Tag SLAM: Marker Based Mapping for Large Scale Augmented Reality

Oisin Feely

A dissertation submitted for the degree of

HDip in Computer Science

Department of Computer Science

Faculty of Science and Engineering

National University of Ireland Maynooth

September 2018.

Head of Department: Dr. Adam Winstanley

Supervisor: Dr. John McDonald

Acknowledgements

Firstly I would like to thank my supervisor Dr. John McDonald without whom this project would not have been completed.

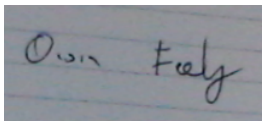
I like to thank Louis Gallagher for putting up with a litany of questions and providing answers and ideas throughout the entirety of the project.

Finally I would like to thank Jane Daly for the constant encouragement and helping me stay on track.

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study as part of the HDip in Computer Science qualification, is entirely my own work and has not been taken from the work of others - save and to the extent that such work has been cited and acknowledged within the text of my work.

SIGNED: OISIN FEELY DATE: 28/03/2018



Contents

Abstract

The objective of this project is to combine a state-of-the-art dense visual mapping system with off-the-shelf 2D tag based augmented reality systems to create a combined approach for computing large scale 2D tag based maps of an environment. Dense visual mapping systems operate by taking as input a video stream from a Microsoft Kinect type camera, and produce as output a rich photo-realistic 3D model of the scene. Such systems provide the basis for next generation high-end augmented and virtual reality. At the other end of the spectrum many current mobile AR systems operate by detecting a single known 2D bar-code or image in the environment and then overlay 3D content relative to that tag. The advantage of this latter approach is that it is particularly suited to low-powered and computationally limited mobile platforms. The disadvantage is that with such systems the "scale" of the AR experience is limited to the scale of a single tag. The system developed in this project overcomes this limitation by using the dense mapping system to compute a large scale map of the placement of tags within an environment which can then be compactly encoded for use within a mobile AR application. To achieve the above we combine ElasticFusion, a state-of-the-art open source dense SLAM system, with ARToolkit, a freely available commercial tag based AR system. We demonstrate the capability of the combined platform through a number of real-world tests involving multiple tags distributed across table and room-scale setups. We firstly present results showing the resulting large scale tag maps, and then demonstrate the efficacy of the approach for large scale multitag AR.

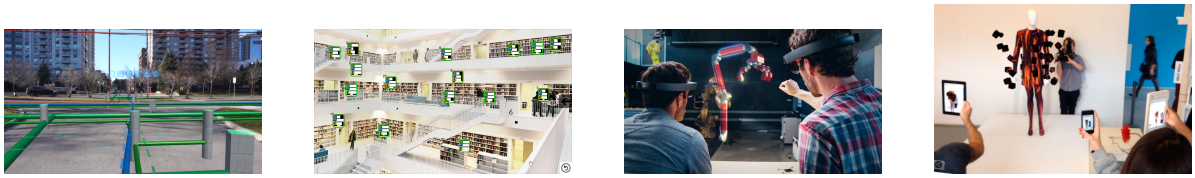
Chapter 1

Introduction

1.1 Augmented Reality and ARToolKit

The aim of this project is to develop a computationally efficient large scale AR (augmented reality) system by combining the strengths of both 2D marker based AR and 3D dense SLAM (simultaneous localization and mapping) systems. The resultant system should have the advantage of allowing the development of large scale collaborative (multi-user) AR projects without the need for the intensive processing associated with the localization and mapping of SLAM required for computing the pose (position and orientation) of markers. In collaborative AR projects the pose of markers must be predetermined by measurement and in some cases this is impractical or indeed impossible in large scale environments. There are solutions to this localization and mapping problem but they often require more processing power than is available to the average user on their device. These solutions also require tight integration of AR software with the localization and mapping software which can be difficult to implement and overall quite complex depending on the mode of implementation. This project will separate the intensive processing required of localization and mapping and output marker positions allowing the implementation of AR applications which can be run on mobile devices with relatively low computational power.

Augmented reality is the overlay of computer generated images over a user's view of the world which can be obtained in a variety of ways. This is opposed to virtual reality which is a computer generated user view. Augmented reality is finding more and more applications as the techniques for generating augmented reality become more advanced and as users gain more access to the technology for augmented reality, not just mobile devices but smart glasses and others. Examples of AR in use can be seen in Fig. 1. This just illustrates some of the uses of AR and there are many more applications for this type of technology especially as more of the limitations surrounding this technology, such as the difficulty of large scale AR, are overcome. Many companies see the value in AR and are investing in it such as Microsoft with HoloLens, Daqri with Smart Glasses in the workplace, and others focusing on outdoor collaborative projects.



(a) (b) (c) (d)

Figure 1: Geolocation based AR (a), library based AR application (b), workplace AR using HoloLens (c), and wearable AR (d).

The AR library used in this project is ARToolKit [1]. ARToolKit is a library for the creation of strong augmented reality applications and was first publicly demonstrated in 1999 at the computer graphics event SIGGRAPH. It was acquired by Daqri in 2016 and was released open source under a LGPL licence.

AR can be split into roughly two categories, marker based and markerless AR. Markerless AR relies on camera sensor data, both RGB and depth if available, to accurately position generated images over the user's view but this requires more complex algorithms and more processing power on the user's end device. In comparison ARToolKit uses tracking markers, Fig. 2a and Fig. 2b, to provide a point of reference for overlaying a computer generated image on the user's view. This ensures the overlaid image is at the correct

scale, orientation, and position relative to the user's view. Using markers is computationally less expensive due to the square nature of the markers and their predetermined size allows the straightforward computation of distance and orientation. A simple application of this can be seen in Fig. 2c and Fig. 2d. In this example the ARToolKit library detects the square marker, computes the pose of the marker relative to the camera, and overlays the square cube so that it is at the correct scale and pose relative to the user. While this is one of the most simple examples quite complex applications can be achieved through the use of multimarkers.

Figure 2: ARToolKit Hiro (a) and Kanji markers (b). Simple AR application with raw image of the Hiro pattern (c) and generated AR cube on marker (d).

1.2 Multimarkers

In ARToolKit multimarker refers to the use of multiple markers that have an arbitrary but fixed relationship to each other, an example of a multimarker can be seen in Fig. 3a. This allows for improved stability as well as tracking. As the relationship of markers to each other is known markers can be obscured and AR images still be drawn, this also allows AR images to be drawn in between markers or move between markers. Due to this more complex AR applications can be developed that are not feasible with single markers. An example application is geolocation within buildings, if the position of markers in relation to each other are known users can scan the markers and get precise directions. This could be applied to museums or a variety of public and private buildings. With the growing popularity of AR there are many applications possible using multimarkers in the public and private sector.

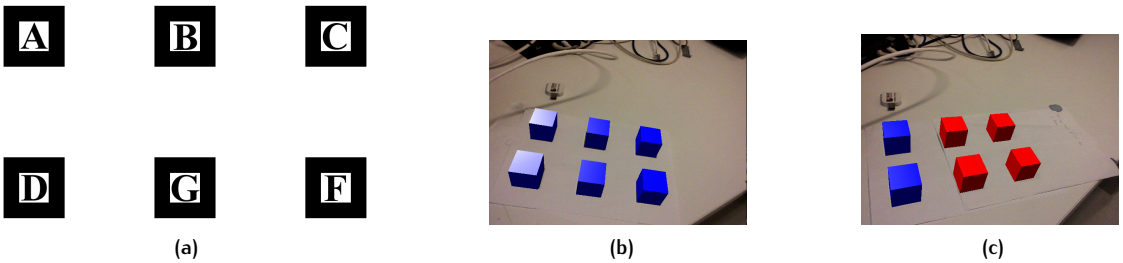


Figure 3: Multimarker comprising of 6 patterns (a) and multimarker pattern in use (b), note how cubes are still drawn even though some markers patterns are obscured by a sheet of paper (obscured markers denoted by red cube drawn on them).

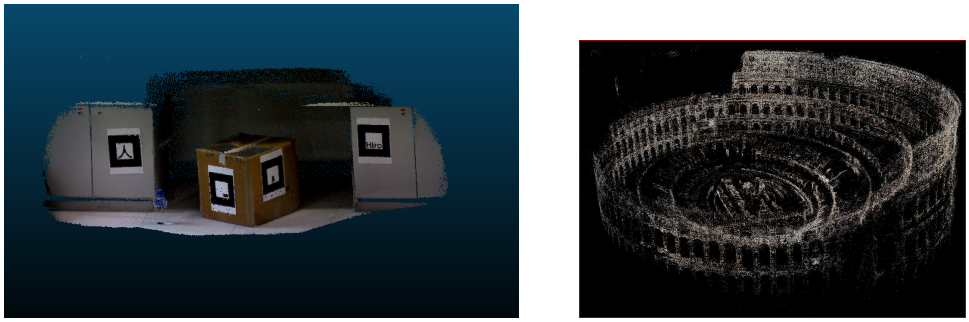
The current issue with multimarker AR is that the distance and orientation between markers must be physically measured by hand and as such the most common arrangement is to have all markers on a single flat sheet aligned at 90 degree angles to each other as shown in Fig 3a. Depending on the positioning of markers it is either impractical or impossible to measure the pose of markers relative to each as such there needs to be a system in place to compute pose of markers relative to each other.

1.3 Visual SLAM

Simultaneous localization and mapping (SLAM) refers to a fundamental problem in robotics where a robot is required to construct a map of an unknown environment while keeping track of its location in that environment. Solving this problem allows for long term robot autonomy [2]. This problem has been studied

for a number of decades with robust solutions now available through various open source libraries. These solutions rely on variety of methods such as infra-red, lidar, sonar, and RGB-D (colour and depth) for the construction of a map of an unknown environment and localization of the robot within that environment. One type of SLAM is visual dense SLAM that relies on RGB-D data to construct a map of an environment. This is what is used in this project as to localize markers within an unknown environment they must first be identified and to do so requires visual data.

Dense SLAM refers to dense scene reconstructions over more sparse point-based reconstructions [3]. Dense in this context refers to dense point cloud reconstruction. A comparison between dense and sparse scene reconstruction can be seen in Fig. 4. As substantially more information about the scene is captured with RGB-D data it is both possible to identify markers in the captured RGB data and to evaluate any estimation of their position by using the detailed dense scene reconstruction. In this project EF (Elastic Fusion), a visual dense SLAM approach developed by Thomas Whelan, will be used.



(a) (b)

Figure 4: Dense scene reconstruction of an environment with AR markers present (a), note the greater detail including colour, and a less detailed sparse scene reconstruction of the Colosseum (b).

1.4 Proposed Approach

The proposed approach for this project will be to use EF to map an unknown environment and to localize the mapping camera's position in the environment from captured RGB-D data with the camera pose in the environment as an output. The ARToolKit libraries will be used to detect markers in the RGB data and estimate the detected markers' poses relative to the camera with the detected markers' poses being the output of this step. By the relation between the detected markers' poses and the camera poses the poses of the markers in the environment can be computed. The poses of the markers in the environment will then be outputted in formats suitable for evaluation and use with a multimarker AR application. Fig. 5 presents a simple visualization of the approach of this project.

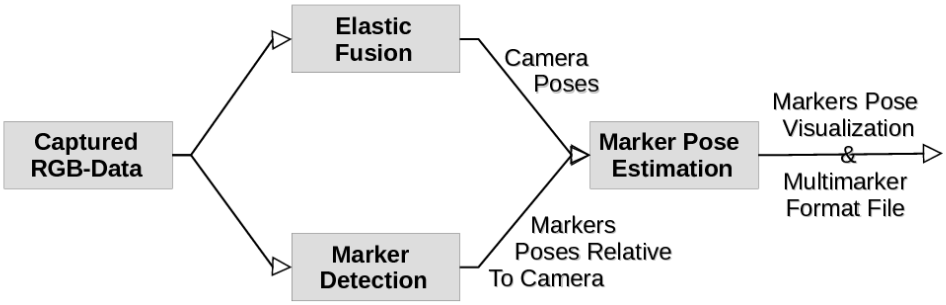


Figure 5: Proposed pipeline for estimating marker poses in unknown environment.

In Section 2 we will go through the background to Elastic Fusion and SLAM that will be used in this project and the processes involved in identifying and computing the poses of markers. In Section 3 the implementation of the programs and algorithms needed to address the problem will be covered. Section 4 will show how the results of this implementation were evaluated and Section 5 will summarise the implications of this work and discuss future work that can be done.

Chapter 2

Technical Background

Creating the proposed AR system for this project requires 3 elements.

1. A map of the environment,
2. An ability to compute the pose of the camera relative to that map and finally,
3. Using 1. and 2. a method for projecting or overlaying virtual content into the user's field of view.

In this chapter we will describe how using EF and marker based AR systems such as ARToolKit can provide for each of these elements as well as the mathematical details behind computing pose first starting with how to characterise poses.

2.1 Characterising Camera and Marker Poses

Pose refers to the position and orientation of an object in a coordinate system. It is essential for this project to be able to characterise the pose of the camera and of the markers detected in a convenient way. A common way of representing pose is in Euclidean space using the Cartesian coordinate system. While this is useful in many cases using Cartesian coordinates in computer vision makes it difficult to model camera projection i.e. the mapping of 3D points in the world to 2D points in an image. Instead a more typical approach in computer vision is to use homogeneous coordinates and projective geometry. Using homogeneous coordinates provides more convenient methods for the manipulation of points, vectors, and planes in computer vision and graphics.

2.1.1 Projective Geometry and Space

Projective space can be viewed as an extension of Euclidean space [4]. The main reason for the use of projective space in computer vision and graphics is that using Euclidean space can lead to one major problem when lines run in parallel. This is the example that Hartley and Zisserman use [4]. While these lines in the real world never meet when they are mapped from 3D to 2D they are then said to meet at infinity. The use of infinity in computer vision and graphics algorithms can be extremely troublesome to implement. This difficulty can be resolved by adding these points at infinity were parallel lines meet. By doing this Euclidean space is transformed into projective space and homogeneous coordinates can be used.

2.1.2 Homogeneous Coordinates

A 3D vector $P = ai+bj+ck$ where i, j, and k are unit vectors can be represented in projective space as a homogeneous coordinate P where $P = (x,y,z,w)$ [5] and where $a=\frac{x}{w}$,$b=\frac{y}{w}$, and $c=\frac{z}{w}$. By this representation in homogeneous coordinates it is possible to say that vectors are equivalent if they only differ by a non zero-scale factor i.e. $(x,y,z,1)=(2x,2y,2z,2)$ or for any vector (wx,wy,wz,w). Doing so provides a more convenient method for projecting points in space. Here the above example point P is represented as a 4x1 vector as shown in Fig. 6a.

In reality an object has both position and orientation i.e. a pose. A common approach to represent pose is to attach a coordinate frame to the object. This coordinate is described by a 4x4 matrix in homogeneous form as shown in Fig. 6b which shows both the coordinate frame attached and the 4x4 homogeneous matrix

representation. The structure of the matrix can be understood as shown in Fig. 6c. Here the upper left 3x3 sub-matrix represents the orientation of the object (rotation matrix *R*) whereas the position of the origin of the object frame corresponds to the rightmost column (translation vector *T* and scale factor). By attaching a frame and generating a 4x4 matrix in homogeneous form it is now possible to rotate and translate a 3D point through matrix multiplication. This matrix effectively maps the 4x1 vector representation of the point relative to a global frame into a 4x1 vector representation in the object's frame.

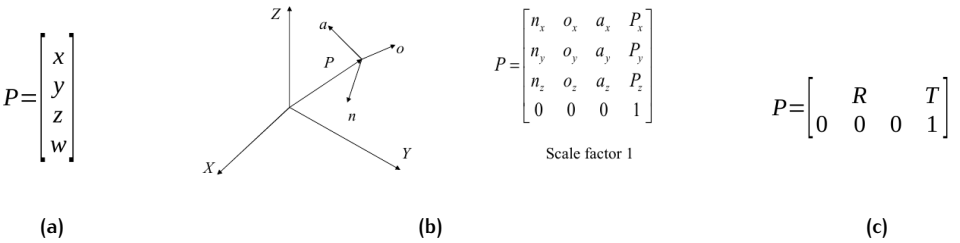


Figure 6: Homogeneous point (a), 4x4 homogeneous transformation matrix representing orientation and position of point P, and equivalent transformation matrix (c) with R and T representing the 3x3 rotation matrix and translation vector respectively.

2.1.3 Pose Estimation with Homogeneous Coordinates

The pose of an object can be described by a 4x4 homogeneous matrix as shown above. An object can be transformed (rotated and translated) through matrix multiplication. From Szleliski [5] an important property of a 4x4 homogeneous matrix for this project is that multiplying itself by its inverse produces the identity matrix. This is a square matrix with ones on the main diagonal and zeroes elsewhere. This matrix can be said to represent an object at the origin point of a coordinate system and with an orientation in-plane along the x, y, z axes of a coordinate system.

This property can be used to estimate the pose of objects relative to each other in a global coordinate system. Multiplying one object's pose P_n by the inverse of another P_{n+1}^{-1} gives the pose of P_n relative to P_{n+1}. Furthermore by choosing one object to be the origin, P_1, and multiplying itself by its inverse, P_1^{-1}, to place it at the origin. It is then possible to multiply all other objects, $P_{1\ldots n}$, by P_1^{-1} to get the pose of each object relative to P_1 and all objects are then in the coordinate system of P_1 with P_1 being the identity matrix and the origin.

If given multiple measurements of a pose of an object it is also possible to find the average position of the object simply by adding up all x's, y's, z's i.e. translation vector *T* and dividing by total number of measured poses. While this is extremely useful if given multiple measurements of the one object it is not possible to get the average orientation of the object this way. However a possible approach here is to instead use an alternative representation of rotation which we describe in the following section.

2.1.4 Quaternions

Quaternions are a number system that extends complex numbers, invented by Irish mathematician William Rowan Hamilton in 1843 [6]. Quaternions find particular application in calculations involving three-dimensional rotations in the fields of computer vision and graphics as an alternative to rotation matrices and Euler's angles.

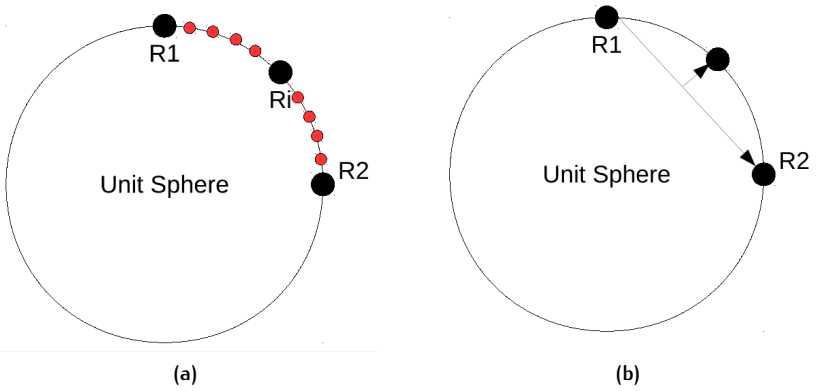
By using quaternions problems associated with Euler's angles such as gimbal lock can be avoided [7]. As quaternions describe a 4 vector in space of the form $q_0 + iq_1 + jq_2 + kq_3 = 0$ they have an advantage over 3x3 rotation matrices as they are first smaller, 4 vs 9 scalars, and second quaternion multiplication is much faster than matrix multiplication. Overall using quaternions provides a straightforward method of describing rotation about an arbitrary axis.

One distinct advantage of quaternions is they simplify methods for describing motion where one orientation is just a modification of the previous orientation. This has particular application in computer graphics for smoothly animating sequences of arbitrary rotations. As the orientation of a detected marker from frame to frame can be regarded of as a sequence of arbitrary rotations this application of quaternions gains particular importance in the scope of this project. Spherical Linear Interpolation (SLERP) is one method for animating

rotations with quaternions which we will detail in the following section and explain how it may be used in computing the average orientation of a detected marker.

2.1.5 SLERP

SLERP is a method introduced by Kevin Shoemake for animating rotation with quaternions [7]. His paper presents a method using quaternions on the unit sphere to create a spline curve on that sphere suitable for smoothly interpolating between arbitrary rotations. As shown in Fig. 7a by interpolating a midpoint rotation *Ri* along the curve of the unit sphere animation occurs smoothly along curve of the sphere as shown by the red dots and avoids shaky animation caused by interpolating a midpoint directly between *R1* and *R2* and then calculating the midpoint rotation as in Fig. 7b.



In this project the use for SLERP is to find the average orientation of an object over a sequence of captured frames. Rather then averaging the objects rotation matrices over time step δ which due to the nature of data may contain errors and outliers it is preferable to use SLERP to interpolate smoothly between each arbitrary orientation recorded, thus getting rid of any possible deviations, and then averaging by δ. By doing so a more accurate average is computed as SLERP minimizes deviations that would affect the average.

2.1.6 Pose Summary

From this section we now have a way of characterising both camera poses from EF and detected markers' poses in a convenient way. Furthermore due to the nature of this characterisation it is now possible to transform detected markers' poses into EF's coordinate system through matrix multiplication, average markers' poses through homogeneous translation vectors and quaternions and finally compute detected markers' poses relative to each other through matrix multiplication again. Before this is done though we must first look at the processes involved in EF and ARToolKit to get camera and marker poses.

2.2 EF SLAM

2.2.1 EF SLAM Introducton

Elastic Fusion (EF) is a real time visual dense SLAM system developed Thomas Whelan [8]. This system is capable of capturing consistent surfel-based scenes providing a dense scene representation of the environment. Surfel refers to surface element where each surfel can be considered as an infinitesimal surface patch with both position and orientation, a scene can be represent by a dense set of surfel points. The reason for using EF is due to its camera pose estimation and dense scene reconstruction. As explained by Whelan methods do exist to compute high quality maps from RGB-D data but are unable to operate over very large scales [3]. He also explains that in dense vision systems the number of points measured in each frame to generate surfels is much higher then in feature based systems. Due to this the camera pose has to be held fixed during the mapping step and the resulting scans aligned in post processing. For the purpose of marker pose estimation it is much more feasible to have a moving camera for mapping the scene in which the markers are present.

The strengths of the EF approach in this context are

1. That it provides a robust algorithm for computing a dense surfel based model and camera pose.

2. It subdivides the map into active and inactive regions thereby allowing large scale mapping.
3. It re-fuses new data with previously mapped but possibly inactive regions.
4. It provides a global loop closure mechanism allowing the creation of large scale maps

By doing so EF produces a more accurate environment model and camera pose. EF can be seen processing a log in Fig. 7

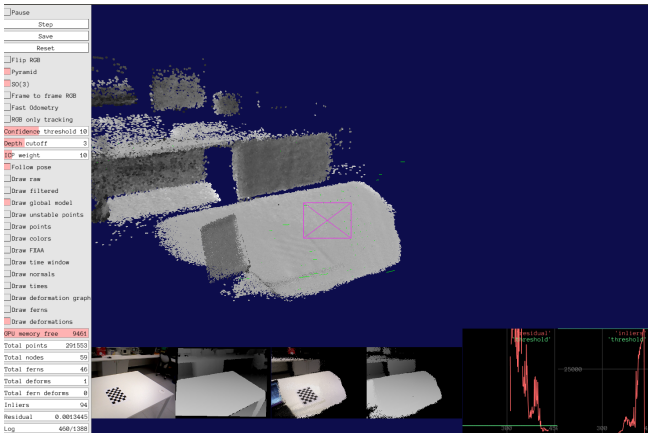


Figure 7: EF processing a log, the main screen is the dense scene reconstruction. The small screens (left to right) show the colour image input, the depth input, the predicted colour image, the predicted depth image, and the pose graph confidence threshold.

2.2.2 Using EF for Camera Pose Estimation

EF processes RGB-D data in the form of log (.klg) files captured via an RGB-D camera, for this project an ASUS Xtion Pro Live 2 was used, with Logger2 [9]. Logger2 was also developed by Thomas Whelan. EF outputs a polygon format file (.ply) that can be viewed with point cloud and mesh visualization software such as MeshLab or CloudCompare, an example of this visualization can be seen in Chapter 1 in Fig. 4a.

Camera pose estimation in EF is done through photometric and geometric frame to model rendering. Frame to model rendering refers to the processing of RGB-D data in each frame captured and rendering the environment model from that information. Photometric rendering is based on the differences in light intensity and colour to render the model while geometric is based on the difference in points to render the model.

For both photometric and geometric frame to model rendering the aim of EF is to find the motion parameters ϵ that minimize the cost over the photometric error and the cost of point to point plane error between the current frame and the predicted model (geometric and photometric) from the last frame. By doing so the photometric and geometric frame to model rendering in EF produce two maps of the environment and two current pose estimates that can then be fused together for a more accurate map and pose estimate. EF minimises the joint cost of the photometric and geometric frame to model render through optimisation for an overall more accurate model and pose. More detail on this process can be found in Whelan's paper [8]. The source code for EF is freely available through GitHub [10] as is Logger2 for capturing RGB-D logs [9].

2.2.3 Camera Pose Usage

Using Elastic Fusion it is now possible to obtain an accurate map of a scanned environment which can be used to evaluate computed markers poses. More importantly an accurate estimation of camera pose at each frame from RGB-D data can be obtained. By having an accurate camera pose of each frame it is now possible, if a marker is detected in that frame, to estimate the pose of the marker relative to the camera and through matrix multiplication, as mentioned in the first section of this chapter, to translate the marker pose into the global coordinate system of the map outputted by EF. In the next section we will cover the processes involved in detecting markers and markers' poses with ARToolKit.

2.3 ARToolKit

The ARToolKit library's primary purpose is to provide a framework for the development of marker based AR applications. It achieves this through detecting markers in a given frame and estimating the poses of the detected markers. From the estimated poses it is possible to generate and overlay images over the markers. The generation of these images is the final stage in the ARToolKit pipeline. This is done through GLUT (Open GL Utility Toolkit) [11] and can be seen as a graphics based procedure. We can separate these stages by providing an overview of the ARToolKit pipeline.

2.3.1 ARToolKit Image Processing Pipeline

The ARToolKit image processing pipeline can be summarized in the following steps

1. A frame is converted to a black and white binary image.
2. The frame is then analysed for a marker.
3. If a marker is detected the pose of the marker relative to the camera is computed.
4. The symbol identifying the marker is then matched against markers in memory.
5. The virtual objects to be generated are aligned with marker using the marker's pose.
6. Finally the virtual objects are rendered in the frame.

This pipeline can be seen in Fig. 8. For the purpose of estimating the markers' position relative to each other and generating a multimarker configuration file it is the first 4 steps in the pipeline that are of importance. To understand how ARToolKit achieves this we must first look at the ARToolKit architecture.

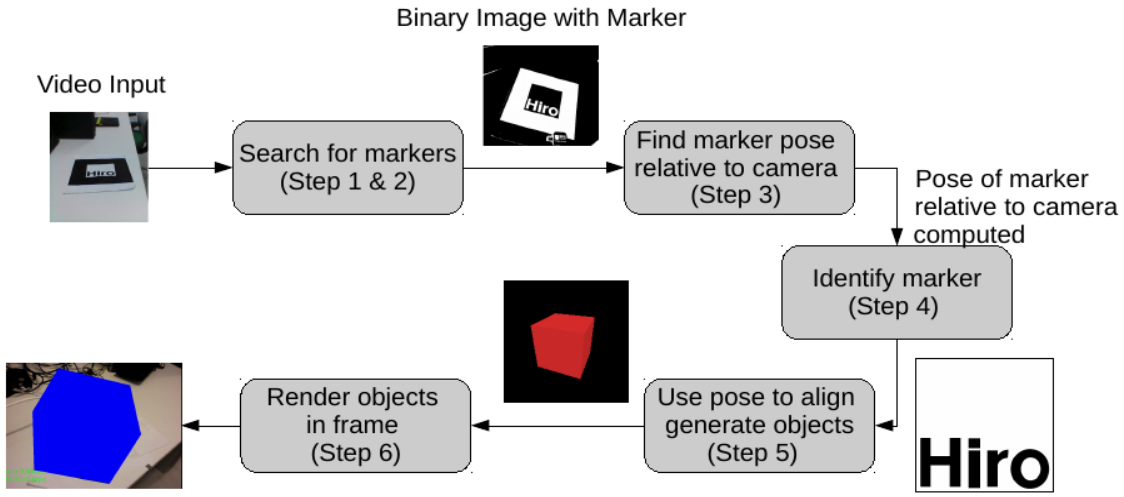


Figure 8: Diagram illustrating the ARToolKit pipeline.

2.3.2 ARToolKit Architecture

The ARToolKit architecture for creating AR applications can be summarized in Fig. 9. As shown in the diagram an AR application uses ARToolKit to interact with underlying libraries. ARToolKit uses OpenGL for rendering, GLUT to handle window/handler events and hardware dependent video libraries and standard API for different platforms [1].

The ARToolKit library itself is broken down into 4 modules. The core AR module which with marker tracking, calibration and parameter collection is done. A video module for capturing video frames from different inputs and on different operating systems and finally a graphics module for the interacting with GLUT and OpenGL.

For the purpose computing marker pose and generating multimarker configuration files it is only the core AR module that is of concern. In the next section we will look at markers and multimarkers in more detail to understand what it is that the core AR module deals with.

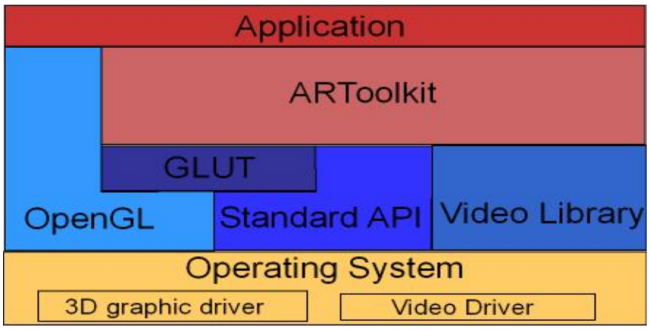


Figure 9: ARToolKit application framework architecture. Source https://www.hitl.washington.edu/artoolkit/documentation/devframework.htm

2.3.3 ARToolKit Markers

Black square markers containing a pattern are how ARToolKit superimposes virtual imagery on a live scene as shown in Fig. 2 in the first chapter. These markers must be square, have a continuous border and the pattern inside the border must be rotationally asymmetric. The size of the markers is arbitrary but it is generally preferable to have larger markers as this gives increased range in marker detection because the markers then occupy more pixels in the camera's view.

An issue with using single markers is occlusion, virtual imagery will only appear when the marker is in view, if the marker is somehow obscured this prevents the imagery being generated. An example of this can be seen in Fig. 10 This is one reason why using multimarkers is preferable and can be seen by comparing Fig. 10, whereby the cube is not generated due to marker pattern being obscured, to Fig. 3c where cubes are still generated even though certain patterns on the multimarker are obscured by a sheet of paper.

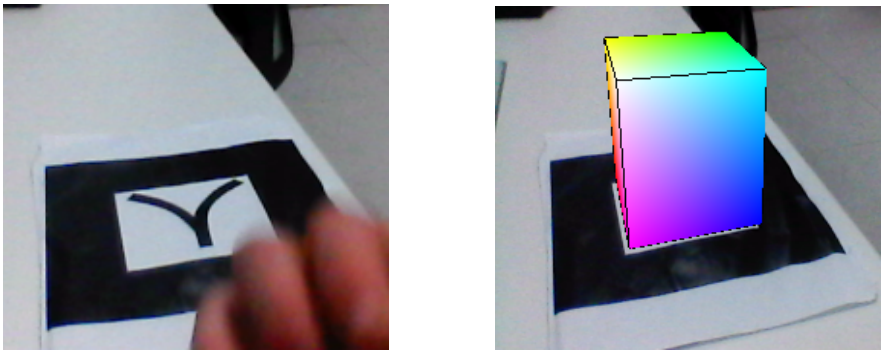


Figure 10: Hiro marker slightly obscured by hand (a) so that marker cannot be detected and image generated as in (b).

New marker patterns can be designed by editing the marker template provide with ARToolKit. Generating a marker pattern (.patt) file for use with ARToolKit is done using the utility programe mk_patt. In ARToolKit it is also possible to generate pattern files based on images i.e. a picture could be taken of some feature which can be then trained as a pattern. This is known as NFT (Natural Feature Tracking). The process for doing this is very similar to the process for generating a pattern file for a standard square marker.

As mentioned there are limitations on single marker applications and that is why using multimarker patterns is preferable for large scale complex AR applications. To understand how ARToolKit applications work with multimarkers we must look at the multimarker configuration file.

2.3.4 Multimarkers and Multimarker Configuration Files

Multimarkers are a type of marker that comprises of multiple markers with an arbitrary but fixed relationship to each other as shown in Fig. 3a in Chapter 1. As these multiple markers are considered as one marker this allows for a variety of applications that are not possible with single markers. As mentioned this allows for objects to be drawn in between markers as well as drawing objects that overlay multiple markers allowing for larger AR applications. It is also possible to use a detected marker that comprises a multimarker set to direct the user to other markers in the multimarker set if precise localized geolocation was needed. There are

a variety of uses for multimarkers and it is through the multimarker configuration file that the relationship between markers comprising the multimarker set is defined and how ARToolKit computes this relationship.

The multimarker configuration file (.dat) defines a relationship between an origin marker and the pose of all other markers in the set relative to that origin marker. The maximium amount of markers in a multimarker set that ARToolKit provides as an example is 48. The benefits of using multimarkers are improved robustness, if one marker is partially obscured another may be visible and the scene can still be rendered as shown in figures in Chapter 1. Marker pose estimation is also improved as markers' poses relative to each other are known from the multimarker configuration file. Finally ARToolKit can improve rejection of mis-read marker poses.

Multimarker configuration files are generated in the following format. The first line in the file specifies the number of markers to be read from the file. Then for each marker the first line out of five is the path to the marker pattern file relative to the multimarker file. The second line specifies the size of the marker in millimetres. The last three lines are the first three rows of a standard homogeneous coordinate transformation matrix.

An example of a multimarker configuration file is shown in Fig. 11 and the corresponding multimarker can be seen in Fig. 3a in Chapter 1. As explained above the size of each marker is specified, in this instance at 40 mms, and the path to the .patt files for each marker in the set is specified. The first marker of the multimarker set is at the origin i.e. it is the identity matrix. Every other marker in the set is in plane with the origin marker and therefore the rotation matrix for each remains the same. The translation vector specifies the distance each marker is from the origin marker, in this example it is only the x or y components of the translation vector that differ as the markers are on the same page and thus the z component is always the same.

```
#the number of patterns to be recognized
6

#marker 1
a.patt
40.0
1.0000 0.0000 0.0000 0.0000
0.0000 1.0000 0.0000 0.0000
0.0000 0.0000 1.0000 0.0000

#marker 2
b.patt
40.0
1.0000 0.0000 0.0000  100.0000
0.0000 1.0000 0.0000   0.0000
0.0000 0.0000 1.0000   0.0000

#marker 3
c.patt
40.0
1.0000 0.0000 0.0000  200.0000
0.0000 1.0000 0.0000   0.0000
0.0000 0.0000 1.0000   0.0000

#marker 4
d.patt
40.0
1.0000 0.0000 0.0000   0.0000
0.0000 1.0000 0.0000 -100.0000
0.0000 0.0000 1.0000   0.0000

#marker 5
g.patt
40.0
1.0000 0.0000 0.0000  100.0000
0.0000 1.0000 0.0000 -100.0000
0.0000 0.0000 1.0000   0.0000

#marker 6
f.patt
40.0
1.0000 0.0000 0.0000  200.0000
0.0000 1.0000 0.0000 -100.0000
0.0000 0.0000 1.0000   0.0000
```

Figure 11: Multimarker and corresponding multimarker config file data.

2.3.5 ARToolKit Source Code

ARToolKit can be downloaded as prebuilt binaries or cloned from GitHub [12] and built using make. ARToolKit is able to capture image data from a variety of sources such as webcam, video, or images using its video library. Specifying the image source is done through the video configuration settings environment variable. Image data must be in RGB or similar format, ARToolKit is unable to process RBG-D data which is why the module of interest in the context of of this project is the core AR module.

2.4 Summary

As outlined at the start of this chapter the creation of the proposed AR system for this project requires 3 elements. In this chapter we have shown that the an accurate map of an environment can be computed by EF, that the ability to compute the pose of the camera relative to that map can be done through the use of homogeneous coordinates, and that finally ARToolkit provides a method for using the first 2 elements for projecting or overlaying virtual content into the user field of view.

The next chapter will deal with the design and implementation of algorithms that implement these 3 elements and outputs a multimarker configuration file for use with a multimarker AR application and a visualization of markers within the map of the environment for evaluation.

Chapter 3

Design and Implementation

3.1 Design and Implementation Overview

In this chapter we present the approach developed to combine EF with ARToolKit. A detailed diagram of this approach can be seen in Fig. 12. This approach can be summarized as follows.

1. Capture logs of RGB-D data using Logger2 via an ASUS Xtion Pro Live 2 RGB-D camera. These logs will comprise of unknown environments containing multiple markers.
2. Processing logs with EF to ouput
   - Maps of scanned environments in .ply format.
   - Camera pose at each frame of captured logs.
3. Processing logs with program inputImage to detect markers and output detected markers' poses relative to camera, frames markers detected in, and detected markers' IDs.
4. Parse data from inputImage and EF text ouputs with program markerPosition to compute average markers' positions in the map and output
   - Multimarker configuration file for use with a multimarker application.
   - .ply file visualizing the markers' poses in the map coordinate system.
5. Evaluate estimated marker positions by overlaying environment map and marker position .ply files with MeshLab.
6. Use multimarker configuration file to create a multimarker AR application.

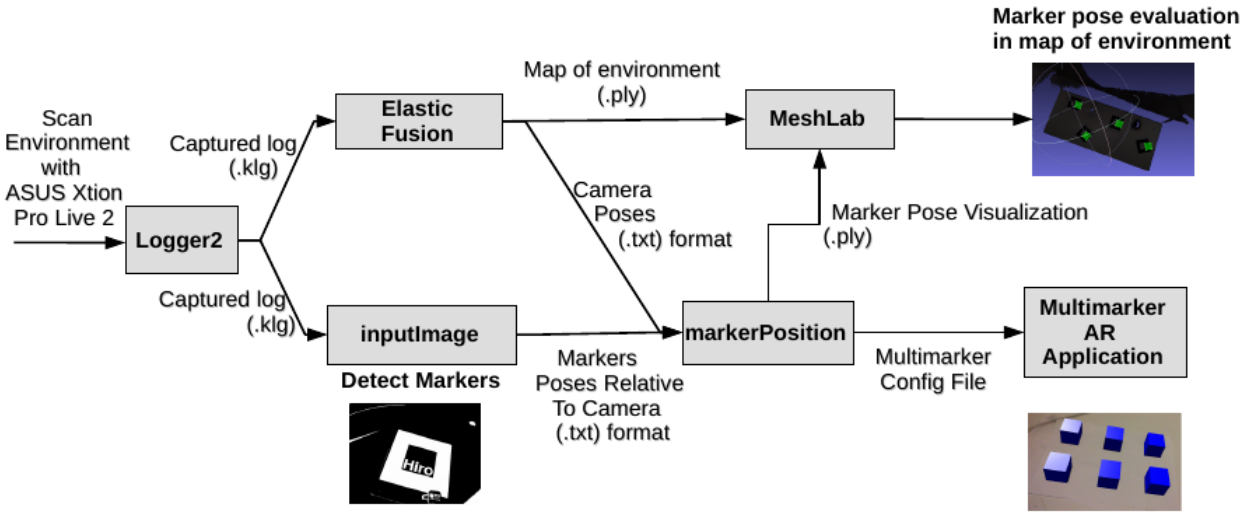


Figure 12: Detailed project pipeline.

A once off process for the computation of the ASUS Xtion Pro Live 2 camera parameters for use with the ARToolKit library for optimal marker detection was also implemented. This process involved

1. Capturing a log of a checkboard at various angles,
2. Outputting frames of this log from EF in JPEG format,
3. Processing these frames with ARToolKit utility program calib_camera for computation of camera parameters.

The rest of this chapter will detail the steps of this pipeline and how the combination of EF and ARToolKit was achieved.

3.2 Capturing RGB-D Data

Capturing RGB-D data for use with EF and inputImage was done with Logger2. Logs were captured of environments containing multiple markers. A once off log was also captured of a checkboard at various angles for use computing camera parameters.

In this project pipeline it was important to capture multiple logs of the same configuration of markers as the global loop closure in EF is prone to errors and can lead to a deformed and useless map of the environment. This also causes the camera poses to be incorrect. Another reason for capturing multiple logs of the same configuration of markers is that while EF may work correctly it is not uncommon for marker detection in inputImage to fail due to lighting or marker angle and for correct computation of markers' poses in the captured environment the markers must be detected in a relatively large number of frames to compute an accurate average marker pose.

Once logs are captured they can then be used by EF and ARToolKit. We will first detail how EF was modified to fit the purpose of this project.

3.3 EF

Overall minimal changes to EF were needed. Its main purpose for this project was to output camera poses at each captured frame and a .ply map of the scanned environment. As EF already outputs a .ply map of the environment when it completes processing a log file the only modification to EF was an addition to make it output the current frame and pose at that frame to a text file. An example this can be seen in Fig. 13. Rows (r0, r1, etc.) were added to facilitate parsing later on.

```
Frame 2
r0 0.999959   0.00709186   0.00561581   0.00564609
r1 -0.00705422 0.999953    -0.00670091   0.0042975
r2 -0.00566307 0.00666102   0.999962   -0.00182566
r3 0       0        0        1

Frame 3
r0 0.999959   0.00709203 0.00561634 0.00564572
r1 -0.00705445 0.999953  -0.00669971 0.00429647
r2 -0.00566359 0.00665982 0.999962  -0.00182642
r3 0       0       0       1
```

Figure 13: EF camera pose text output detailing pose of camera in frame 2 and frame 3.

An additional once off ouput was of frames from the calibration log in JPEG format for use with the ARToolKit utility program to compute camera parameters. Before moving onto the implementation of inputImage to detect markers in captured logs the camera parameters of the RGB-D camera used in this project needed to computed.

3.4 RGB-D Camera Calibration

ARToolKit provides an application (calib_camera) for outputting camera parameter files. As mentioned above frames for calibration were outputted from EF in JPEG format. A short Python script (pythonVCONF.py) was

written to output a terminal command for changing the ARToolKit video configuration settings environment variable. The ARToolKit camera calibration application was then ran with the JPEG frames as input and outputted a camera parameter file.

3.5 Marker Detection

3.5.1 Reading RGB-D Data

As the ARToolKit video module is unable to read RGB-D data the RGB data needed to be separated from captured logs. This was achieved through the use of log reader classes provided by Louis Gallagher. These classes were compiled separately as a C++ library that were then added as a dependency to the marker detection program make file.

3.5.2 Detecting Markers

For detecting markers and outputting detected markers' poses the program inputImage was compiled with a make file specifying dependencies on the ARToolKit core AR module and the log reader library. This program comprises of multiple steps listed as follows

1. Parameter set up includes specifying pixel size of frames to be processed (640x480) and specifying camera parameters from the computed RGB-D camera parameter file and encapsulating this information in an AR struct for use when processing frames.
2. Set up of marker information of markers to be detected which involves
   - Specifying file path to marker pattern (.patt) files.
   - Attaching individual pattern information to same AR struct containing camera parameters. Up to 50 individual marker patterns can be attached to one AR struct.
   - Storing individual pattern IDs for marker identification when a marker is detected
3. Separation of RGB data from RGB-D data was done through the implementation of log reader inherited from the log reader library dependency and formatting the RGB data for marker detection.
4. Iterating through each formatted frame and detecting if a marker was present in a frame with an ARToolKit method.
5. If a marker is detected the particular marker pattern was then identified by the information contained in the AR struct detailed in previous steps.
6. If the identification of the marker passes a confidence threshold the transformation matrix of the marker from the camera i.e. the pose of the marker relative to the camera, is then retrieved using another ARToolKit method.
7. The pose of the marker relative to the camera is then outputted along with the ID of the marker pattern identified and the frame the marker is identified in.

This is the process that inputImage follows for detecting markers and outputting a text file of markers' poses, marker IDs and the frame marker is detected in. The final text output also includes the number of patterns detected and file paths to those markers at the beginning of the file. An example fo this output can be seen in Fig. 14

3.6 Computing Arbitrary Marker Poses

The third program implemented (markerPosition) takes as an input the outputted text files from both EF and inputImage, computes the markers' positions relative to each other, and outputs both a multimarker configuration file and a polygon format file for evaluation.

```
NumPatterns 4
PN Data/hiro.patt
PN Data/kanji.patt
PN Data/sample1.patt
PN Data/sample2.patt
Frame 24
Patt 0
r0  0.786120  0.604811  0.127353 -191.555744
r1  0.598403 -0.693200 -0.401731 -91.825314
r2 -0.154691  0.392017 -0.906859  1202.946555
```

(a)

```
Frame 120
Patt 1
r0 0.972762 -0.059891  0.223934 -4.250163
r1 0.035515 -0.916133 -0.399297 -243.870050
r2 0.229067  0.396374 -0.889053  1439.095377

Frame 120
Patt 2
r0 0.917633  0.331906  0.218605 266.719399
r1 0.390473 -0.855394 -0.340341 201.641854
r2 0.074032  0.397668 -0.914538 1297.916331
```

(b)

Figure 14: Start of inputImage text ouput specifying number of patterns, file paths to patterns and first frame marker was detected in and ID of that marker(a). (b) describes frame 120 where two markers were detected

3.6.1 Parsing Camera and Marker Pose Text Files

A method was implemented to parse camera pose and marker pose text files. A flag was used to differentiate between each file. As shown previously rows were specified for each homogeneous pose. Two vectors of different type structs were used to store the parsed information. For each marker pose the marker pattern id, frame detected and marker pose were contained in a struct and added to a vector of marker poses. The number of markers to be detected and marker pattern file paths were also parsed from inputImage text output and stored.

For each camera pose only the frame detected and camera pose were contained in a struct and added to a vector for camera poses. Before adding the camera pose struct to the vector the x, y, and z positions were multiplied by 1000 as EF computes pose in metres while the ARToolKit libraries computes pose in millimetres.

3.6.2 Marker Pose Average

Once the marker and camera poses were encapsulated in vectors it was then possible to compute both the markers average position and orientation. For this a method was implemented that took a marker pattern id as an argument. For each marker pattern id passed in the method iterated through the vector of marker poses, if the marker pattern id matched one in the vector of marker poses the vector of camera poses was then iterated through. If the current marker's frame matched a camera pose frame this meant the camera pose at that point was the pose where the marker was detected and pose computed.

ARToolKit computes the marker's pose with the camera being the origin. To place the marker's pose in the EF (global) coordinate system the marker's pose must be transformed by the camera pose i.e. matrix multiplication. The two matrices were multiplied and the product is pose of the marker in the maps coordinate system.

As stated in section 2 simply adding up all poses and averaging is not suitable for computing pose. To do so each marker pose in the global coordinate system was separated into position and orientation at this point in the form of a quaternion and translation vector.

For computing the average position it was sufficient to add up all the translation vectors and divide by the total amount frames that marker was detected in. The total amount of frames was obtained through a counter incrementing each time a marker pose was transformed into the map coordinate system.

For computing the average orientation of a marker each time a marker was split into a quaternion and translation vector the quaternion was stored in a vector. A method was then implemented to iterate through the vector of quaternions. The total orientation was computed by SLERPing between the current total orientation and the next quaternion in the vector. The method then returned the total orientation. The average orientation was obtained by dividing the total rotation by the total number of frames the marker was detected in.

The final average was the computed by converting the average rotation and position into a single 4x4 homogeneous matrix. For each marker the average transformation matrix was added to a vector.

Col. 1: Origin	0	100	100	0
Col. 2: X axis unit distance	0	0	100	100
Col. 3: X & Y axes unit distance	0	0	0	0
Col. 4: Y axis unit distance	1	1	1	0

Figure 15: Transformation matrix to obtain marker vertices from marker average.

3.6.3 Marker Pose Evaluation Output

For basic quantitative evaluation a polygon format file (.ply) was needed to evaluate against the .ply file outputted by EF. This was done by multiplying the marker average matrix by another homogeneous matrix Fig. 15 to get the vertices of the marker. This matrix translated the single point into 4 points at unit distance on the x and y planes. This matrix was added to a vector. A .ply file was outputted by iterating through the vector containing matrices representing the 4 vertices of a marker and specifying vertices and faces of each marker. This file could then be viewed using CloudCompare or Meshlab for basic quantitative evaluation.

3.6.4 MultiMarker Configuration File

Finally a multimarker configuration file (.dat) was outputted. This file specified the number of markers and then for each marker specified the marker file path from parsed data and the marker size. The pose of each marker was computed by iterating through the vector of marker averages and multiplying each matrix by the inverse of the first marker in the vector. This transformation computed the position all markers relative to the first marker which is the origin i.e. identity matrix, as detailed in Chapter 2.

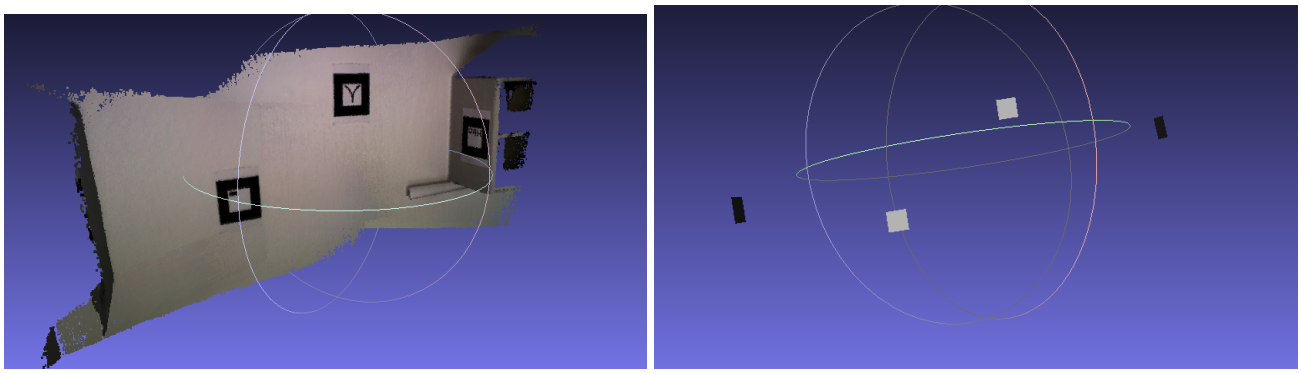
Chapter 4

Evaluation and Results

4.1 Evaluating Marker Poses

4.1.1 Evaluating Polygon Format Files

A basic quantitative method of evaluating estimated markers' poses is through the polygon format files outputted as part of the marker pose estimation process. These files contain the estimated markers' vertices and can be viewed with point cloud and mesh viewing software such as MeshLab or CloudCompare. An implementation of this method of evaluation is to overlay the estimated markers' positions .ply file over the outputted .ply map of the captured environment from EF. While this does not provide an accurate measure of evaluation it does provide a visualization of the marker pose estimation compared to the actual markers' poses. An example of the marker pose .ply and EF .ply can be seen in Fig. 16a and Fig. 16b respectively. Both of these .ply files are from the same log.



(a) (b)

Figure 16: .ply output from EF (a) and .ply output of markers from marker pose estimation (b).

4.1.2 Evaluating Multimarker Configuration Files

Accurately evaluating the multimarker configuration files outputted as the end product of the marker pose estimation process is a more difficult process. A basic evaluation method is to use the multimarker configuration file in a multimarker application, cover some markers so that the application has to overlay an image based on the data from the multimarker configuration file and observe the results of this. Again this does not provide an accurate evaluation of the marker pose estimation only that it is within an acceptable range.

4.1.3 Further Evaluation

For accurate quantitative evaluation more in depth tests would need be to be devised to accurately measure the error margin of the marker pose estimation. This evaluation could be done in a variety of ways.

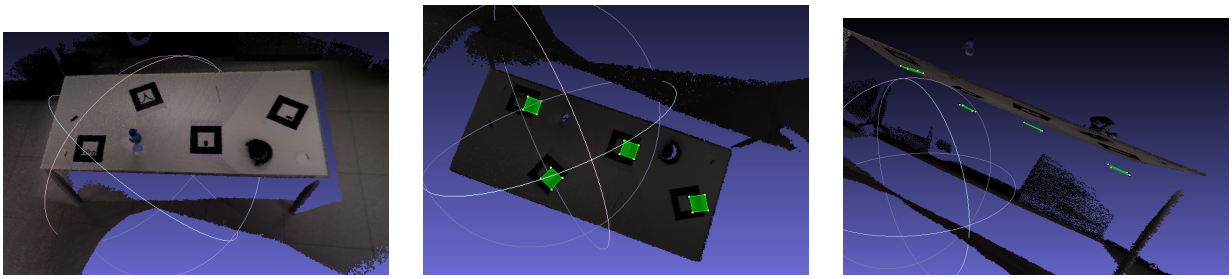
One option would be to implement a process to select the vertices comprising the markers in the EF .ply output and the vertices comprising the markers in the marker pose estimation .ply output and analyse the

difference between those two outputs. Another evaluation option would be to implement an AR application that makes use of a generated multimarker configuration file and with all markers' poses comprising the set measured. To evaluate the error margin of the marker pose estimation a marker could be placed a known distance from the markers comprising a multimarker set. From the set of markers an image could be generated to overlay the marker not in the set. As the position of the marker relative to the markers in the multimarker set is known, how precisely the marker is overlaid could be measured and a quantitative evaluation done.

4.2 Results

4.2.1 Marker Pose Visualization Results

The main purpose of marker pose visualization is to provide feedback on the accuracy of the marker pose estimation process. It is by no means an exhaustive evaluation but is useful in providing visual feedback. Multiple logs were captured and markers' poses estimated from them. An example of one of these visualizations is Fig. 17.



(a) (b) (c)

Figure 17: Top of table containing 4 markers (a), visualization of estimated marker poses (b) from underneath table as there is a depth margin of error (c).

As shown in the figure there is an error margin present in the estimated markers' poses but overall it is within acceptable limits. Each estimated marker pose largely overlays the actual marker. One issue that may have cause for miscalculation is that there may not have been enough frames in which the markers were detected and thus the averaging process may not have had a large enough data set to produce an accurate average.

4.2.2 Multimarker Configuration Files Results

As mentioned above a very quick evaluation of the generated multimarker configuration files is to use them in a multimarker application, cover some of the markers and observe the difference in the overlaid images. Fig. 18 shows a generated multimarker configuration file in use with a multimarker AR application. In this example application blue cubes are drawn over markers that are visible and red cubes if they are not. For the purpose of evaluation first one marker, then two, and finally three markers were obscured. In Fig. 18a we can see the initial arbitrary marker layout. One marker is obscured in Fig. 18b and Fig. 18c. Two markers are obscured in Fig. 18d and Fig. 18e. Finally 3 markers are obscured in Fig. 18f.

This form of evaluation is not as accurate as desired as much depends on the angle of the camera as it is viewing the markers and radical shifts have been noted in the positioning of the cubes drawn due to this. Overall though the results are positive as in general the red cube drawn over obscured markers is within the markers' borders.

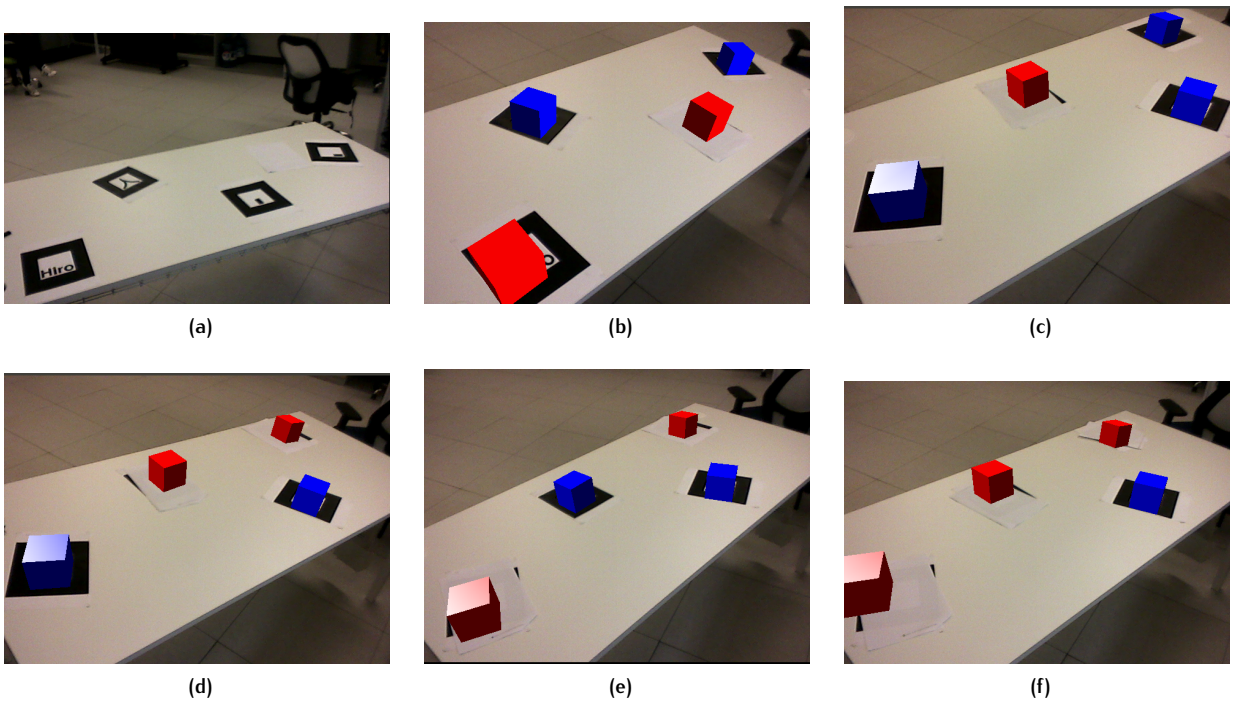


Figure 18: Example multimarker AR application using a multimarker configuration file outputted by markerPosition.

Chapter 5

Conclusion and Further Work

This project achieved a computationally efficient large scale AR system by combining the strengths of 2D marker based AR and 3D dense visual SLAM. This system has the advantage of allowing the development of large scale AR applications without the need for the intensive processing of SLAM. It is now possible to use this system as framework to develop large scale AR applications through the multimarker configuration file output of this system.

There is further work to be done on this system to improve usability and accuracy as well as more thorough evaluation but the framework is in place to do so.

5.1 Accuracy

Overall the results gained were positive but higher accuracy is extremely desirable. One issue that may have effected the accuracy of the marker pose averages is that the markers may not have been detected in enough frames to compute an accurate average. To overcome this a method could be implemented in inputImage to count the amount of frames a marker is detected in and if below a certain number prompt the user to retake the log.

There is also method available in ARToolKit that computes the pose of the detected marker relative to the camera as a continuation of the previous detected marker pose. This method was not used in this project but could be implemented to provide more accurate markers' poses for averaging.

5.2 Improvements to System Pipeline

While this system performs its objective admirably there are improvements to the system pipeline that if implemented would provide a more user friendly system.

A major improvement to the usability of the pipeline is encapsulating the programs inputImage and markerDetection into one class that could be then implemented in EF. Thereby allowing all the computation to occur at once and providing a single output as currently there is a variety of outputs and inputs that must be managed.

A minor improvement to the usability of the pipeline is allowing the specification of inputs and outputs as a terminal command for inputImage and markerDetection as currently all inputs and outputs(text files, logs, etc.) are hardcoded into each of the mentioned programs. Therefore each time an input is changed or renaming an output is needed these programs must be recompiled. This affects the usability of the project and with minor changes this issue could be resolved.

5.3 Further Work

Due to time constraints this project evaluated generated multimarker configuration files with a small amount of markers over a relatively small area. It would be desirable to implement a large scale (room or building size) AR application. It would also be desirable to implement this large scale application using NFT. An example of this could be in a museum where each of the paintings is trained as a marker. The museum could then be scanned and a multimarker configuration file generated comprising of the paintings. An application could then implemented providing the user information on each viewed painting as well as directions to related paintings.

This and other applications would be an ideal way to demonstrate the usability, efficacy, and value of this system in the real world.

Bibliography

[1] ARToolKit. Artoolkit documentation. https://www.hitl.washington.edu/artoolkit/documentation/index.html. Last accessed 28-03-18.

[2] John McDonald. *Multi-session Visual Simultaneous Localisation and Mapping*. PhD thesis, National University of Ireland Maynooth, December 2013.

[3] Thomas J. Whelan. *Real-time Dense Simultaneous Localisation and Mapping over Large Scale Environments*. PhD thesis, National University of Ireland Maynooth, August 2014.

[4] Richard Hartley and Andrew Zisserman. *Multiple view geometry in computer vision*. Cambridge university press, 2003.

[5] Richard Szeliski. *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010.

[6] Jack B Kuipers et al. *Quaternions and rotation sequences*, volume 66. Princeton university press Princeton, 1999.

[7] Ken Shoemake. Animating rotation with quaternion curves. 19(3):245–254, 1985.

[8] Thomas Whelan, Renato F Salas-Moreno, Ben Glocker, Andrew J Davison, and Stefan Leutenegger. *ElasticFusion: Real-time dense SLAM and light source estimation*. PhD thesis, 2016.

[9] Thomas Whelan. Logger2. https://github.com/mp3guy/Logger2, 2016.

[10] Thomas Whelan. Elastic fusion. https://github.com/mp3guy/ElasticFusion, 2017.

[11] Mark Kilgard. Glut. https://www.opengl.org/resources/libraries/glut/, 2018. Last access 26-03-18.

[12] Philip Lamb Hirokazu Kato. Artoolkit5. https://github.com/artoolkit/artoolkit5, 2017.