself.
Other than running ElasticFusion with data from the Tango tablet, in this project for experimentation purpose the application is also able to use dataset placed on the tablet to run ElasticFusion program, a RGBD data file with pre-specified file name has to be on the tablet before that, and a Pose data file is optional, if the pose data is provided, ElasticFusion will be run using the input pose without using the camera tracking process.

Originally, in the design of this project, the Android application developed is supposed to render the 3D reconstructed surface on the screen in real time, and augmented reality application can be built on it if the implementation of ElasticFusion on Tango tablet is successful and performance is good enough for interactive application(runs at frame rate higher than 30 Hz). However, during the progress of

[24] `http://zlib.net/`
[25] `http://opencv.org/platforms/android/`
[26] `https://github.com/mp3guy/ElasticFusion`

this project, it is found that ElasticFusion runs poorly and very slowly on the Tango tablet, this is due to the data quality from depth sensor is not very good and also the processing power on the tablet is much less than what ElasticFusion requires, also partly caused by the time and effort limitation, the 3D reconstructed surface by the application on Tango tablet is still not 100 percent correct, and there was no enough time before project deadline for doing optimization of the application.

However, other this Android application developed, in this project a simple way of running ElasticFusion in close loop mode using input pose is created to allow for the input pose to be corrected so that the current pose stored in the ElasticFusion is consistent with the input pose, and no camera tracking step is need thus processing time and power is saved. The change of applying this method is implemented both in the application on Tango tablet and the desktop version of ElasticFusion.
The overall structure of the Android application developed in this project is shown in graph Fig 26, the graph clearly shown three different thread used for different functions and the frontend-backend structure of the SLAM system developed in this application.

6 Project Implementation

6.1 Environment Configuration

In this project an Android application is developed to run on the Tango tablet. The Tango tablet runs Android 4.2.2 (KitKat), which corresponds to Android API 19. To be able to develop Android application, Eclipse Mars[27] under Linux Ubuntu[28] operating system is used with installed ADT(Android Development Tools)[29] plugin. The reason for choosing Ubuntu environment is due to the NVIDIA Tegra Android Development Pack version [30] used for compiling CUDA part of ElasticFusion is only available on Ubuntu.

After Eclipse and ADT plugin are installed, Android SDK for API version 19 and the corresponding Android NDK version r13b[31] is downloaded and their path is configured in Eclipse to allow Android NDK application to be developed for the Tango tablet.

Lastly for using the Tango C API to develop Tango application on the tablet, the Tango Client API, Tango 3D Reconstruction API and the Tango Support API is needed, they can be downloaded from the Tango C API example repository on Github [32]. Also contained in the Tango C example repository the *glm(OpenGL Mathematics)* library[33] that is used for matrix calculation is also imported along with the Tango C API files.

To build the source code of the Android application into an APK file to be installed on the Tango tablet, *Android.mk*[34] built system is used. The *Android.mk* file in the *JNI* directory compiles the C++ source source files under Android NDK environment, and links the static library and shared library described in the file to the application. If the Android NDK library path is set up in Eclipse, after importing the project folder into Eclipse, added OpenCV as a library and set up path for NDK include folders, right clicking on the main project directory and choose run configuration will allows Eclipse to build the Application and run on the Tango tablet connected to the computer.

6.2 Tango Platform

Google's Tango[1] platform provides mobile devices with the abilities to sense the 3D space around the device and allow developers to build augmented reality appli-

[27] https://eclipse.org/mars/
[28] https://www.ubuntu.com/
[29] https://developer.android.com/studio/tools/sdk/eclipse-adt.html
[30] https://developer.nvidia.com/AndroidWorks-TADP-Archive
[31] https://developer.android.com/ndk/downloads/revision_history.html
[32] https://github.com/googlesamples/tango-examples-c
[33] https://github.com/googlesamples/tango-examples-c/tree/master/third_party/glm
[34] https://developer.android.com/ndk/guides/android_mk.html

cations on the device. Tango devices have three main functionalities: Motion Tracking, Area Learning, and Depth Perception, these are achieved using the RGB color camera, IMU (accelerometer and gyroscope), depth sensor and wide angle fisheye lens motion tracking camera on Tango devices. The motion tracking feature mainly estimate current position and pose of device using data from IMU and wide angle camera and techniques that use Kalman filter such as [4].

Given the cheap commodity mobile hardware and software, Tango device and tango platform provides the potential to build a fully self-contained 3D dense surface reconstruction solution using state-of-the-art RGB-D(color and depth) SLAM (Simultaneous Localization And Mapping) algorithms in this project. Among all the Tango devices that's released until the date when this report is written, there is the first Tango device the Tango Peanut mobile phone, the Tango "Yellostone" tablet, and Lenovo Phab 2 Pro mobile phone with the latest ASUS ZenFone AR phone. All of them are equipped with Android OS[35], but all the other devices carries Qualcomm processor, except for the "Yellostone" Tango tablet that's used in this project. The "Yellostone" Tango tablet carries Android 4.2.2 (KitKat) OS, this is the latest version of Android update it has until Google has officially stopped support for it. And it's equipped with a quadcore Nvidia Tegra K1 graphics card that supports CUDA(Compute Unified Device Architecture) function, which is used in the ElasticFusion program for camera tracking step. The Nvidia Tegra K1 processor has 192 CUDA cores, and supports CUDA 6.0 and OpenGL ES 3.1. The tablet also has 4GB of RAM shared by the CPU and GPU, an IMU that contains six-axis accelerometer and gyroscope, a wide FOV(field of view) motion tracking camera runs at 60Hz, and a projective depth sensor runs at 3Hz, and lastly a 4 MP RGB camera runs at 30Hz. The unique hardware configuration of the "Yellowstone" tablet mades it the perfect Tango device for this project, since with CUDA supported the ElasticFusion doesn't need a big rewrite for the camera tracking part.

The Tango SDK for developers is offered in three programming languages: C and Java, Unity. The this project only the C API is used since C++ is what ElasticFusion is originally written, therefore using the Tango C API enables the code for Tango frontend to be combine with ElasticFusion without a huge amount of work for rewriting. Also the program runs faster in C under NDK environment on Android, where performance is crucial for the ElasticFusion program.

The Motion Tracking feature on Tango platform provides the position and pose of Tango device of a certain frame with respect to a source frame at any time, the position and pose data is encapsulated inside a C++ struct *PoseData*, the simplified version of this structure is given below.

```
struct PoseData {
       double timestamp;
   double orientation[4];
   double translation[3];
```

[35] https://www.android.com/

```
}
```

The orientation array defines a quaternion that represents the rotation of the target frame with respect to the base frame. And the translation array represents the 3D vector of displacement between the target frame and the base frame. For the pose data to be able to be used by ElasticFusion, it have to be converted to affine transformation matrix, a transformation matrix is a 4 by 4 matrix with a 3 by 3 rotation matrix at left up corner and a 3D translation vector at the right most column. At time t the global pose of the camera $\boldsymbol{P}_t$ (w.r.t. global frame $\xrightarrow{F}_G$) can be shown below, where rotation matrix is $\boldsymbol{R}_t \in SO(3)$ and translation vector is $\boldsymbol{t}_t \in \mathbb{R}^3$.

$$\boldsymbol{P}_t = \begin{bmatrix} & R_t & t_t & \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \in \mathbb{SE}(3) \tag{1}$$

In the application developed, to get the pose of Tango device, the C API function `TangoService_connectOnPoseAvailable` is used, this function take a pair of frame reference, which represents the coordinate of frame that pose is in, in this case is `TANGO_COORDINATE_FRAME_START_OF_SERVICE` and `TANGO_COORDINATE_FRAME_DEVICE`, which means the pose would be the current device pose with respect to the pose device is in when application start. Another parameter is a callback function, the callback function will be called when a new Tango pose is available, and the pose data will be given in the parameter. The signature of this functions is given below.

```
TangoErrorType TangoService_connectOnPoseAvailable(
    uint32_t count, const TangoCoordinateFramePair* frames,
    void (*TangoService_onPoseAvailable)(void* context,
                                         const TangoPoseData* pose)
```

Another way for getting pose data from Tango is the function `TangoService_getPoseAtTime`, this function pulls Tango pose data from the give as parameter, if this parameter is set to 0.0, the latest estimated pose is returned with respect to the coordinate of frames given. The signature of this function is shown below.

```
TangoErrorType TangoService_getPoseAtTime(double timestamp,
                                          TangoCoordinateFramePair frame,
                                          TangoPoseData* pose);
```

The complete structure of Tango Pose Data struct is shown below, however only timestamp, orientation and translation are used to be processed for running ElasticFusion.

```
typedef struct TangoPoseData {
  uint32_t version;
  double timestamp;
  double orientation[4];
  double translation[3];
```

```
  TangoPoseStatusType status_code;
  TangoCoordinateFramePair frame;
  uint32_t confidence;
  float accuracy;
} TangoPoseData;
```

In the Tango application developed, the RGB image data from color camera is not only being rendered to the screen, but also to be input to the ElasticFusion part for 3D reconstruction, and this is achieved by using Tango API and OpenGL ES textures. A class called *GlCameraFrame* is responsible for manage OpenGL ES programs and shaders and texture, and then render the image stored in gl texture to the screen using function *glDrawElements*. To acquire the RGB image from color camera and store it into textures, three Tango API functions are used. The first one is `TangoService_connectTextureId` which the function signature is given below, it takes in a camera id, which in this case is `TANGO_CAMERA_COLOR` for the color camera, and an OpenGL ES texture id. It connects the texture id of the texture to the camera which id is given in the parameter. This allows the second function `TangoService_updateTexture` to be used, it updates the OpenGL ES texture that has been connected to the camera with id given in the parameter, fills in the latest image from that camera, and also returns the timestamps of the image acquired by the camera. The signature of this function is shown below. Lastly another function `TangoService_updateTextureExternalOes` is used so that a specific OpenGL ES EXTERNAL OES texture can be filled with the image acquired by given camera id, it takes in valid OpenGL ES EXTERNAL OES texture id as parameter to update the texture associated with that id, the function signature is also given below.

```
TangoErrorType TangoService_connectTextureId(
    TangoCameraId id, unsigned int tex, void* context,
    TangoService_OnTextureAvailable callback);
```

```
TangoErrorType TangoService_updateTexture(TangoCameraId id, double*
    ↪ timestamp);
```

```
TangoErrorType TangoService_updateTextureExternalOes(TangoCameraId id,
                                                     unsigned int tex,
                                                     double* timestamp);
```

After the RGB image data is stored in the OpenGL ES texture, these data can be read out from the OpenGL ES framebuffer that's associated with the texture using function *glReadPixels*, the RGB data read out is store in a byte buffer which every value is corresponding to the RGB value of each pixel. This step is done in the *GlCameraFrame* class, and the RGB data stored in byte buffer can be later used to write to files or feed into ElasticFusion program.

Another function of Tango platform have been used in the application is Depth Perception, the depth data acquired from the depth sensor is stored in a C++ struct

by tango call *TangoPointCloud*, TangoPointCloud includes the time when the sensor captures this depth data, the number points contained in this data, and the array of 3D data points with their confidence level in the range of [0, 1]. In the 3D point coordinates, X and Y represents the position of data point corresponding to the pixel, and the Z component is the depth that was captured. The definition of this struct is shown below.

```
typedef struct TangoPointCloud {
  uint32_t version;
  double timestamp;
  uint32_t num_points;
  float (*points)[4];
} TangoPointCloud;
```

On Tango platform, a struct called `TangoSupportPointCloudManager` is used to manage point cloud data, it can move the pointcloud data from the callback thread to other threads for render or processing usage. The PointCloud manager is created using the function below where max_points the maximum amount of 3D points allowed to be stored in it.

```
TangoErrorType TangoSupport_createPointCloudManager(
  size_t max_points,
  TangoSupportPointCloudManager **manager
)
```

To register the call back for point cloud data, a similar function to pose data function is called, `TangoService_connectOnPointCloudAvailable` takes in a call back function, so that it will be called when new pointcloud data is available. The function signature is given below.

```
TangoErrorType TangoService_connectOnPointCloudAvailable(void(*)(void
    ↪ *context, const TangoPointCloud *cloud)
```

In the call back thread, point cloud manager is used to put the new point cloud data into the buffer within it, so that the data can be used later, the function `TangoSupport_updatePointCloud` is called to swaps the buffer inside point cloud manager for new point cloud data.

```
TangoErrorType TangoSupport_updatePointCloud(
    ↪ TangoSupportPointCloudManager *manager, const TangoPointCloud *
    ↪ point_cloud)
```

In other thread that does computation or rendering, point cloud data can be acquired from the point cloud manager use the function `TangoSupport_getLatestPointCloudAndNewDataFlag`, it returns the latest point cloud data stored inside the manager, and also returns a flag for the user to know whether

the point cloud data acquired is new compared to the last time. The function signature is shown below.

```
TangoErrorType TangoSupport_getLatestPointCloudAndNewDataFlag(
  TangoSupportPointCloudManager *manager,
  TangoPointCloud **latest_point_cloud,
  bool *new_data
)
```

After RGB and depth data is acquired, processing is needed to later usage, since both RGB and depth data are acquired with respect to their own sensors, therefore correction is needed to place the data under the same frame, in this application, the depth data is transformed to be align with the RGB data. However, the translation between depth sensor and color camera can not be simply calculated since the time of each data being captured is slight different. On tango platform, the function `TangoSupport_calculateRelativePose` is used to calculate relative transformation between different coordinate of frame and timestamp, which in this case is `TANGO_COORDINATE_FRAME_CAMERA_COLOR` and `TANGO_COORDINATE_FRAME_CAMERA_DEPTH`, this function outputs a Tango pose data instance to represent the related transformation calculated. This pose data is then converted into a transformation matrix to bring depth data into alignment with RGB data. The function signature is shown below.

```
TangoErrorType TangoSupport_calculateRelativePose(double
    ↪ base_timestamp, TangoCoordinateFrameType base_frame, double
    ↪ target_timestamp, TangoCoordinateFrameType target_frame,
    ↪ TangoPoseData *base_frame_T_target_frame)
```

In the ElasticFusion program, the camera intrinsics data have to be provided for image processing, these information can be acquired directly on Tango using the Tango API, function `TangoService_getCameraIntrinsics` returns a `TangoCameraIntrinsics` struct instance corresponds to the input camera id. The Intrinsics struct contains the image width and height, focal length, and distortion coefficients. These information is later being input to the ElasticFusion. The function signature and struct definition is given below.

```
TangoErrorType TangoService_getCameraIntrinsics(TangoCameraId cameraid
    ↪ , TangoCameraIntrinsics* intrinsics);
```

```
typedef struct TangoCameraIntrinsics {
  TangoCameraId camera_id;
  TangoCalibrationType calibration_type;
  uint32_t width;
  uint32_t height;
  double fx;
```

```
  double fy;
  double cx;
  double cy;
  double distortion[5];
} TangoCameraIntrinsics;
```

6.3 Basic running framwwork

This as application in this project is developed on of the code for Advanced Estimation in Robotics (433H) course practical exercise which was used as a basic frame work, it's an Android application running on the Google Tango tablet that made use of the Tango C API.

To start the project, all the unnecessary code in the application is removed. The code left in the application simply renders the RGB image acquire from the color camera onto the screen, this is done by creating a *GLSurfaceRenderer* class inherited from the *Renderer* class in Android *GLSurfaceView*, and a *CameraOverlay* inherited from the *SurfaceView* and implements the call back function in Android *SurfaceHolder*.

To aquire the live RGB image captured by the color camera, the Tango C API is used, by using the `TangoService_connectTextureId` from the API, a created OpenGL ES texture is connected to the color camera, which allows later the caprtured image to be stored in the OpenGL ES texture. After that the OpenGL ES function *glDrawElements* is used to render the image to the screen.

6.4 Logging data from Tango

After the RGBD and pose data from Tango are acquired, data of each frame is encapsulated into a C++ struct *RGBDPosedata* and stored in a circular buffer. The definition of *RGBDPosedata* is given below, other than RGBD and pose data, it also contains both the timestamp for depth and RGB data for calculating the transformation between color camera and depth sensor at the time their data is captured.

```
struct RGBDPosedata {
  double pointCloudTimestamp;
  uint32_t pointCloudNumpoints;
  float *pointCloudPoints;
  uint8_t *image;
  double colorTimeStamp;
  int64_t m_lastTimestamp;
  TangoPoseData pose;
};
```

The first function implemented in the application developed is to write and RGBD and pose data recorded into their files, so that they can be used as dataset to run ElaticFusion on desktop computer. The processing of data into correct format is done in a data logging thread to reduce the workload on the main thread, and so that the writing of data is asynchronous with the capture for data. The circular buffer for storing data is an array of *RGBDPosedata* structs with fixed size 50, after data is captured in the main thread and encapsulated into a *RGBDPosedata* struct instance, it's stored in the position in circular buffer pointed by a variable called *latestBufferIndex*. This variable records the last position in the array where data is stored and so that data can be put into the front of array when the end of buffer is full, thus the circular effect is achieved where old data is gradually replaced by new data come in. This variable also allows the data logging thread to always read the latest frame of data that was stored in the buffer, in this way the speed difference between data producing and consuming is handled.

The code processing and writing RGBD and pose data is combined and adapted from the Google Tango application RGB depth synchttps://github.com/googlesamples/tango-examples-c/tree/master which produce a depth map in sync with the RGB image, and data logger for ElasticFusion[36], which originally is used to save data from RGBD camera that supports OpenNI[37] interface. Firstly the relative transformation between RGB and depth coordinate of frame and timestamp is calculated using the function `TangoSupport_calculateRelativePose` which outputs a *TangoPoseData* instance as result, this pose data is then converted into a transformation matrix for producing the depth map. To produce the depth map, firstly the 3D point coordinates in Tango point cloud is brought into alignment with RGB data using the transformation matrix calculated above, after that an upsample method is used filter

[36] https://github.com/mp3guy/Logger1
[37] https://structure.io/openni

the Tango Point Cloud data for smoothness, in the process the depth value of each 3D point coordinate in Tango Point Cloud is upsampled at a fixed size window around the 3D point's X and Y value, which corresponds to the pixel coordinates in the RGB image from color image. In this way, the depth map produced is in sync with the RGB image in both time and coordinate of frame, after that the data compressed for writing into files.

In order to save space and reduce amount of data written to files, the RGB image data is compressed into JPEG format from byte buffer using the OpenCV library for Android[38], also the depth buffer is compressed using the *compress2* function from zlib library[39]. After that the timestamp, compressed RGB data and depth data with their sizes of each frame is written to the RGBD data file. and the pose data of each frame is written to the pose data file in the order of: *x, y, z, qx, qy, qz, qw* where *x, y, z* is the 3D position of the device and *qx, qy, qz, qw* is the elements of quaternion that represents the orientation of the device.

After the data being saved to two files on the Tango tablet, a slight modification is needed for the original ElasticFusion program to be able tu use them as datasets. The ElasticFusion program is designed to be able to use data from datasets, however in the *GIU* part of ElasticFusion program the *Resolution* struct instance that holds width and height of RGB image is hard-coded with fix width and height, while the *Intrinsics* struct instance can use the focal length and distortion parameter from a given calibration file, therefore a small change is made to the *GIU* part of ElasticFusion program to allow for both image size and intrinsics data to be read from the calibration file.

Using the logging function implemented in the application, experiments were carried out on desktop computer to run ElasticFusion using data captured from the Tango tablet for 3D reconstruction, the screen shots of that is shown in Fig **??**, Fig 34, Fig 35 and Fig 36. From the screenshots it can be seen that the surface is able to be reconstructed however the points in the global map is vert sparse and the speed is very slow, this is mainly due to the noise and sparseness of depth data from depth sensor, also the pose estimated from Tango may not be very accurate, as the RGB image has even higher resolution (1980 x 720) then the image size in original ElasticFusion program. Also to allow for the surface to be reconstructed evenly instead of sparse points, when recording data using the Tango tablet, the device has to be held and move in a very slow and stable fashion.

[38] `http://opencv.org/platforms/android/`
[39] `http://zlib.net/`

6.5 Compiling ElasticFusion on Tango

As mentioned above, the desktop ElasticFusion program includes three parts: the *Core* part contains the code for ElasticFusion algorithm, and can be used as an API for other application, the *GIU* part is responsible the displaying the user interface of program and handles input setting from the command line, reading from files provided for the program, the there is a *GPUTest* program that user can test the performance of their GPU for running ElasticFusion. In this project the *core* part is compiled and adapted on the Tango tablet to allow it to run on Android platform, and please be mindful that although open source code of ElasticFusion is available on Github[40], in this project an internal version of source code is used for its better structure.

6.5.1 Dependencies

Originally in the ElasticFusion program, the dependencies include OpenGL 3.0[41] that is used to manage the global map and predict the views, CUDA(Compute Unified Device Architecture) 7.0 that is used to implement the reduction process in camera tracking, OpenNI2[42] that is used to connect the program to a RGBD camera such as Microsoft Kinect or ASUS Xtion Pro Live. And SuiteSparse[43] is used to do solve the Cholesky decomposition for optimization problem during camera tracking process. The header only library eigen3[44] is used for matrix calculation. The zlib library[45] is used uncompress depth data from dataset file and the libjpeg library[46] is used to uncompress the RGB image data in dataset file from JPEG format into RGB pixel data. Lastly the Pangolin[47] library that is used for building the GUI of program and also manage some OpenGL objects. Lastly the Boost C++ library[48] is mainly used for thread management in the application.

Among the dependencies mentioned above, the OpenGL part in the source code of ElasticFusion is adapted to OpenGL ES for running on Android, and OpenNI is not used as since it's not needed in the Core part of ElasticFusion, the libjpeg library is removed and its function is implemented using OpenCV instead. Lastly the Pangolin library is removed and the part where it manages OpenGL objects is replaced with plain OpenGL ES implementation that is adapted from the Pangolin source code. The rest of dependencies are all compiled to be able to used in the Application for running on Android.

[40] https://github.com/mp3guy/ElasticFusion
[41] https://www.opengl.org/
[42] https://structure.io/openni
[43] http://faculty.cse.tamu.edu/davis/suitesparse.html
[44] http://eigen.tuxfamily.org/index.php?title=3.0
[45] http://zlib.net/
[46] http://libjpeg.sourceforge.net/
[47] https://github.com/stevenlovegrove/Pangolin
[48] http://www.boost.org/

6.5.2 Hardware requirements

Since ElasticFusion is a RGBD SLAM system for 3D dense reconstruction, it needs to process full RGB image data as well as the depth map, also the pose estimation made heavily use of CUDA for GPU general data processing in the camera tracking stage if no dataset is provided, not to mention the local and global loop closure detection in each frame. Therefore by the hardware requirement for running ElasticFusion is already very high, quote from the *README* of ElasticFusion on Github: "A very fast nVidia GPU (3.5TFLOPS+), and a fast CPU (something like an i7).", 3.5T FLOPS(Floating-point operations per second) is equal to 3500 G FLOPS, this kind of processing power is only available on graphics card like Nvidia GeForce GTX 970 which has 1664 CUDA cores. However the Nvidia Tegra K1 equipped on Tango tablet has only 192 CUDA core, and with 192 CUDA cores * 2 FLOPS per core * 950MHz, the processing power is only about 365 G FLOPS, which is approximately the power on a GT620 graphic card, only about one tenth what is required. This kind of processing power is considered quite high on a mobile device, however is still too low compare to the requirement.

6.5.3 OpenCV

The OpenCV library for Android[49] is used to encode the RGB image data to jpeg when saving Tango data to files and used to decode the color image data from jpeg format to RGB pixel data when ElasticFusion is running using datasets on the Tango tablet.

The Android version of OpenCV 2.4 source code in C++ is simply downloaded and imported in Eclipse as a project and attached to the Android application as a library. After that the android makefile of OpenCV *OpenCV.mk* is included in the *Android.mk* file of the Android application, so that OpenCV is compiled and linked as a static library for the application and the code in application can use it as normally done in C++ programs.

6.5.4 CUDA

CUDA (Compute Unified Device Architecture) is a compute platform for general purpose computing on Nvidia graphic card. CUDA 7.0 is needed in ElasticFusion program and since CUDA 6.0 supported by the NVIDIA Tegra K1 processer on Google Tango tablet, no change is needed to make to adapt this part of code.
In the *core* part of the ElasticFusion code, a folder nameed *CUDA* contains all the files that includes source code run on GPU using CUDA, include it there are normal C++ files and CUDA head files with extension ".cuh" and CUDA source files with extension ".cu".

[49] `http://opencv.org/platforms/android/`

To compile the code inside this folder for the Android application on Tango tablet, under Ubuntu environment the NVIDIA Tegra Android Development Pack version 3.0r4[50] need to be download and installed. It contains the Android CUDA toolkit for CUDA 6.0 and can be used to compile CUDA code for Android platform.

Firstly, the code in *CUDA* folder is compiled into a static library by creating an *"Android.mk"* makefile, in the makefile CUDA *nvcc* compiler and *g++* compiler from the Android standalone tool chain is used to compile the CUDA files and C++ files in the folder. After that a library file *libCudaLib.a* is created and in the *"Android.mk"* makefile for application the *libCudaLib.a* is linked as a static library to the Android application. This allows the source code of ElasticFusion in C++ to call the CUDA code inside *CUDA* folder and no change is needed to made.

6.5.5 Pangolin

The Pangolin[51] library in the ElasticFusion source code was used to render the GUI of program and manage some OpenGL objects, however the *Core* part of ElasticFusion does not contain GUI part and only use Pangolin in a few place, therefore compiling the Pangolin library on Android is an overkill. And so when porting the code for ElasticFusion onto Tango tablet, Pangolin is removed, and the part that Pangolin is used to manage encapsulated OpenGL objects is replaced by plain OpenGL ES code extracted from Pangolin's source code.

The *GlTexture* class in Pangolin stores an OpenGL texture object and its behaviour methods such as binding and uploading texture image. When adapting the ElasticFusion program on Tango tablet, the source code for *GlTexture* class in Pangolin is extracted and put into a file in the ElasticFusion program, and then is place in ElasticFusion where Pangolin's *GlTexture* is used is then replaced by the new *GlTexture* class. The *Shader* class originally in the ElasticFusion program is inherited from the *GlSlProgram* class in Pangolin, it's mainly used for managing OpenGL programs and attached shaders. This class is adapted in a similar way, the part of source code that is useful in Pangolin is taken out and put inside the *Shader* class. The *GlRenderBuffer* and *GlFramebuffer* class from Pangolin used are also adapted in similar way. The table 1 given below includes all the Pangolin values and class adapted or removed in the ElasticFusion program on Tango tablet.

6.5.6 SuiteSparse

SuiteSparse[52] is a suite of sparse matrix software, and it's used in the ElasticFusion program to do solve the Cholesky decomposition for optimization problem during

[50] `https://developer.nvidia.com/AndroidWorks-TADP-Archive`
[51] `https://github.com/stevenlovegrove/Pangolin`
[52] `http://faculty.cse.tamu.edu/davis/suitesparse.html`

Table 1: Pangolin values and class adapted or removed

pangolin::GlTexture
pangolin::GlSlProgram
pangolin::GlSlVertexShader
pangolin::GlSlGeometryShader
pangolin::GlSlFragmentShader
pangolin::GlRenderBuffer
pangolin::GlFramebuffer
pangolin::OpenGlMatrix

camera tracking process. For doing Cholesky decomposition, the actual library used in ElasticFusion is the *cholmod* library, however the *cholmod* library depends on other libraries in SuiteSparse.

The dependency of *cholmod* library includes the `SuiteSparse_config`, AMD, CAMD, CCOLAMD, COLAMD, CXSparse, clapack[53] and metis[54]. To compile *cholmod* library for Android platform, first clapack and metis are compiled into static libraries since both their source code is not in the SuiteSparse source code. The srouce code for clapack is downloaded from the official website and the source code for metis-5.1.0[55] is used. After they are compiled, two files *libmetis.a* and *libclapak.a* are generated, they are then used to compile the choldmod library using other compiled dependencies of cholmod library into a static library file. Finally the choldmod static library is linked in the *Android.mk* makefile for Android application so that the cholmod library can be used.

6.5.7 OpenGL

OpenGL[56] is used to manage map and predict views in the ElasticFusion program, it mainly made use of the transform feedback feature for off-screen rendering in OpenGL using the Vertex shader, Geometry shader and Fragment shader. To adapt this part of code on the Android platform for Tango tablet, the original OpenGL language need to be change to OpenGL ES language for mobile platform. The Tango tablet supports OpenGL ES version 3.1[57], this is the latest version of OpenGL carried on Android API version 19, corresponds to Android 4.2.2 (KitKat), which is the last official update of Tango tablet gets. Since Google has officially stopped to support the Tango tablet, no newer version of Android can be installed and therefore newer version of OpenGL ES cannot be acquired on the tablet. The OpenGL ES version 3.1 on Tango tablet does not some features that's used in ElasticFusion program using OpenGL, so in some places workaround is used to adapt the code.

[53] `http://www.netlib.org/clapack/`
[54] `http://glaros.dtc.umn.edu/gkhome/metis/metis/download`
[55] `https://github.com/scibuilder/metis`
[56] `https://www.opengl.org/`
[57] `OpenGLES3.1https://www.khronos.org/registry/OpenGL-Refpages/es3.1/`

Table 2: EGL functions used

eglGetDisplay
eglInitialize
eglChooseConfig
eglCreateContext
eglCreatePbufferSurface
eglMakeCurrent

Firstly, in the Android application on Tango tablet, OpenGL ES is already used to render the RGB image acquired from color camera to the screen, and the ElasticFusion program ported to the application is designed to be running on a separate thread. To let the ElasticFusion thread use OpenGL ES, a new OpenGL ES context need to be created and attached to the ElasticFusion thread, this is done by using the EGL[58] platform available under Android NDK environment, a new EGL pixel buffer surface is created using the *eglCreatePbufferSurface* function in EGL to allow for off-screen rendering into buffer using OpenGL ES in this thread. The EGL functions used to do that is given in the table 2 below.

After the OpenGL ES context is created, original code in OpenGL needs to be adapted. In the original ElasticFusion code, the code for shaders are written in separate files and are read by Pangolin in the program to produce a C++ string of type *const char ** that is then used as the source code for shaders by function *glShaderSource*. However it will be troublesome to do the same thing on Android application, also because the Pangolin dependency is removed, the source code for shaders are all put into hardcoded C++ char array of type *static const char []* for fast reading in a file, the source code string is then placed into a tupe together with the name of the shader for east identification of shader source code. The Geometry shader originally stored in the file `copy_unstable.geom` is adapted and the created code is shown below as an example.

```
static const char copy_unstablegeom_source[] =
    "#version 310 es\n"
    "#extension GL_EXT_geometry_shader : require\n"
    "precision highp float;\n"
    "precision highp int;\n"
    "layout(points) in;\n"
    "layout(points, max_vertices = 1) out;\n"
    "in vec4 vPosition[];\n"
    "in vec4 vColor[];\n"
    "in vec4 vNormRad[];\n"
    "flat in int test[];\n"
    "out vec4 vPosition0;\n"
    "out vec4 vColor0;\n"
```

[58] `https://www.khronos.org/egl`

Table 3: OpenGL ES Shader adaptation

```
                                    "#version 310 es\n"
                                "precision highp float;\n"
                                  "precision highp int;\n"
                               "precision lowp sampler2D;\n"
                              "precision lowp usampler2D;\n"
  "#extension GL_EXT_geometry_shader : require\n"
```

```
    "out vec4 vNormRad0;\n"
    "void main() {\n"
    " if(test[0] > 0) {\n"
    " vPosition0 = vPosition[0];\n"
    " vColor0 = vColor[0];\n"
    " vNormRad0 = vNormRad[0];\n"
    " EmitVertex();\n"
    " EndPrimitive(); \n"
    " }\n"
    "}\n";
static const std::tuple<std::string, std::string>
   ↪ copy_unstablegeom_tuple =
    std::make_tuple("copy_unstablegeom", copy_unstablegeom_source);
```

Other than putting the source code of shaders into char array, the actual code for shaders also need to be changed for OpenGL ES format, firstly the version indicator at the top of shader code is changed from `#version 330 core` to `#version 310 es` since the Tango tablet run at OpenGL ES version 3.1. Also the precision need to set for numerical types *int* and *float* and also the texture samplers. In OpenGL the data types *int* and *float* can be converted to each other implicitly, however this has to be done explicitly in OpenGL ES so type conversion is manually added to the shader source code. Lastly in the code for Geometry shaders the OpenGL extension for it need to be declared using
`#extension GL_EXT_geometry_shader : require`. The table 3 below gives all the declarations added to the beginning of shader source code.

After the shader source code is adapted, other OpenGL code need to be changed as well. In OpenGL the format and internal format of textures can be totally different, for example in the ElasticFusion code RGB image texture format use `GL_RGB` as format and `GL_RGBA` as internal format, however this is not allowed in OpenGL ES, also some OpenGL texture formats used in the ElasticFusion program is not available in OpenGL ES, so adaptation is made using similar type of format such as changing from `GL_RGB` to `GL_RGBA` and use `GL_RED` to replace `GL_LUMINANCE`. Other than some function of transform back feature uses the Nvidia extension in OpenGL but not available in OpenGL ES, such as the *glGetBufferSubData* function, this is replaced by the *glMapBufferRange* function. With functions that simply doesn't exist in OpenGL ES and also no replacements like *glDrawTransformFeedback*, a work around is used,

Table 4: OpenGL ES adaptation

original function or values	adaptation
glDrawTransformFeedback	extract function pointer using eglGetProcAddress
glGetBufferSubData	glMapBufferRange and glUnmapBuffer
GL_RGB	GL_RGBA
GL_LUMINANCE16UI_EXT	GL_R16UI
GL_LUMINANCE_INTEGER_EXT	GL_RED_INTEGER
GL_LUMINANCE32F_ARB	GL_R32F
GL_LUMINANCE	GL_RED
glGetVaryingLocationNV	removed
glTransformFeedbackVaryingsNV	glTransformFeedbackVaryings, changed to be used before linking the shader to OpenGL ES program
glPushAttrib	removed
glPopAttrib	removed

by defining the function ourselves like below and using the *eglGetProcAddress* function in EGL to retrieve the function pointer, the actual function pointer of OpenGL in driver of graphic card can be retrieved if they exist, then by casting the function pointer back to the function signature, this function can be used normally.

```
void glDrawTransformFeedback(GLenum mode, GLuint id);
typedef void(GL_API_CALL_P PFNGLDRAWTRANSFORMFEEDBACKNVPROC)(GLenum
    ↪ mode, GLuint id);
```

The similar method is used to define the GL_GEOMETRY_SHADER value for using Geometry shader, since the OpenGL ES version 3.1 on Tango tablet does not support Geometry shader on its own. However the OpenGL implementation inside the Nvidia Tegra K1 graphic card driver supports Geometry shader, so the work around is usable.

Also in order to check for error in the OpenGL ES code, the function *glGetError* and *glCheckFramebufferStatus* are used to detect error and incorrect OpenGL ES framebuffer status that may not cause the application to crash, since normal OpenGL ES error will not be logged automatically. The function signatures are given below.

```
GLenum glGetError(void);
GLenum glCheckFramebufferStatus(GLenum target);
```

A table 4 is given below to list the adapted functions of OpenGL in the original ElasticFusion program.

6.5.8 Eigen

Eigen[59] is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms. In the ElasticFusion program, Eigen is used to store the position and pose data uisng Eigen structures like *Eigen::Vector4f* for homogeneous coordinates and *Eigen::Matrix4f* or *Eigen::Isometry3f* for pose data representing in affine transformation matrix.

Eigen3[60] source code is used in the Application on Tango tablet, since it's just a header only library that uses templates, the source code folder is pasted in the project source code and the header needed is simply included in the code that use Eigen.

6.5.9 Eigen

In the Android application developed on Tango tablet, Boost[61] is mainly use for manage thread, such as creating new thread to compress RGB image and depth data in the data logging function, or creating new thread to run ElasticFusion program. The version of Boost used in the application is 1.53.0[62], the source code is downloaded from Github[63], which is a built version of Boost for Android using the code from `https://github.com/sorccu/Boost-for-Android`. The Boost library is compiled into a static library and it's linked to the main application in the *Android.mk* makefile.

[59] `http://eigen.tuxfamily.org/`
[60] `http://eigen.tuxfamily.org/index.php?title=3.0`
[61] `http://www.boost.org/`
[62] `http://www.boost.org/doc/libs/1_53_0/`
[63] `https://github.com/emileb/Boost-for-Android-Prebuilt`

Table 5: OpenGL ES configuration

glReadPixels
`glPixelStorei(GL_UNPACK_ALIGNMENT, 1);`
`glPixelStorei(GL_PACK_ALIGNMENT, 1);`
`glEnable(GL_DEPTH_TEST);`
`glDepthMask(GL_TRUE);`
`glDepthFunc(GL_LESS);`

6.6 Running ElasticFusion on Tango

After the ElasticFusion program is compiled on the Tango tablet, actual functions need to be implemented to run the ElasticFusion program using different data sources. In the Android application developed in this project, the ElasticFusion can be either run using the live RGBD and pose data from the cameras and IMU sensor on board, or run using the `dyson_lab.klg` dataset that was used as the example in ElasticFusion paper[2], or running using the Tango RGBD and pose data saved earlier in files. However, they are all implemented in similar way by starting a new thread to run ElasticFusion.

6.6.1 Running ElasticFusion using data from Tango

To run the ElasticFusion program using the RGBD and pose data from Tango tablet, a similar method like the data logging function is used. When the button for this function is pressed on screen, the function called is passed from the event listener in Java part of the application to the C++ part using the Java Native Interface (JNI) on Android platform. Firstly it is checked that no other function is running before the ElasticFusion thread can start, after that when the function started, the RGBD and pose data from Tango will be put into the circular buffer of *RGBDPosedata* structures in the class for this function.
On the other hand, a new ElasticFusion running thread starts when the button is pressed, firstly a new OpenGL context is created and attached to the current thread as described above, and then the *Resolution* and *Intrinsics* structs from ElasticFusion are configured using Tango camera data acquired use the `TangoService_getCameraIntrinsics` function. After that the *ElasticFusion* class is initialised using configurations that makes the program run at fastest speed, such as disable SO(3) pre-alignment, enabling fast odometry by using single level tracking pyramid, disable pose bootstrap, and disable loop closure detection function. Also the *confidenceThreshold* value is set to 0.45 instead of the default value 10.0, the reason for doing it will be explain later. Before the ElasticFusion can process any data, OpenGL ES is configured using the code shown below in Table 5.

After the set up is done, inside a whileloop the latest RGBD and pose data structure instance from the circular buffer is read out if new data is available. Then the RGBD and pose data is converted into the correct format for ElasticFusion to process.

In the adaptation of OpenGL code to OpenGL ES code, the format of RGB image has to be changed from `GL_RGB` to `GL_RGBA` in order to match the limitation of OpenGL ES, therefore in order to use the color image data in `GL_RGBA` format when the image data is read out in the *GlCameraFrame* class using the *glReadPixels* function, it's read out in `GL_RGBA` format instead of the `GL_RGB` format used in data logging function.

In the data processing stage, the depth data is brought into alignment with the camera data coordinate of frame, up-sampled use the same method as in data logging function and converted to millimetre scale to produce a depth map, and the pose data is converted into a *Eigen::Matrix4f* structure for ElasticFusion.

Then the data is processed by calling the *processFrame* function of ElasticFusion with RGB image data, depth data, pose and timestamp, which the function signature is shown below.

```
void processFrame(const unsigned char *rgb, const unsigned short *
    ↪ depth,
const int64_t &timestamp, const Eigen::Matrix4f *inPose = 0,
const float weightMultiplier = 1.f,
const bool bootstrap = false);
```

After a frame is processed, the state of ElasticFusion program can be checked by getting some critical values store in it such as the current number of surfels in the global map or the number of frames storing in the fern database. These data is logged out and can be viewed using the *adb logcat* command on computer if the Tango tablet is connecting to it.

The purpose of running ElasticFusion program is to dense 3D reconstruction of the space that the Tango tablet is scanning, therefore the 3D reconstructed map by ElasticFusion must be able to saved in a ply file for viewing or other usage as the desktop version does. On the application developed this can be done by simply pressing the "save" button on screen, this will set a thread safe flag using "mutex lock", so that after a data frame is processing when the ElasticFusion thread checks this flag, it will save the map if the flag is set. The saving of is map into ply file is the same as in original ElasticFusion program, the map is downloaded as 3D points from the *Globalmodel* class and then each point's confidence value is checked against the *confidenceThreshold* value set before, if the point's confidence value is above the threshold then it's saved to the file. The saved ply file is supposed to be able to view on applications on the Tango tablet such as "MeshLab for Android", however no attempt is successful for unknown reason.

In order to show the speed of ElasticFusion running, a frame counter is displayed on the screen in application, every time the ELasticFusion processed a frame, the frame counter is incremented. This is done by when the screen is rendered by using the `request_render}` function, the increment function is called from the C++ part to the Java part of program using Java Native Interface (JNI), this function call in

Java then starts a new *Runnable* using the *runOnUiThread* function to increment the counter and set the text of the counter label. This function is also implemented on all other functions, including the data logging function and others that runs ElasticFusion.

6.6.2 Running ElasticFusion dataset on Tango

Other than use the live RGBD and pose data from the Tango tablet itself, for experimenting and testing purposes, the application developed is also implemented to run the `dyson_lab.klg` dataset that was used as the example in ElasticFusion paper[2], the Dyson Lab dataset however, only contains the RGBD data as it is the data recorded using pure 3D sensor camera. This means the camera tracking function that previous has been turn off by inputting the pose directly to ElasticFusion now has to be running, however the main method to implement this function is similar to the previous function, except for the data does not come from Tango device but the dataset file placed on the device manually.

To be able to read the RGBD data from dataset file, the *RawLogReader* inherited from *LiveLogReader* that is used to read data from dataset files in the *GUI* part of original ElasticFusion program is extracted out and put into the *runDatasetEF* class created for this function. When data is read from the dataset file, it's checked if it's compressed, if the RGBD data is compressed then the depth data is uncompressed using *uncompress* function in zlib library[64] into a byte buffer, and the RGB image data is uncompressed from JPEG format using OpenCV library for Android[65]. As mentioned above, since after the adaptation the format of image texture in ElasticFusion is in `GL_RGBA` format, but the RGB image data stored in JPEG format in dataset file does not have the Alpha channel, an Alpha value of 255 is manually added to the back of each "RGB" values in the byte buffer acquired after uncompress is done, also since the OpenCV library for Android[66] reads data from JPEG image in "BGR" fashion instead of "RGB", the order is corrected when the Alpha channel value is added.

The same as to run ElasticFusion from data on Tango, when the button for using dataset to run ElasticFusion is pressed, a new thread for this function is started, for each frame it first read the RGBD data from the dataset, and let the ElasticFusion process the data in a synchronise way, instead of using a circular buffer like the previous function did.

Since the dataset does not contains pose data, the pose is estimated in this function using the camera tracking process in ElasticFusion, it make use of the CUDA and SuiteSparse function that are mentioned early. In order to monitor the pose estimation when ElasticFusion is running, the current pose stored in ElasticFusion

[64] `http://zlib.net/`
[65] `http://opencv.org/platforms/android/`
[66] `http://opencv.org/platforms/android/`

is acquired using the *getCurrPose* function after frame is process, this pose is stored in Eigen structure *Eigen::Vector4f*, representing by the affine transformation matrix. This pose is then logged out to allow for monitoring, and can be viewed by using the ADB Logcat function when connected to a computer.

The same as the previous function, when running ElasticFusion using dataset on Tango tablet, the global map reconstructed can be saved to a ply file on the Tango tablet, and the live processed frame count is displayed on the screen. However, due to the time limitation in this project, no GUI is built for the application to allow the user to choose which file to run as dataset, instead a fixed file named *dataset.klg* is used as the dataset to run ElasticFusion, and error will be given if the file is not found.

6.6.3 Running Tango dataset on Tango

One of the problem found when try to run the ElasticFusion using live data from Tango tablet is that, since ElasticFusion runs very slowly on the Tango tablet, and the RGBD and pose data is acquired from the latest position in the circular buffer, the actual frame processed by ElasticFusion is not continuous, as discovered that averagely for each frame ElasticFusion is processed, seven frame has been putted into the circular buffer. Since the frame rate of Tango data is already slow on the tablet, only about 3 to 6 Hz, a gap of seven frames is already a large discontinuity in data for ElasticFusion, this results in very poor result of ElasticFusion even when the Tango tablet is held very stable and moving very slowly. One of way that may help is to extend the fixed size of circular buffer, however the uncompressed RGB image and depth data takes up a certain amount of space in memory, and the current 50 frames stored in the buffer is already a considerable space occupied in the memory, since only 4GB memory is available on the Tango tablet to share between CPU and GPU.

Therefore, for experimenting purpose the function is implemented to use the previously saved RGBD and pose data files to run the ElasticFusion program on Tango tablet, since the data is read and used in a synchronise way, the problem described above would not arise. To do this a similar class is created as in the prevoius function, so new thread can be started to read data from the RGBD and pose files on the tablet and use them to run ElasticFusion. The RGB image data is also processed to correct the order of ”RGB” value and alpha channel value is added and depth data is also uncompressed into a byte buffer. For the pose data reading, the files for class *GroundTruthOdometry* are copied from the *GUI* part of original ElasticFusion program to the Android application and adapted, it's used to read the pose data in the file stored in position and quaternion format and convert it to transformation matrix format using the *Eigen::Vector4f* structure. The read pose is then feed to the ElasticFusion program for processing. One thing to be mindful here is that the *Core* ElasticFusion should take the current pose estimation as input instead of a incremental transformation matrix with respect to the last frame. There are errors

in original Elasticfusion source code that have confused between these two, as the pose inputted to ElasticFusion can not only be used as a ground truth pose but also can be used as a estimation to bootstrap the optimization in camera tracking stage, the errors are corrected here.

Lastly, with RGBD and pose data feed into the ElasticFusion from dataset on Tango, the reconstructed global model can also be saved into a ply file on device, and the current processed frame count is displated on screen. Again, due to time limitation no GUI is built for choosing dataset file to run and fixed file names ”tangoRGBD-Data.klg” and ”tangoPosedata.klg” are used.

6.7 Combine ElasticFusion with pose from Tango in Loop closure mode

Beside to get the ElasticFusion program running on the Tango tablet, one other objective of this project is to find a way to allow the ElasticFusion program can be run in loop closure detection mode with input pose estimated but no use it to bootstrap the camera tracking process. This allows allow the estimated pose input to ElasticFusion to be used in loop closure mode when the pose estimated may not be very accurate but extra processing time and power can be saved, which is the perfect case for running ElasticFusion on Google Tango tablet, since previously for reduce processing time and increase frame rate the loop closure detection of ElasticFusion is disabled, but having this adaptation at least made it theoretically possible.

In the ElasticFusion[2] paper, When local or global loop closure happens, the predicted surface renderings of current model prediction and frame registered together is brought into alignment by a relative transformation matrix $\boldsymbol{H} \in \mathbb{SE}(3)$ outputs by the optimization problem, however the changes the current pose stored in the ElasticFusion program, and theoretically if more loop closure happens, as the pose from Tango platform does not correct drifts, the pose input to ElasticFusion will be inconsistent of the true pose of the Tango tablet.

To correct the pose input to ElasticFusion, an extra correction matrix is store in the *ElasticFusion* class using the *Eigen::Vector4f* structure. This matrix is initialised to be identity matrix, and every time a local or global loop closure happens, the relative transformation matrix $\boldsymbol{H}$ matrix calculated is multiplied to the correction matrix to accumulate the difference. And at the beginning of each frame processed, the correction matrix is multiplied to the input pose from Tango to produce the corrected pose which will be used in the 3D reconstruction process. In this way the pose feed into ElasticFusion is always consistent with the pose stored in ElasticFusion, and drifts will be corrected to prevent error in pose estimated to build up.

This change is not only implemented in the ElasticFusion program on Tango tablet but also made in the original ElasticFusion program on desktop computer, and it's tested on desktop computer using the Dyson Lab dataset. Since the Dyson Lab dataset only contains RGBD data, the dataset is used to run ElasticFusion for one time first with loop closure detection, and make use of the saved pose graph file generated when the ElasticFusion program exits. The saved pose graph file format is slightly different from the format required for input pose dataset, so the program is adapted to generated the pose graph file as a pose dataset file directly. This allows the ElasticFusion program to run using the Dyson Lab RGBD dataset and pose dataset we generated, and so the adaptation of the program can be tested, since in the Dyson Lab dataset there are many local and global loop closures so it's a perfect example to demonstrate the effect made by adaptation.

6.8 Debugging and testing

To debug and test the Android application developed in this project, the Android log library is linked to the application which allows the logcat[67] function in Android ADB (Android Debug Bridge)[68] to be used, in the code of application, different levels of message logging can be done by use the functions *LOGI* for info, *LOGW* for warning and *LOGE* for error messages. The definition of these functions using macros are shown below, the parameter of these functions can be used as normal format output for printing.

```
#define LOG_TAG "JNI Log"
#define LOGI(...) __android_log_print(ANDROID_LOG_INFO, LOG_TAG,
    ↪ __VA_ARGS__)
#define LOGW(...) __android_log_print(ANDROID_LOG_WARN, LOG_TAG,
    ↪ __VA_ARGS__)
#define LOGE(...) __android_log_print(ANDROID_LOG_ERROR, LOG_TAG,
    ↪ __VA_ARGS__)
```

To see the logged messages, the Tango tablet can be connected to computer with ADB installed using USB cable, by typing in *adb logcat* in the command line tool window, the messages will be print out if the Android application is running. Alternatively by opening Eclipse or Android Studio[69], the logcat print-outs can be easily filtered and searched which provides a better way for testing.

If the application crashed, the error and stack trace will be logged out in the logcat tool which allows the function and file where cause the error or crash to be found and analysed.

To debug the C++ code in the application, GDB[70] debugger can be used. In the original code for Advanced Estimation in Robotics (433H) course practical exercise that this application used as basic frame work, the folder `debug_scripts` contains scripts to set-up GDB. After the set-up script is run, by running the application in Eclipse using the debugging mode, the code can be debugged by pause to inspect status or adding break points.

The one kind of error that has to be explicitly checked is the OpenGL ES error, since it may not cause the application to crash but the code will just not work. To check for error in the OpenGL ES code, the function *glGetError* and *glCheckFramebufferStatus* are used to detect general OpenGL ES error and incorrect OpenGL ES framebuffer status, their function signatures are given below. These functions have to be manually written in places where the OpenGL ES to be checked.

[67] `https://developer.android.com/studio/command-line/logcat.html`
[68] `https://developer.android.com/studio/command-line/adb.html`
[69] `https://developer.android.com/studio/index.html`
[70] `https://www.gnu.org/software/gdb/`

```
GLenum glGetError(void);
GLenum glCheckFramebufferStatus(GLenum target);
```

7 Evaluation

In the original planning of this project, the evaluation process is intended to use the same mechanism as in original ElasticFusion[2] paper if the implementation process went successful in this project, as the goal of this project is to deploying the ElasticFusion algorithm on Google Tango tablet for a portable dense 3D reconstruction solution.

In the original planning, to measure the performance of the solution by quantity and quality. Both algorithm and computational performance need to be benchmarked as what is done in the ElasticFusion[2] paper.

For the algorithm, trajectory accuracy can be tested using sample datasets to provide RGBD data input on the tablet, then the output pose graph can be used to compared with the ground truth data that is obtained by using industrial standard system. The RMSE(root mean square error distance) between every estimated pose and ground truth pose can be calculated to provide the error at each time-stamp. In the 3D reconstruction part, similar method can be used to compare the reconstructed surface to benchmark data for accuracy estimation. Similarly, since the function of running ElasticFusion using saved dataset from Tango is implemented, benchmark using data from Tango tablet can also been done by comparing the result of Tango running Tango data and use the Tango data to run ElasticFusion on desktop computer.

For computational Performance, the frame rate during ElasticFusion running can be plotted to check if it is within the acceptable range for real time SLAM application.

However, before the project ends the experiment of using Dyson Lab dataset to run ElasticFusion on Tango tablet have not achieve good results. Firstly for reason that still not to be known, somehow the confidence value of surfels are extremely low compared to the default confidence threshold 10.0, which results in no surfel is saved when the global map reconstructed should be saved to a ply file. After this is discovered the very low confidence threshold 0.45 is set to try to allow some surfels from the global model to be saved to the ply file, but by viewing the saved the ply file in MeshLab[71] application on computer, it can be seen that the 3D reconstructed surface is slightly wrong compare to the reconstructed model by running ElasticFusion on desktop computer. However due to time limitation on this project, the reason for that or the place caused this error is still unknown. Secondly, from all the experiments to run ElasticFusion on Tango tablet, it can be seen that even when all the settings are set to allow ElasticFusion run at fastest speed, such as disabling $\mathfrak{so}(3)$ pre-alignment, use single level tracking pyramid by enabling fast odometry, and disable loop closure detection. Despite that, even when running the Dyson Lab dataset with its manually generated pose data on Tango tablet, the frame rate is very slow

[71] http://www.meshlab.net/

that the processing of each frame takes up more then a second, which corresponds to the time seven frames are newly stored in the circular buffer. This is actually already faster then using data from the Tango tablet since the resolution of Dyson Lab dataset image is 640 and 480, but the Tango tablet camera resolution is 1280 and 720, which means using Tango data will require more processing time and memory usage. As the Tango tablet only has 4Gb memory to used by both CPU and GPU, the memory is not sufficient for 3D reconstruct large scenes, and in the implementation stage the `TEXTURE_DIMENSION` value that represents dimension of reconstructed scene in *GlobalModel* class has to be reduced from the original 3072 to less 1000. Therefore the performance of running ElasticFusion on Tango is no where near the frame rate required for real time interactive SLAM application (30 Hz).
The slow performance of running ElaticFusion on Tango tablet is mainly due to the lack of processing power on the tablet, as described below, the Nvidia Tegra K1 processor is still quite far behind from the processing power required although it's already quite power for mobile devices, since ElasticFusion is a dense 3D reconstruction SLAM system, both memory and processor usage is intensive.

On the other hand, although first recording the RGBD and pose data using Tango tablet and then use the data for running ElasticFusion on desktop is possible, only a small part surface is able to be re-constructed and the performance is not very good compared to other RGBD sensor cameras like Microsoft Kinect or ASUS Xtion Pro Live or Intel RealSense 3D Camera. This is due to the IR projective depth sensor is not very good on the Tango tablet, the depth points captured is very sparse and results in performance of ElasticFusion not as good as using other depth sensors.

8 Conclusions and Future Work

After this project, it is concluded that currently the ElasticFusion algorithm may not be able to run on the Tango Yellowstone tablet for real time interactive application due to problems like processor, memory and sensors. However due to the time limitation and effort that can be put on this project, no optimization is done and no extension have been developed in this project.

If more time and effort is given, optimization could be done such as to use CUDA for pre-processing the RGBD data from data or adapt ElasticFusion algorithm for higher noise depth data. On the application side, extra features can be added such as displaying the 3D reconstructed surfel map on the tablet screen in real time, add interactive function to allow for drag and zoom operation, showing the camera's current pose on the map. Further more, demo application and be built to utilising the augmented reality ability on Tango platform, such as placing a virtual static object on the 3D reconstructing surface. Or building GIU to allow user choose what file to be used as dataset to run ElasticFusion.

Although great result has not been achieved in this project, it have still shown the possibility of using state-of-the-art SLAM system on mobile device for dense 3D reconstruction. It have also shown the possible way to use Tango tablet as a pure 3D data recording device and then upload the recorded data to a desktop computer for 3D reconstruction.

On the other hand since this project has only attempted to deploy the ElasticFusion on the Google YellowStone Tango tablet, most of the problem arises actually is only related to the tablet itself. Since this tablet is the first officially release Tango device and has been stopped supporting by Google for quite some time, the hardware and software on the tablet is out of date. With the newly released ASUS ZenFone AR[72] Tango smart phone, much better depth sensor and software is carried onboard. This means that ElasticFusion may very well achieve better performance on the new Tango device. However all other Tango device carries Qualcomm processor and the ASUS ZenFone AR phone carries Qualcomm Snapdragon 821 processor, this means that if the camera tracking process will be used then the part that's implemented using CUDA has to be written. But the new Tango platform on new Tango device have already improved the accuracy of the trajectory estimated by performing drift corrections and also the Tango constructor tool[73] has been released, which means the usage of ElasticFusion algorithm on Tango device will be better in the future.

Overall, this project has demonstrate the possibility of using a commodity mobile hand held device for 3D dense surface reconstruction, and have shown the potential of developing augmented reality using on top of it, for example the new feature released by Tango to use environment lighting information to shading virtual objects

[72] https://www.asus.com/us/Phone/ZenFone-AR-ZS571KL/

[73] https://play.google.com/store/apps/details?id=com.projecttango.constructor

in real time.

9 appendix

9.1 planned project timeline

The original planned project timeline is shown in Fig 33 with each task corresponding to a stage above.

10 User Guide

After the application starts running, the screen will show an interface similar to what is shown in Fig **??**, from the picture we can see that the screen will display the live RGB image captured by the color camera, also a few buttons and labels are shown on the Screen. The buttons on the screen are used for: log Tango data to files on tablet, run ElasticFusion use live data on Tango, run ElasticFusion use the Dyson Lab dataset on tablet, run ElasticFusion use saved Tango data without pose, run ElasticFusion use saved Tango data with pose, save the current reconstructed model to the ply file on tablet. The rightmost label is used to display the current processed or logged number of frames. Any function can be start by press the corresponding button once and stop by press it again, however if another function is running an error will pop up to alert the user. To close the application simply exit normally.

11 Bibliography

References

[1] Google. Google project tango, 2017. pages 2, 27, 30

[2] Thomas Whelan, Renato F Salas-Moreno, Ben Glocker, Andrew J Davison, and Stefan Leutenegger. Elasticfusion: Real-time dense slam and light source estimation. *The International Journal of Robotics Research*, 2016. pages 2, 11, 12, 15, 48, 50, 53, 56

[3] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, José Neira, Ian Reid, and John J Leonard. Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Transactions on Robotics*, 32(6):1309–1332, 2016. pages 9, 10

[4] Anastasios I Mourikis and Stergios I Roumeliotis. A multi-state constraint kalman filter for vision-aided inertial navigation. In *Robotics and automation, 2007 IEEE international conference on*, pages 3565–3572. IEEE, 2007. pages 11, 27, 31

[5] Richard A Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *Mixed and augmented reality (ISMAR), 2011 10th IEEE international symposium on*, pages 127–136. IEEE, 2011. pages 11, 12, 13, 15, 16, 20

[6] Yang Chen and Gérard Medioni. Object modelling by registration of multiple range images. *Image and vision computing*, 10(3):145–155, 1992. pages 12

[7] Paul J Besl and Neil D McKay. Method for registration of 3-d shapes. In *Robotics-DL tentative*, pages 586–606. International Society for Optics and Photonics, 1992. pages 12

[8] Fuxing Yin. *Volumetric Representation on KinecFusion*. csdn, 2016. pages 13

[9] Stefan Leutenegger, Simon Lynen, Michael Bosse, Roland Siegwart, and Paul Furgale. Keyframe-based visual–inertial odometry using nonlinear optimization. *The International Journal of Robotics Research*, 34(3):314–334, 2015. pages 14, 15, 16

[10] Maik Keller, Damien Lefloch, Martin Lambers, Shahram Izadi, Tim Weyrich, and Andreas Kolb. Real-time 3d reconstruction in dynamic scenes using point-based fusion. In *3DTV-Conference, 2013 International Conference on*, pages 1–8. IEEE, 2013. pages 16

[11] Ben Glocker, Jamie Shotton, Antonio Criminisi, and Shahram Izadi. Real-time rgb-d camera relocalization via randomized ferns for keyframe encoding. *IEEE transactions on visualization and computer graphics*, 21(5):571–583, 2015. pages 17

[12] Richard A Newcombe, Dieter Fox, and Steven M Seitz. Dynamicfusion: Reconstruction and tracking of non-rigid scenes in real-time. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 343–352, 2015. pages 17

[13] Matthew Klingensmith, Ivan Dryanovski, Siddhartha Srinivasa, and Jizhong Xiao. Chisel: Real time large scale 3d reconstruction onboard a mobile device using spatially hashed signed distance fields. In *Robotics: Science and Systems*, 2015. pages 19

[14] Matthias Nießner, Michael Zollhöfer, Shahram Izadi, and Marc Stamminger. Real-time 3d reconstruction at scale using voxel hashing. *ACM Transactions on Graphics (TOG)*, 32(6):169, 2013. pages 19, 20

[15] O. Kahler, V. A. Prisacariu, C. Y. Ren, X. Sun, P. H. S Torr, and D. W. Murray. Very High Frame Rate Volumetric Integration of Depth Images on Mobile Device. *IEEE Transactions on Visualization and Computer Graphics (Proceedings International Symposium on Mixed and Augmented Reality 2015*, 22(11), 2015. pages 20

[16] Raul Mur-Artal and Juan D Tardos. Orb-slam2: an open-source slam system for monocular, stereo and rgb-d cameras. *arXiv preprint arXiv:1610.06475*, 2016. pages 21

[17] Dorian Gálvez-López and Juan D Tardos. Bags of binary words for fast place recognition in image sequences. *IEEE Transactions on Robotics*, 28(5):1188–1197, 2012. pages 22

[18] Jakob Engel, Thomas Schöps, and Daniel Cremers. Lsd-slam: Large-scale direct monocular slam. In *European Conference on Computer Vision*, pages 834–849. Springer, 2014. pages 23

[19] Thomas Schöps, Jakob Engel, and Daniel Cremers. Semi-dense visual odometry for ar on a smartphone. In *Mixed and Augmented Reality (ISMAR), 2014 IEEE International Symposium on*, pages 145–150. IEEE, 2014. pages 23

[20] Zhenfei Yang and Shaojie Shen. Monocular visual–inertial state estimation with online initialization and camera–imu extrinsic calibration. *IEEE Transactions on Automation Science and Engineering*, 14(1):39–51, 2017. pages 25

[21] Sameer Agarwal, Keir Mierle, and Others. Ceres solver. `http://ceres-solver.org`. pages 26

[22] Dimitrios G Kottas, Joel A Hesch, Sean L Bowman, and Stergios I Roumeliotis. On the consistency of vision-aided inertial navigation. In *Experimental Robotics*, pages 303–317. Springer, 2013. pages 27

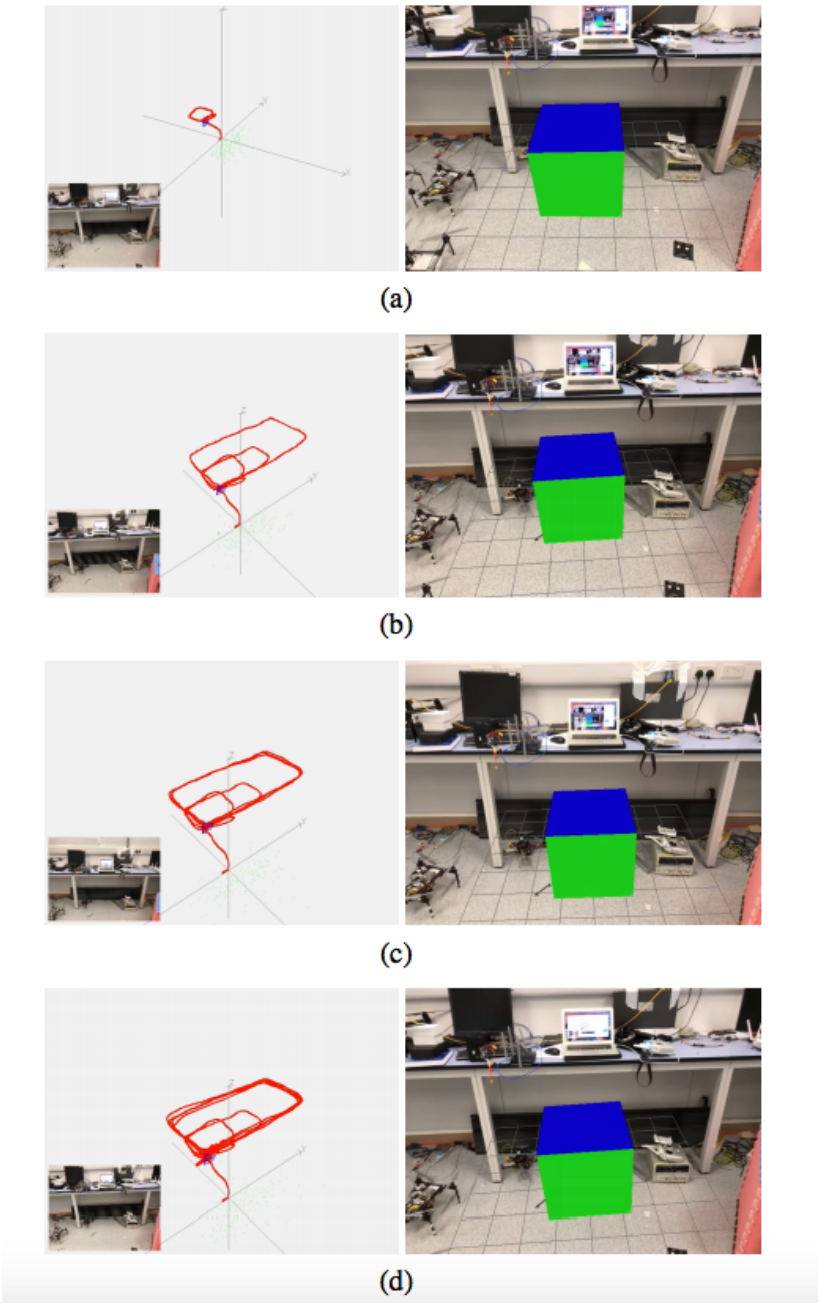

Figure 23: Trajectory estimated with virtual objects in VINS-Mobile

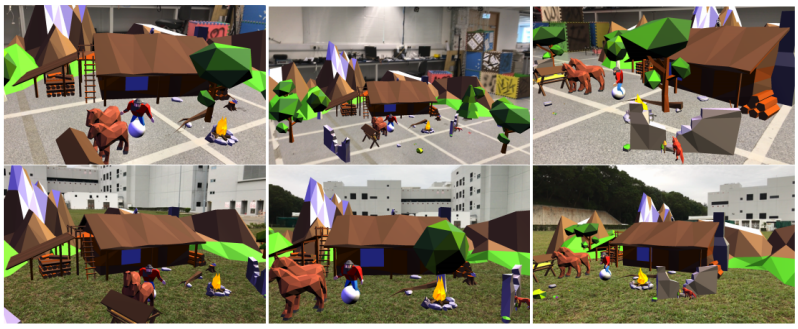

Figure 24: VINS-Mobile AR application

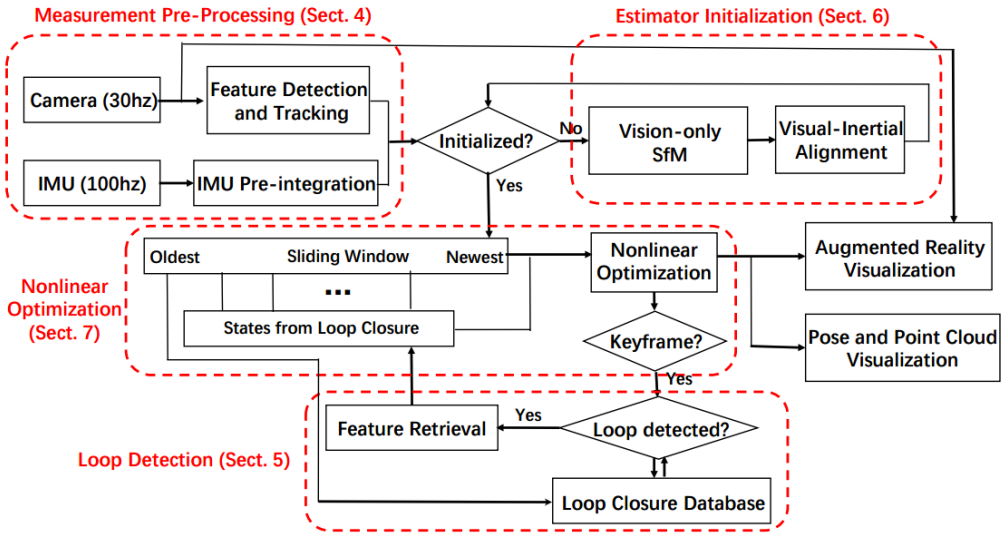

Figure 25: VINS-Mobile system structure

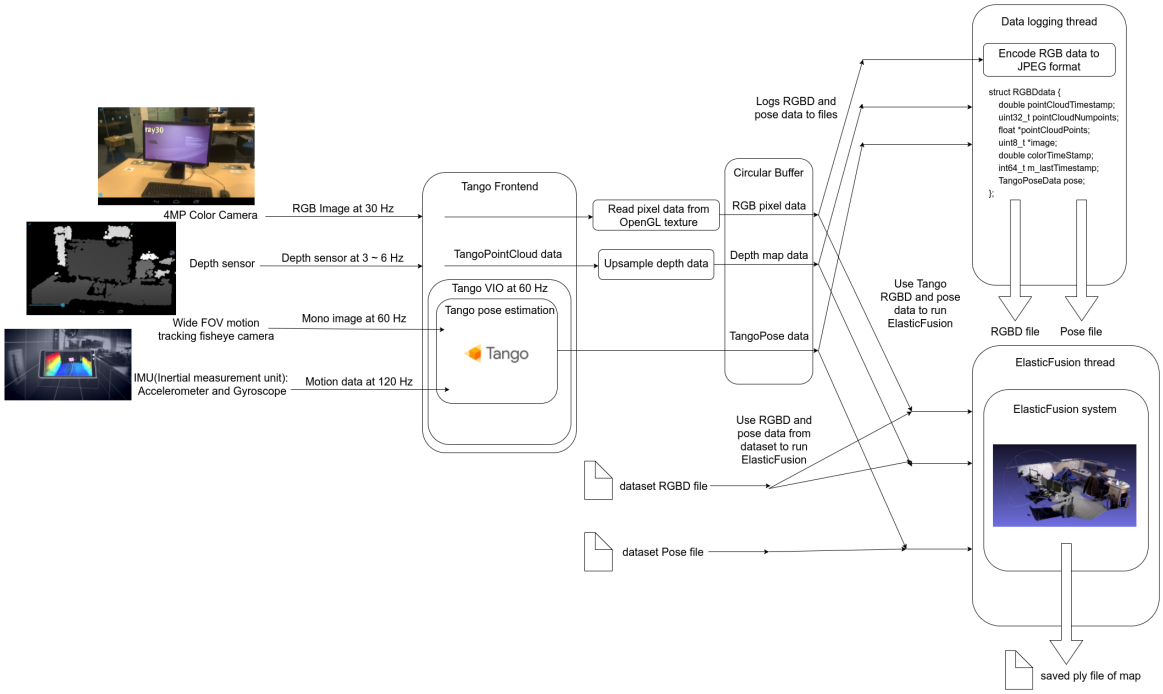

Figure 26: projectStructure

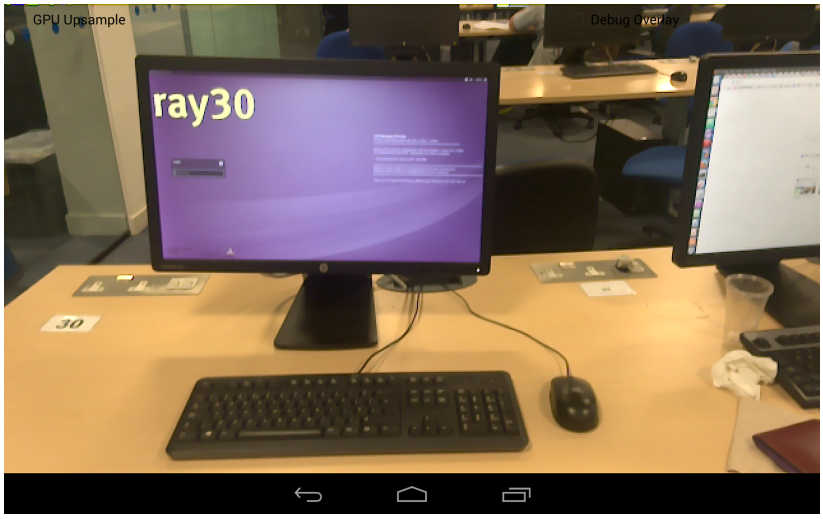

Figure 27: RGB Depth Sync Demo application 1: pure RGB image

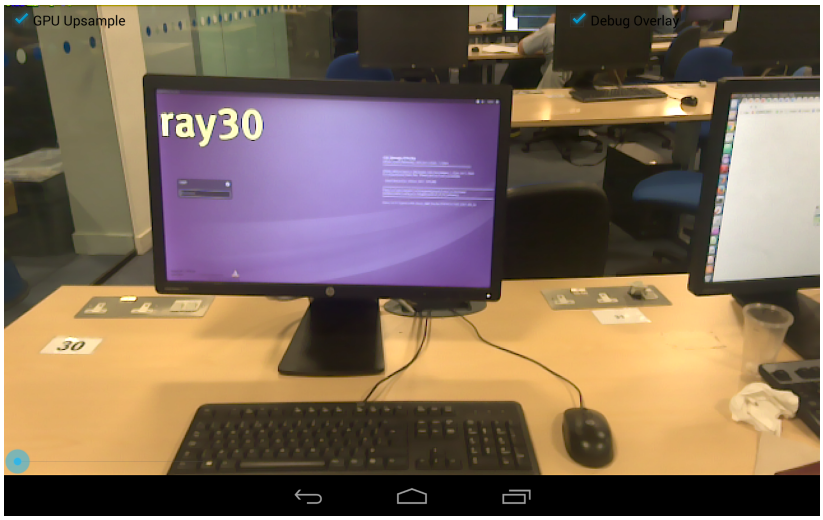

Figure 28: RGB Depth Sync Demo application 2: pure RGB image

Figure 29: RGB Depth Sync Demo application 3: pure RGB image

Figure 30: RGB Depth Sync Demo application 4: mixed RGB and depth

Figure 31: RGB Depth Sync Demo application 5: mixed RGB and depth

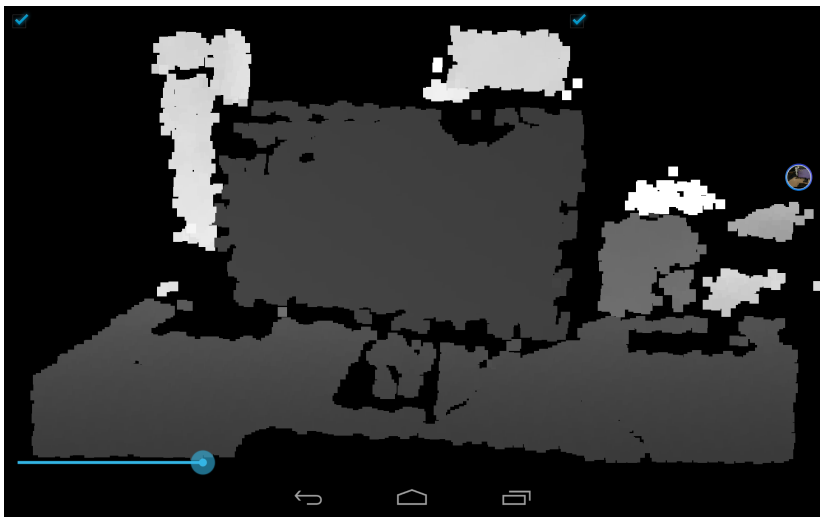

Figure 32: RGB Depth Sync Demo application 6: pure depth map

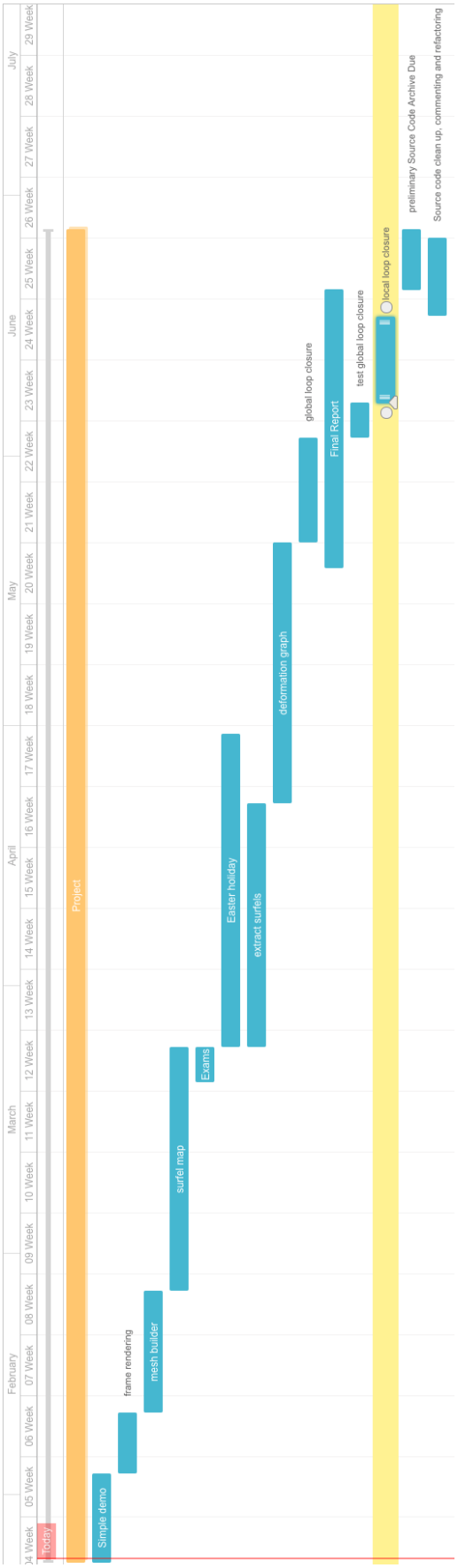

Figure 33: Graph for project timeline

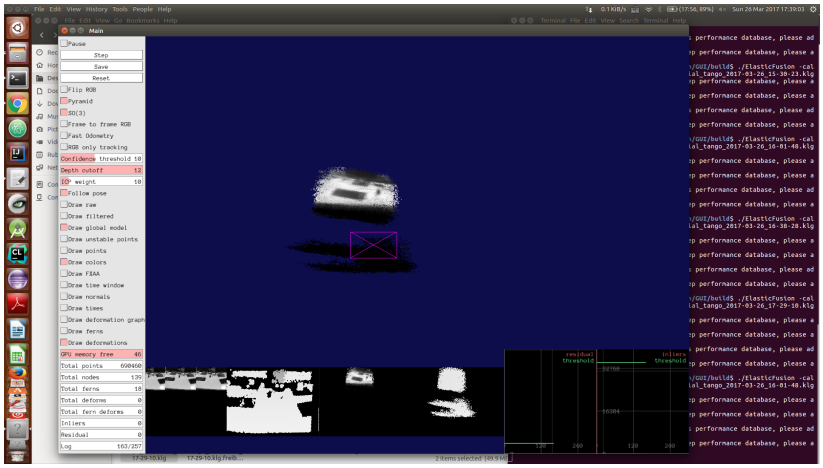

Figure 34: Running ElasticFusion using saved Tango data 1

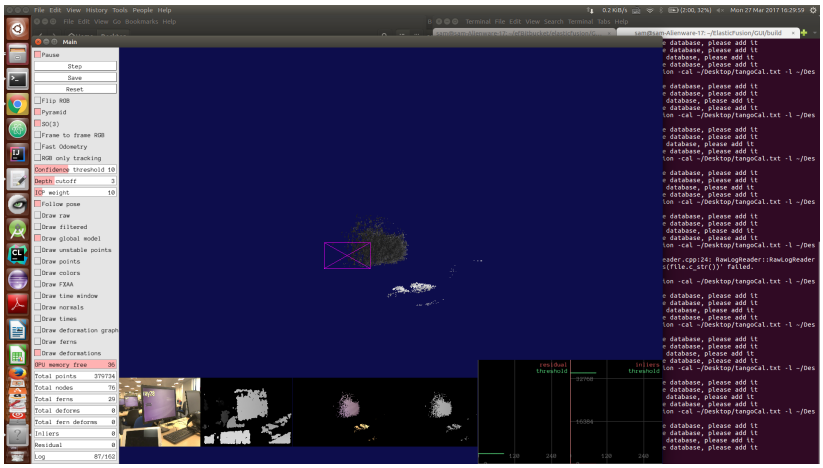

Figure 35: Running ElasticFusion using saved Tango data 2

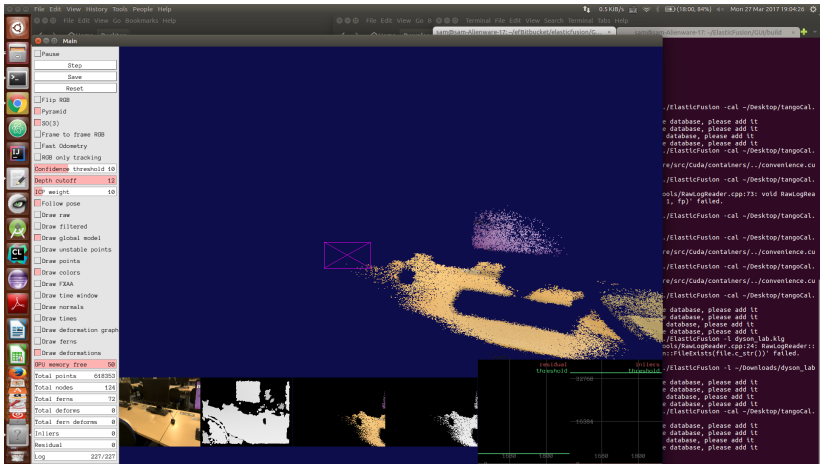

Figure 36: Running ElasticFusion using saved Tango data 3

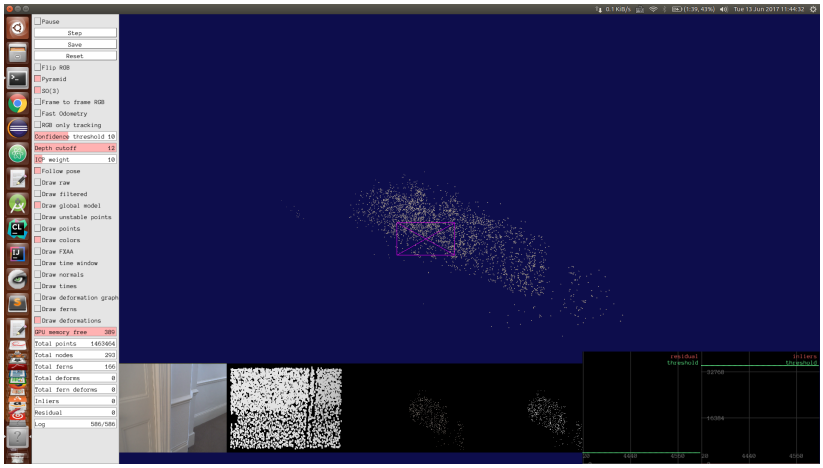

Figure 37: Running ElasticFusion using saved Tango data 4

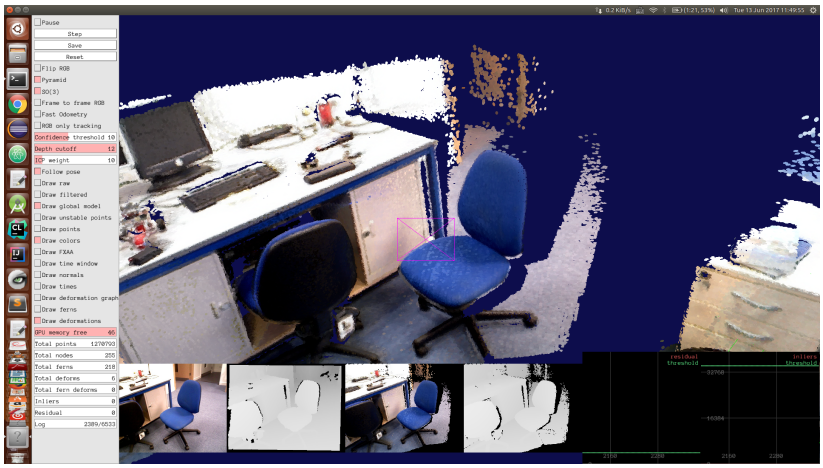

Figure 38: Running ElasticFusion using Dyson Lab dataset

Figure 39: Running ElasticFusion using Dyson Lab dataset

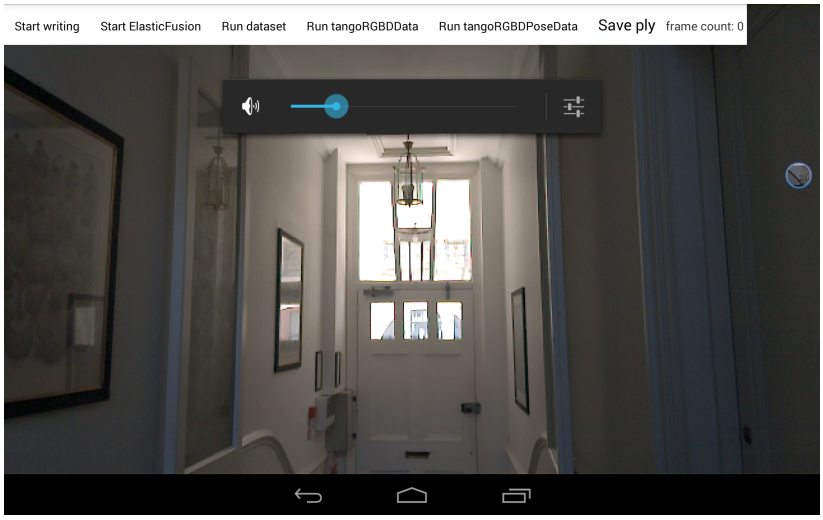

Figure 40: Screenshot of Application on Tango