Malte Schwarzkopf

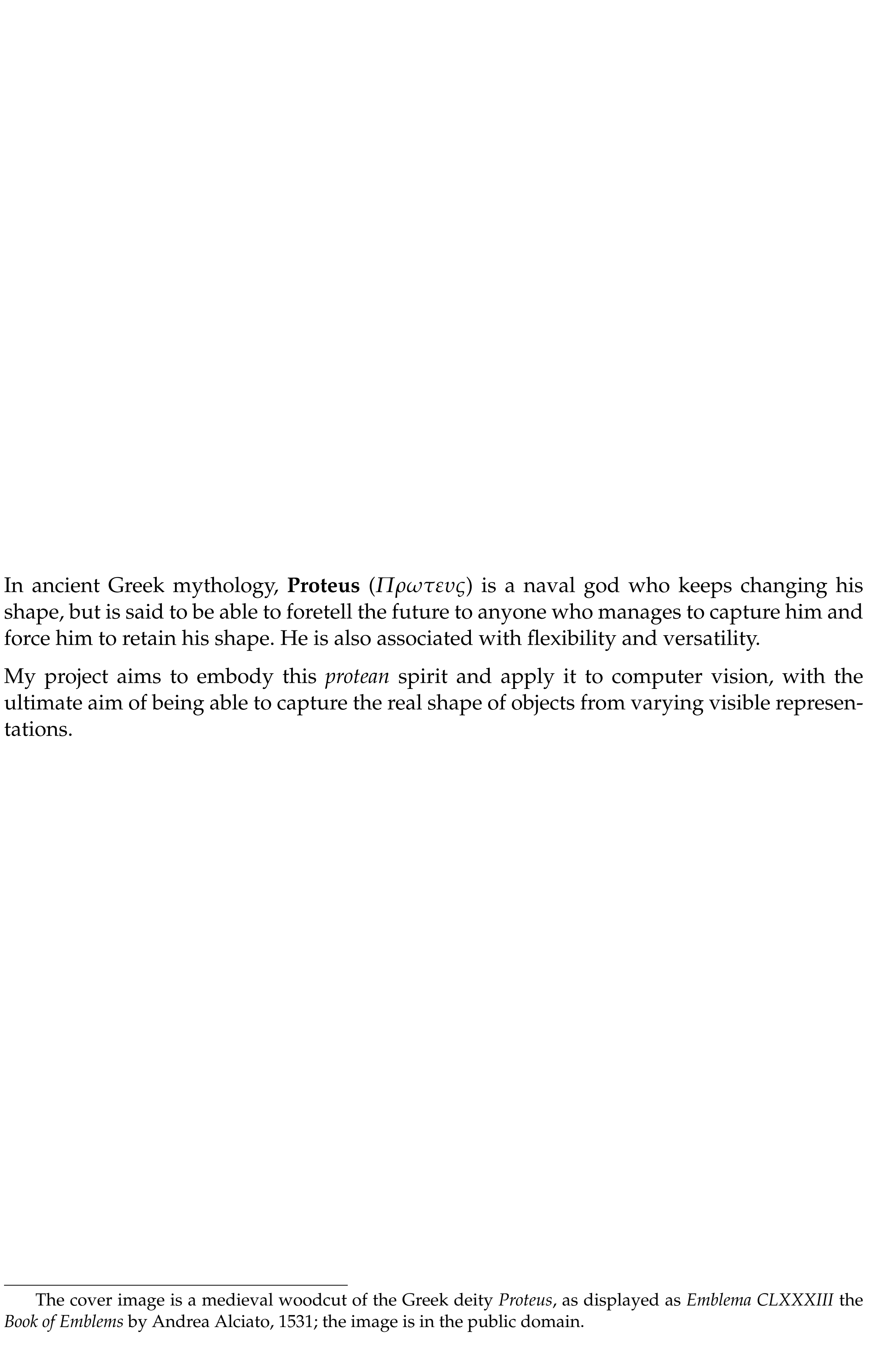
# Proteus

## Interactive Annotation-Based 3D Structure-from-Motion

Computer Science Tripos, Part II

St John's College

May 14, 2009



In ancient Greek mythology, **Proteus** (*Πρωτευς*) is a naval god who keeps changing his shape, but is said to be able to foretell the future to anyone who manages to capture him and force him to retain his shape. He is also associated with flexibility and versatility.

My project aims to embody this *protean* spirit and apply it to computer vision, with the ultimate aim of being able to capture the real shape of objects from varying visible representations.

---

The cover image is a medieval woodcut of the Greek deity *Proteus*, as displayed as *Emblema CLXXXIII* the *Book of Emblems* by Andrea Alciato, 1531; the image is in the public domain.

# Proforma

| | |
|---|---|
| Name: | **Malte Schwarzkopf** |
| College: | **St John's College** |
| Project Title: | **Proteus – Interactive Annotation-Based 3D Structure-from-Motion** |
| Examination: | **Computer Science Tripos, Part II, June 2009** |
| Word Count: | **11,783 words** |
| Project Originator: | **Malte Schwarzkopf and Christian Richardt** |
| Supervisor: | **Christian Richardt** |

## Original Aims of the Project

The design and implementation of an interactive structure-from-motion (SfM) toolkit, suitable for obtaining simple 3D models from content created using consumer-grade digital cameras. This includes creation of a graphical interface for interactively annotating video frames to guide the SfM process. In order to transform a set of feature correspondences in 2D image space into a three-dimensional model, a projective reconstruction needs to be computed, then upgraded to a metric reconstruction and finally textured. The end result should be an accurate, textured 3D model of the annotated object in the input.

## Work Completed

The complexity of the proposed work turned out to greatly exceed the initial anticipation. Nonetheless, the project has been highly successful. All objectives outlined in the proposal have been met: a sequence of video frames can be annotated interactively, supported by edge detection techniques, subsequently converted into a fully textured metric reconstruction of a three-dimensional object and finally displayed using OpenGL. Several optional extensions have been implemented, including an interface to the Kanade-Lucas-Tomasi feature tracker for automatic feature identification, export into the PLY mesh format and serialisation of internal data structures for permanent storage.

## Special Difficulties

None.

## Declaration

I, Malte Schwarzkopf of St John's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date May 14, 2009

# Acknowledgements

A large number of people have contributed advice, feedback and ideas to this project on different levels, but I owe special thanks to:

- **Christian Richardt**, for encouraging me to do this ambitious project, for supervising it, and for providing advice and input on many levels.
- **Emma Burrows**, for all her feedback, encouragement and support. Without you, my time at Cambridge would have been a very different experience.

# Contents

**1 Introduction** **1**
1.1 Motivation . . . . . 1
1.2 Challenges . . . . . 2
1.3 Related work . . . . . 3

**2 Preparation** **5**
2.1 Introduction to Epipolar Geometry . . . . . 5
2.2 Introduction to Structure-from-Motion . . . . . 7
2.2.1 Ambiguities of 3-Space . . . . . 8
2.2.2 Overview of the Computation . . . . . 9
2.3 Requirements Analysis . . . . . 10
2.4 Implementation Approach . . . . . 11
2.5 Choice of Tools . . . . . 13
2.5.1 Programming Languages . . . . . 13
2.5.2 Libraries . . . . . 14
2.5.3 Other Tools . . . . . 16
2.5.4 Development Environment . . . . . 16
2.6 Software Engineering Techniques . . . . . 18
2.7 Summary . . . . . 18

**3 Implementation** **19**
3.1 System Architecture . . . . . 21
3.1.1 Video Decoding . . . . . 21
3.1.2 Data Structures . . . . . 21
3.2 Annotation Interface . . . . . 23
3.2.1 Modes of Operation . . . . . 23
3.2.2 Adding Annotations . . . . . 25
3.2.3 Interactive Guidance using "Snapping" . . . . . 25
3.2.4 Adding Surfaces . . . . . 29
3.3 The Structure-from-Motion Pipeline . . . . . 30
3.3.1 Computing the Fundamental Matrix . . . . . 30
3.3.2 Finding the Camera Matrices . . . . . 32
3.3.3 Triangulation of Points . . . . . 33
3.3.4 Transformation to Metric Reconstruction . . . . . 37
3.4 Texture Extraction . . . . . 38
3.4.1 Texture Data Extraction . . . . . 39
3.4.2 Texture Coordinates . . . . . 39

3.5 Model Viewer . . . 41
3.5.1 Display Modes . . . 41
3.5.2 OpenGL with Qt . . . 42
3.6 Kanade-Lucas-Tomasi Feature Tracker . . . 43
3.7 Mesh Export . . . 44
3.7.1 Choice of Model Format . . . 44
3.7.2 PLY Exporter . . . 44
3.8 Serialization of Annotation Data . . . 45
3.9 Summary . . . 46

**4 Evaluation** **47**
4.1 Overall Results . . . 47
4.2 Testing . . . 48
4.2.1 MATLAB Unit Tests . . . 48
4.2.2 Verification of the C++ Implementation . . . 49
4.3 Empirical Evaluation . . . 50
4.3.1 Performance Evaluation . . . 50
4.3.2 Structure-from-Motion Pipeline . . . 50
4.3.3 Projective Reconstruction . . . 51
4.3.4 Metric Reconstruction . . . 52
4.4 User Study . . . 55
4.4.1 Conduct . . . 55
4.4.2 Identification of Variables and Choice of Tasks . . . 55
4.4.3 Annotation Performance . . . 55
4.4.4 Preference for Edge Snapping Algorithms . . . 57
4.4.5 Overall Impression . . . 59
4.4.6 Discussion . . . 60
4.5 Summary . . . 60

**5 Conclusions** **63**
5.1 Achievements . . . 63
5.2 Lessons Learnt . . . 64
5.3 Future Work . . . 64

**Bibliography** **65**

**Appendix** **69**
A.1 Detailed List of Requirements . . . 69
A.2 Unit Test Code . . . 71
A.2.1 Example Test Case – `test_triang_optimal.m` . . . 71
A.2.2 Batch Testing Script – `run.sh` . . . 72
A.3 Laplacian Operator for Edge Detection . . . 73
A.4 Derivation of the Discrete Line Integral . . . 74
A.5 Derivation of the Fundamental Matrix . . . 74
A.6 Techniques to Optimise Reconstruction Quality . . . 75
A.7 Derivation of $n$-view Homogeneous DLT Triangulation . . . 76
A.8 Test Content . . . 77

A.8.1 Content Creation and Acquisition . . . . . . . . . . . . . . . . . . . . . . . . 77
A.8.2 List of Test Content Sequences . . . . . . . . . . . . . . . . . . . . . . . . . 77
A.9 User Study Material . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
A.9.1 Identification of Variables . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
A.9.2 Informed Consent Form . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
A.9.3 Outline of running the Study . . . . . . . . . . . . . . . . . . . . . . . . . . 79
A.9.4 Post-Study Questionnaire . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
A.10 Original Project Proposal . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81

# Chapter 1

# Introduction

*In vain, though by their powerful Art they bind*
*Volatile Hermes, and call up unbound*
*In various shapes old Proteus from the Sea,*
*Drain'd through a Limbec to his native form.*

— John Milton, Paradise Lost III.603–06

This dissertation describes the creation of a framework that makes it straightforward for anyone, even if not yet an expert in the field, to obtain a virtual three-dimensional representation of a real-world object from a digital video sequence.

## 1.1 Motivation

In recent years, we have seen multimedia applications using digital video as well as 3D graphics becoming increasingly ubiquitous in mainstream computing. Many such applications depend on user-contributed visual content. Although tools to aid with the process of acquiring this exist, there does not yet exist a tool that unifies both automated analysis of video data and interactive user input. This would be especially useful for the purpose of generating 3D content such as 3D building models to use in a virtual online world. For example, the virtual cities in Google Earth (see Figure 1.1 overleaf) consist of thousands of virtual 3D buildings.

There are various diverse use cases for such a tool, but the most important ones seem to be urban modelling, augmented reality, object tracking and video post-production.

As an example, one could imagine 3D city models (such as those currently used in Google Earth) being generated automatically from aerial videos; or individual users creating videos in order to use these for augmented reality scenarios, such as the virtual world of Second Life[1], a level for a computer game, or even a simulation of how they could rearrange the furniture in their homes.

[1] `http://www.secondlife.com/`

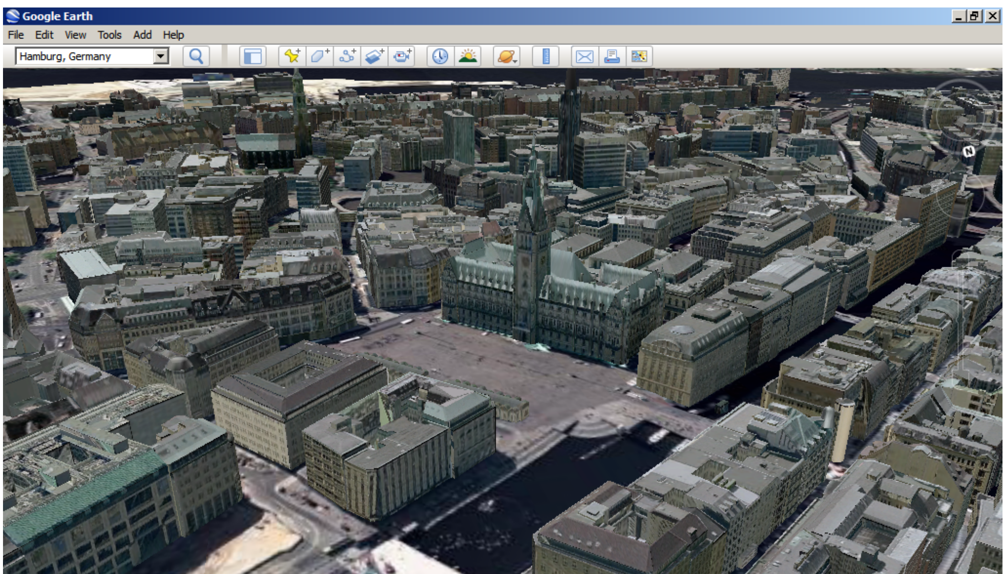



*Figure 1.1: Virtual 3D city in Google Earth: St Michael and the Town Hall in Hamburg, Germany, as seen from above the Alster lake.* *3D Buildings © 2008 CyberCity AG, Images © 2009 AeroWest*

Significant fundamental work was done on 3D structure-from-motion (SfM) – that is, the process of obtaining a three-dimensional model from object or camera motion – in the 1990s (see Section 1.3). However, the existing implementations lack accessibility to non-scientific user groups. These approaches are usually fully-automated and generate point clouds from video, rather than solid models. Additionally, the quality of any automatic reconstruction is vulnerable to high degrees of ambiguity, degeneracies in camera motion and feature misidentification and mismatching.

The advantage of fully-interactive modelling techniques is that they make use of the user's perception of the object that is being created and can rely on the user to ensure that the model fits the expectations. However, fully-custom modelling is extremely time-consuming and generally requires specialist skills in using a modelling package.

A combination of the two, as presented in this project, would seem like an optimal approach – using structure-from-motion to determine object properties from video data, and exploiting the intelligence of the user to guide this process.

## 1.2 Challenges

A well known anecdote of computer vision is that AI pioneer Marvin Minsky in 1966 gave the task of solving the problem of "making a computer see" to a student as a summer project. In this, he was dramatically underestimating the difficulty of the problem, which turns out to remain unsolved in the general case up to the present day.

Nowadays, computer vision is regarded as a very challenging field, and the concept of 3D structure-from-motion is generally recognised within the research community to be a difficult problem. Despite working algorithms existing, implementations are still seen as chal-

lenging and are generally expected to be imperfect.

A major complication is that the problem of inferring three-dimensional shape from two-dimensional images is fundamentally *ill-posed* in the sense introduced by Hadamard [10]: it violates the second and third criterion for a well-posed problem; namely, it does not have an unique solution and its solution does not continuously depend on the data.

These facts become immediately obvious if the task at hand is considered. The goal is to infer a 3D shape from images, which are two-dimensional optical projections – and this task comes down to inverting the 3D $\rightarrow$ 2D projection. It is mathematically impossible to find an unique solution to this, since the process of projection is "destructive" in the sense that it discards information.

Consequently, the mathematical problems that this project has to solve suffer from an extremely high degree of ambiguity. Depending on the type of 3-space considered (see Section 2.2.1), between 6 and 15 degrees of freedom exist within the data. Hence any algorithm capable of obtaining results of satisfactory quality is bound to be fairly complex and its implementation a challenging task to undertake.

Finally, the emphasis on creating a framework accessible for interactive use complicated the project more by introducing a whole set of further problems: GUI development, concurrency aspects of live video processing, human-computer interaction and design of data structures allowing for arbitrary input content added additional layers of complexity, despite these being, at best, tangential to the problem of structure-from-motion.

## 1.3 Related work

A comprehensive survey of the entirety of the previous work done in the areas relevant to this project would by far exceed the scope of this dissertation. However, a number of sources were especially useful to me, both for familiarisation with the area in general and implementation of the project in particular.

The standard textbook in the area of SfM is the extensive book by Hartley and Zisserman [12] which covers two-, three- and $n$-view geometry and the associated geometric principles and numerical methods. This book quickly proved to be an indispensable resource for this project, and will be referenced repeatedly throughout this dissertation.

Significant fundamental work was done by Pollefeys *et al.* at the University of North Carolina at Chapel Hill and KU Leuven, resulting in a number of publications [7; 17–19]. Of these, those covering the acquisition of 3D models from uncalibrated hand-held cameras [17; 19] are of particular interest.

Some further work, especially on estimation of the fundamental matrix, degeneracy detection and compensation has been done by Torr [3; 26; 27] at Oxford Brookes University. His work contributed to the Hartley and Zisserman book [12], and he also provides a comprehensive MATLAB structure-from-motion toolkit for two-view geometry [25].

An interesting approach, which is very similar to the aims and design of this project, was presented by Häming and Peters [11]. Their intent was to design a C++ SfM toolkit to be used for teaching. Unfortunately, only a single high-level description was published and the

implementation itself is not available.

The implementation that is probably closest to my approach is VideoTrace [31], a semi-commercial research project of the Australian Centre for Visual Technologies and Oxford Brookes University. VideoTrace focuses on rapid interactive modelling of complex structures with a high degree of detail, supporting advanced concepts such as curved surfaces (using NURBS), extrusions and mirror planes. It differs from this project in the sense that the structure-from-motion analysis is performed prior to interactive user input being sought.

In addition to the fundamental work done in recent years, the augmented reality idea has received some research interest [7], as has the idea of urban modelling using automated structure-from-motion techniques [2]. Vergauwen, Pollefeys *et al.* also presented some work on the use of structure-from-motion techniques in archaeology and cultural heritage conservation [20] as well as control of autonomous vehicles in planetary exploration [32; 33].

Commercial exploitation of the techniques developed by research over the course of the last two decades in this area is only just in its infancy. The VideoTrace project, for example, is aimed at future commercial exploitation in video post-processing, and the Oxford-based company 2d3[2] specialises in products for camera and object tracking, feature detection, visual effects and mesh generation based on 3D structure-from-motion techniques for industrial and governmental use.

[2] `http://www.2d3.com/`

# Chapter 2

# Preparation

Undertaking a project of this scope requires structure, organisation and a high degree of careful planning. Required resources have to be identified and their use optimised – whilst at the same time minimising risks as well as maintaining focus.

This chapter starts off by outlining the basic principles of epipolar geometry and giving an overview of structure-from-motion, as familiarity of the reader with this field of computer vision cannot necessarily be assumed. The analysis of requirements that was conducted and the conclusions drawn from it are described. Finally, these are related to choices made about tools and development methodologies used.

## 2.1 Introduction to Epipolar Geometry

Before I move on to explaining the principles of structure-from-motion, I am briefly going to introduce the essentials of *epipolar geometry* and comment on how they are represented in the *Proteus* project.

The term of *epipolar geometry* is inherently linked to multiple-view vision, and stereo vision in particular, *i.e.* the concept of two cameras viewing a scene from distinct positions. Most of the relations of epipolar geometry can be derived from the familiar *pinhole camera model* of optics. This defines points of interest such as the *focal point* and the *image plane*. The latter is located behind the focal point in the pinhole model and contains a top-down mirrored version of the image (see Figure 2.1).

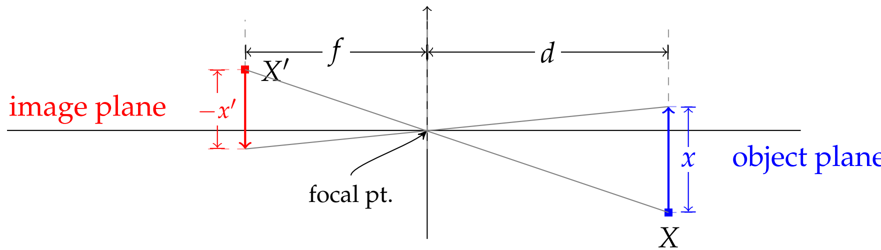



*Figure 2.1: Simplified view of the classical single-view* pinhole camera model *of optics.*

For simplification, epipolar geometry considers a *virtual image plane* in front of the camera which is simply a mirror image of the actual image plane at distance $-f$ from the camera. If we consider a point $\mathbf{X}$ in 3D space, seen by two distinct cameras, say $\mathbf{C}_1$ and $\mathbf{C}_2$, then, following the pinhole model, the point will have a projection $\mathbf{x}_1$ in the image plane of $\mathbf{C}_1$, and a projection $\mathbf{x}_2$ in the image plane of $\mathbf{C}_2$.

Figure 2.2 illustrates the most important relations of epipolar geometry. The common plane defined by 3D points $\mathbf{X}$, $\mathbf{C}_1$ and $\mathbf{C}_2$ is called $\boldsymbol{\pi}$, and its intersections with the image planes define the *epipolar lines* $\mathbf{l}_1$ and $\mathbf{l}_2$ in the image planes. Furthermore, the intersection points between the *camera baseline* (the line connecting the two camera positions $\mathbf{C}_1$ and $\mathbf{C}_2$) and the image planes are called *epipoles*, denoted $\mathbf{e}_1$ and $\mathbf{e}_2$. Any projection $\mathbf{x}_i$ and the corresponding epipole $\mathbf{e}_i$ always lie on the same epipolar line $\mathbf{l}_i$. Note that it is perfectly possible for the epipoles to lie outside the image.

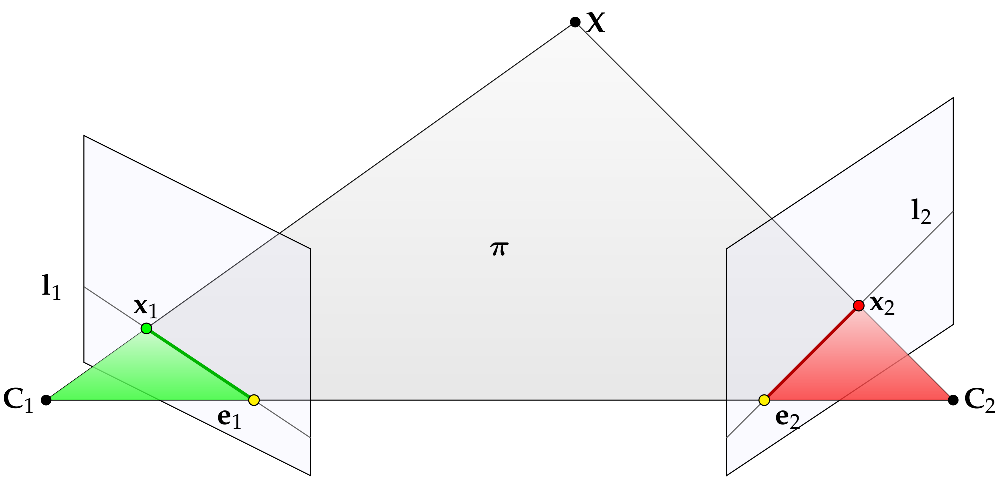



*Figure 2.2: Epipolar geometry basics: The camera baseline intersects the image planes at the* epipoles $\mathbf{e_1}$ *and* $\mathbf{e_2}$*; the* epipolar plane $\boldsymbol{\pi}$ *intersects the image planes in the* epipolar lines $\mathbf{l}_1$ *and* $\mathbf{l}_2$.

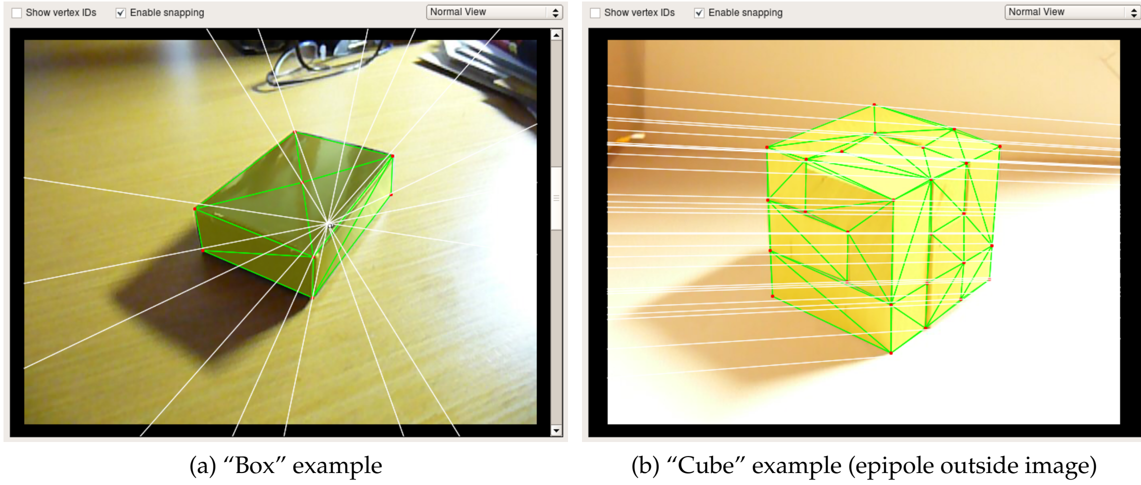
These definitions are relevant to the structure-from-motion process because they immediately lead to the formulation of the *epipolar constraint* and the process of *triangulation*:

- If the projection of a common point $\mathbf{X}$, say $\mathbf{x}_1$, is known, then a line from the camera centre through this projected point can be constructed, and the projection of the line in a different view immediately gives rise to the epipolar line $\mathbf{l}_2$. The projection of the same 3D point in the secondary view's image plane is then *constrained* to lie on this epipolar line. If two projections of a point do not lie on (or close to) the reciprocal epipolar lines, they cannot correspond to the same 3D point.
- By reversal of this, if two projections of a 3D point are known – say $\mathbf{x}_1$ and $\mathbf{x}_2$ – then their projection lines (the lines from the camera centres through the projected points) must intersect in 3-space precisely at $\mathbf{X}$.

The process of calculating the shared 3D point $\mathbf{X}$ from two projections as described above is called *triangulation*. It should be immediately obvious that the assumption of the projective rays intersecting exactly in $\mathbf{X}$ is only likely to be true for extremely accurate synthetic data; in

the case of structure-from-motion using noisy images, there will be a certain degree of error and the projection rays will not meet exactly (although they should still be close).

While implementing this project it became apparent that analysis of the relations of epipolar geometry on given images was a useful tool for debugging. For this reason, I included a facility to draw the epipoles and the epipolar lines for every matched point between two frames of a video. An example of this can be seen in Figure 2.3; it was especially useful to be able to judge the quality of a set of annotations based on whether the epipolar lines coincided in one point.




(a) "Box" example

(b) "Cube" example (epipole outside image)

*Figure 2.3: Example content in* Proteus *with epipolar lines and epipoles drawn as overlays.*

An algebraic representation of epipolar geometry, including the epipolar constraint defined above, is given with the *fundamental matrix* F. Its meaning and computation is described in further detail in Section 3.3.1.

## 2.2 Introduction to Structure-from-Motion

The term *Structure-from-Motion* is, in general, used to describe any process of inferring three-dimensional object structure from spatial displacements of features over time due to object or camera movement.

The particulars of the process depend on what sort of input data is provided, what the viewing parameters are, and what the desired output result is. For instance, the techniques for material obtained from fixed-distance, calibrated stereo camera rigs are very different from those for uncalibrated, arbitrarily translated hand-held cameras. Similarly, the required output quality for a photorealistic model to be used in augmented reality scenarios is very different from the accuracy required for a navigational system in robotics.

| Group | Matrix | Example | DoF | Invariants |
|---|---|---|---|---|
| **Projective** | $\begin{bmatrix} \mathtt{A} & \mathbf{t} \\ \mathbf{v}^\mathsf{T} & v \end{bmatrix}$ | | 15 | Intersection of lines; tangency of surfaces. |
| **Affine** | $\begin{bmatrix} \mathtt{A} & \mathbf{t} \\ \mathbf{v}^\mathsf{T} & v \end{bmatrix}$ | | 12 | Parallelism of lines and planes, centroids, plane at infinity ($\pi_\infty$). |
| **Metric / Euclidean** | $\begin{bmatrix} \mathtt{R} & \mathbf{t} \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix}$ | | 6 | Angles; Volume (Euclidean only). |

*Table 2.1: Common transformations of 3-space and their properties.* $\mathtt{A}$ *is an invertible* $3 \times 3$ *matrix,* $\mathtt{R}$ *is a 3D rotation matrix,* $\mathbf{t} = (t_X, t_Y, t_Z)^T$ *a 3D translation,* $\mathbf{v}$ *an arbitrary 3-vector,* $v$ *an arbitrary scalar and* $\mathbf{0}$ *a zero-3-vector. Summarised from [12, p. 78].*

### 2.2.1 Ambiguities of 3-Space

When considering three-dimensional space (3-space) in the context of structure-from-motion, it is important to note first that all coordinates and transformations are specified in terms of *projective geometry*, using homogeneous coordinates. Depending on the types of transitions that are permissible, the level of correspondence between the original object and its reconstruction in 3-space can differ as a result of varying degrees of ambiguity.

The different classes of ambiguities in 3-space are defined by the transformations that can validly be applied to the original object. Only a brief and highly simplified introduction is given here; literature [12, p. 37–ff., 77–ff.] provides more rigorous and detailed material. Table 2.1 illustrates the properties of the different classes of 3-space described in the following.

**Projective transformations** are only constrained to maintain intersections and tangency – arbitrary rotations, scalings and translations of any coordinate point are permissible.

**Affine transformations** maintain parallelism and vanishing points in addition to the above, but do not preserve angles, so objects can appear skewed.

**Metric transformations** maintain all geometric properties except for volume, and only scaling is permissible. *Euclidean* space is a special case of this and also preserves volume.

By the scale-variant property of metric space, any reconstruction obtained using structure-from-motion techniques can only possibly ever be up to an arbitrary scale factor $k$, unless additional data specifying lengths and distances in the input is provided. This fact is mirrored by the final reconstruction being *metric*, *i.e.* a *Euclidean* reconstruction up to a constant scale factor.

### 2.2.2 Overview of the Computation

In general, the process of transformation from flat projections in 2-space to a 3-space model involves the following four steps (in the spirit of the general algorithm presented by Hartley and Zisserman [12, p. 453], although with a slightly different emphasis):

1. **Identification of features and correspondences**: Distinctive features in the input images are selected and tracked across a series of images. This establishes, for $n$ input images, a set of up to $(n-1)$ 2D point correspondences $\mathbf{x}_i \leftrightarrow \mathbf{x}_j$ for some $1 \leq i < j \leq n$. This is done interactively by the user annotating a video in my implementation – see Sections 3.1.2 and 3.2 for details.

2. **Finding an initial solution for scene structure and camera positions**

   (a) **Determination of the fundamental matrix**: The initial pairwise point correspondences are used to determine the fundamental matrix $\mathtt{F}$ using one of several possible algorithms (linear, non-linear or stochastic), which represents the epipolar geometry relations and constraints algebraically. The implementation of this step is described in Section 3.3.1.

   (b) **Extraction of camera motion** between images: Using the fundamental matrix $\mathtt{F}$, a *camera matrix* $\mathtt{P}_i$ can be determined for the $i^{\text{th}}$ pairwise correspondence (out of an overall of up to $(n-1)$ pairwise correspondences). The computation is explained in Section 3.3.2.

   (c) **Triangulation** of point correspondences: Using triangulation over a set of 2D projections ($\mathbf{x}_i$) in different images using different cameras ($\mathtt{P}_i$), an initial estimate of the corresponding 3D point $\mathbf{X}_i$ is obtained. Multiple triangulation algorithms were implemented for this project – see Section 3.3.3.

3. **Improvement of initial solution**: The initial reconstruction is *projective*, and only identical to the original object up to a projective transformation (see Table 2.1). In order to ameliorate this, the reconstruction must be *upgraded* to a metric reconstruction, which can be achieved in a variety of ways. For this project, the *direct reconstruction* method was chosen, and is described in Section 3.3.4.

4. **Model Building**: In the final step of the structure-from-motion process, surfaces are inserted into the refined reconstruction, textures extracted and the reconstructed rigid model is built from the appropriate primitives. Frequently this is done using dense point clouds, surface modelling with depth maps or volumetric integration, e.g. using the marching cubes algorithm [14; 18].

## 2.3 Requirements Analysis

The initial analysis of requirements was a crucial step in the early stages of this project. The overall architecture of the framework to be implemented had to be established, and dependencies between modules understood so that development could be structured accordingly. The main aims of the project are summarised in Table 2.2; these coincide with the original objectives and success criteria set in the project proposal (see Appendix A.10), bar a few minor modifications. A detailed breakdown of the sub-goals within each of these was created in order to facilitate more fine-grained planning. This list is included in Table A.1 in the appendix.

| Goal Description | Priority | Difficulty | Risk |
|---|---|---|---|
| Build graphical user interface frontend | High | Medium | Low |
| Implement interactive guidance techniques | High | Medium | Low |
| Implement structure-from-motion algorithms | High | High | High |
| Implement texture extraction | High | Medium | Medium |
| Add a facility for exporting the generated model | Medium | Medium | Low |
| Integrate an existing automated point tracker (KLT) | Low | Medium | Low |

*Table 2.2: The top-level project objectives in* Proteus.

Furthermore, I undertook an evaluation of the dependencies of the different modules of the project – this was important both in order to be able to make decisions about the development methodology to be adopted and in order to decide on the order in which to implement the different modules. The major dependencies in this project are outlined in Figure 2.4.

In this process, it became apparent that certain core parts of the project, namely the structure-from-motion pipeline, required a strictly linear implementation as they were part of a chain of dependencies. Other components were more independent and could feasibly be filled in with extremely basic prototypes at first, or even left for a later implementation without impacting on the overall functionality.

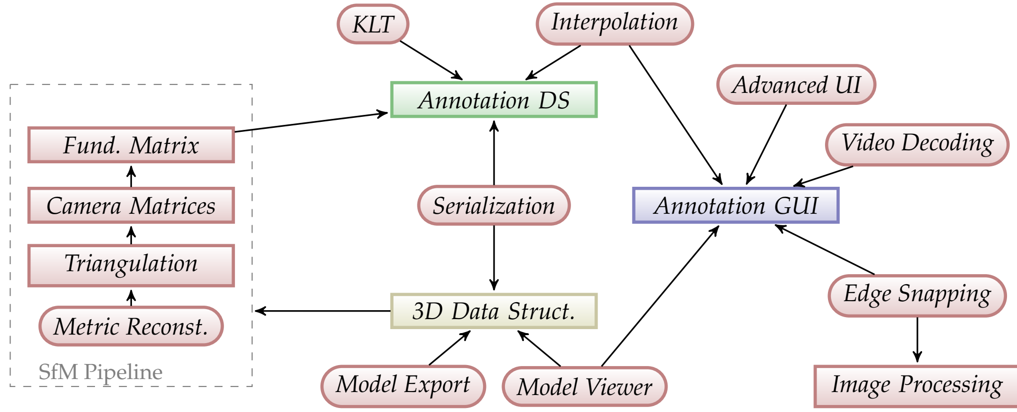



*Figure 2.4: Main dependencies in the* Proteus *project. Dependencies are in arrow direction.*

## 2.4 Implementation Approach

The first step of the conceptual planning stage for the project was bound to be a step of restraining the complexity, and compiling a list of assumptions that had to be made in order to make the implementation of the project tractable. This section outlines the decisions made and aims to explain why boundaries were staked like this.

The general functionality that was required to be implemented in order to meet the requirements is outlined schematically in the diagram overleaf (Figure 2.5).

An immediate consequence of the ill-posed nature of the structure-from-motion problem was that the project had to be limited to a specific domain. The first assumption I made was that the objects modelled could not be moving themselves (*i.e.* limited to videos of stationary, static objects with only the camera moving).

In the first stage of the project, a basic interactive annotation interface had to be implemented and then extended with edge detection to allow interactive "snapping" to image features. In parallel with this, the data structures to hold the annotation information had to be designed and implemented – see Section 3.2 for details.

Since the implementation of the structure-from-motion algorithms was identified as a critical and high-risk component of the project, I decided to limit these to the simplest known algorithms at first, potentially extending them with more complex alternatives that would yield better reconstructions later. The initial algorithms chosen were the *normalised 8-point algorithm* [12, p. 281–ff.] for the computation of the fundamental matrix, the deterministic formula for canonical cameras for finding the camera matrices, and the simple *homogeneous linear triangulation* [12, p. 312–ff.] for estimating the 3D points (see Section 3.3).

Another important fundamental choice I made was the way to implement the upgrade to a metric reconstruction: it became evident early in the project that the complexity of this step, and in particular of attempting a complete implementation of the stratified approach including self-calibration, had been underestimated and was likely to lead to an unmanageable explosion in complexity. For this reason I chose to pursue a simpler approach: a *direct rectification to metric using ground truth* [12, p. 275–ff.] – see Section 3.3.4 for details.

As the last step of the reconstruction of a 3D model, textures are extracted from the input video and applied to the reconstructed model. I decided to communicate with OpenGL directly for this, *i.e.* not requiring intermediate storage of texture data to files. The approach used to extract texture data and coordinates is described in Section 3.4.

In order to be able to display and examine the 3D reconstructions, I chose a combination of two approaches. Firstly, I decided to build an interface to OpenGL and design an internal model viewer; this allows for direct and interactive inspection of the 3D model during the annotation process and hence provides a degree of direct feedback to the user. In addition to this, I created an exporter into the PLY mesh format[1] (see Section 3.7).

A further extension to the project was the integration of an automated point tracker (see Section 3.6); the implementation was designed so that the integration of a library for this purpose was supported.

[1] Also known as the *Stanford Triangle Format*, `http://local.wasp.uwa.edu.au/~pbourke/dataformats/ply/`

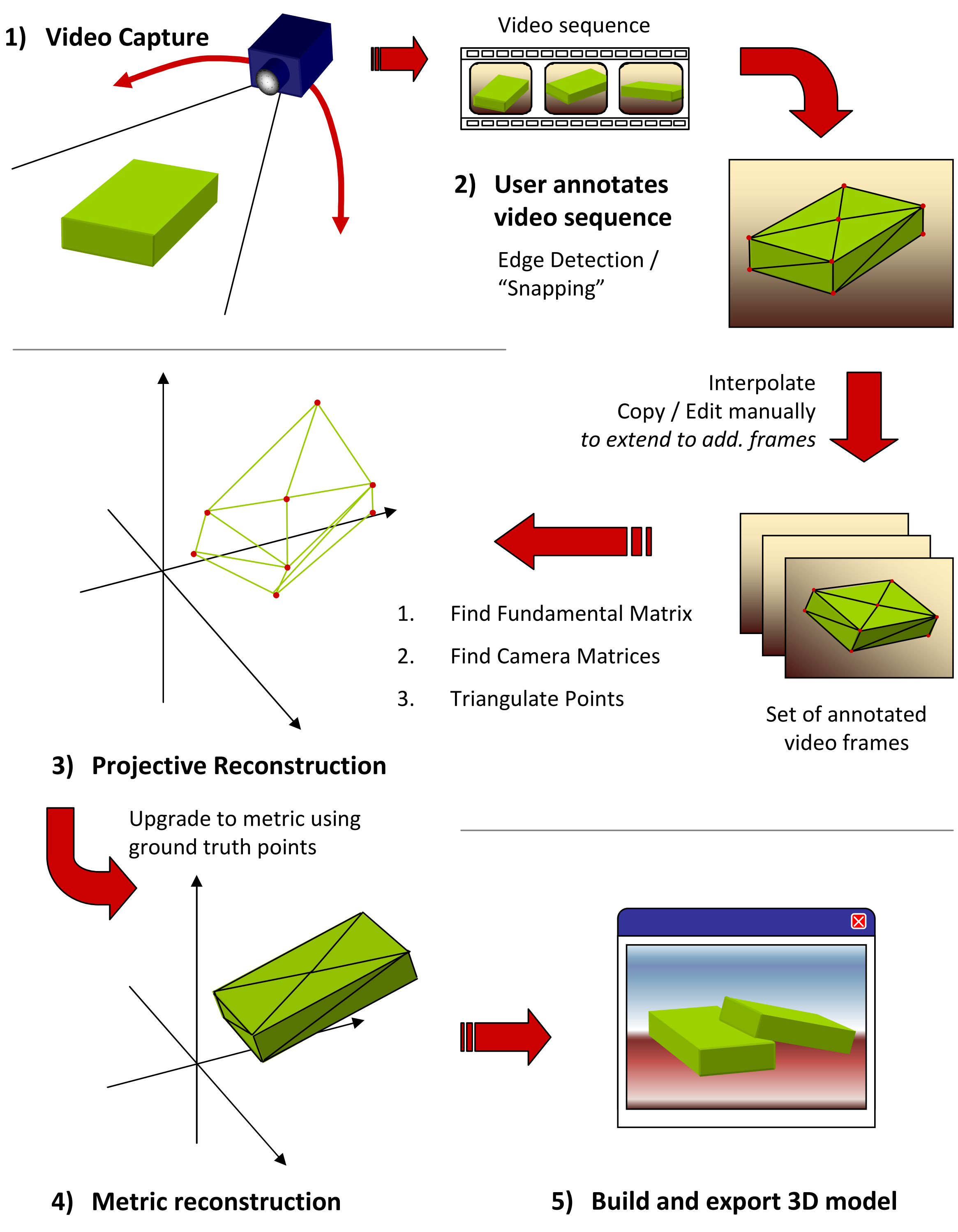



*Figure 2.5: Schematic diagram of the functionality implemented by the* Proteus *project.*

## 2.5 Choice of Tools

### 2.5.1 Programming Languages

#### C and C++

The main programming language chosen for implementing the project was C++. This decision was made two main reasons:

1. The Qt GUI toolkit (see Section 2.5.2) is written in C++ and provides convenient C++ interfaces to its extensive functionality. It even extends the C++ language to an extent by introducing special constructs like `foreach` loops and providing convenient interfaces to standard libraries, such as `QString` to `std::string`.
2. The video decoding library and the high-performance linear algebra libraries are straightforward to interface with C++, as these are written in C and C++.

I already had some substantial experience programming in C to start off with, but had not used C++ beyond small examples and exercises from the taught course in Part IB. This meant that using it required a degree of familiarisation, first with C++ itself and then with the particular dialect used by Qt code.

#### MATLAB

Many of the algorithms were, prior to developing the final C++ implementation, prototyped in the *M* programming language of the MATLAB numerical computing package. MATLAB is a prevalent tool for research and prototype implementations in the area of computer vision and image processing, and some reference implementations of similar algorithms [13; 34] were also available for MATLAB .

```cpp
if (    X.rows()     == 2
     && X.columns() == 1) {
     // 2-column vector
     Matrix T(2,2);
     T[0][0] = 0;     T[0][1] = 1;
     T[1][0] = -1;    T[1][1] = 0;

     Y = transpose(X) * T;
}
```

*Listing 2.1: C++ version (using QuantLib)*

```matlab
if all(size(X) == [2 1])
     % 2-column vector
     Y = X'*[0 1; -1 0];
end
```

*Listing 2.2: MATLAB version*

*Both listings implement:*

$$\mathbf{Y} = \mathbf{X}^{\mathrm{T}} \times \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

*Figure 2.6: Comparison between C++ and* M *implementations of a simple matrix operation.*

*M code* makes it very easy to perform both simple and more advanced operations on vectors and matrices. This is illustrated in Figure 2.6 – note how much closer the *M* implementation is to the mathematical notation. The interactive command line and plotting tools facili-

tate rapid prototyping and aid at initially implementing algorithms without getting bogged down by low-level implementation details or syntactic intricacies.

Furthermore, using MATLAB allowed for tests with synthetic data to be conducted as part of the iterative prototyping process in order to guide and verify the C++ implementations (see Sections 2.6 and 4.2.1).

### 2.5.2 Libraries

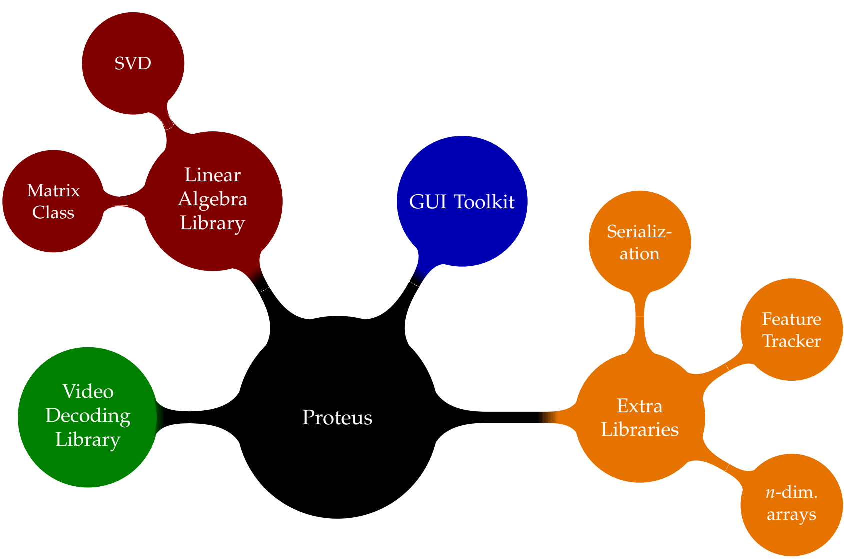



*Figure 2.7: Conceptual diagram of the abstract library functionalities required in* Proteus *and their relationships.*

The nature of this project implied from the very beginning that it would have to rely heavily on libraries to provide much of the basic functionality that was unnecessary and infeasible to reimplement completely. The abstract functionality provided by libraries, and their interaction, are outlined in Figure 2.7, and a concise summary of the libraries used is given in Table 2.3.

I usually chose the libraries from a set of possible alternatives – preference was given to libraries that were written natively in C++ or included C++ interfaces, since these would integrate more easily with the rest of the project.

#### Video Decoding

A crucial choice in the early stages of the project was the choice of **libavcodec** for decoding the video input data. This library was originally developed as part of the `FFmpeg` video transcoding tool and contains decoding functionality for a very large set of common video formats. *libavcodec* seemed to be a good choice due to its widespread use in open source multimedia applications. The poor quality of its documentation later turned out to make it more difficult to use than anticipated.

| *Library* | *Version* | *Purpose* | *License* |
|---|---|---|---|
| libavcodec | CVS | Video decoding (from `FFmpeg` project) | LGPL |
| Qt | 4.4.3 | GUI toolkit | GPL |
| QuantLib | 0.9.0 | Matrix & SVD library | BSD (new) |
| LAPACK | 3.1.1 | Linear algebra library | BSD |
| BLAS | 1.2 | Basic linear algebra library used by LAPACK | Public Domain |
| Boost | 1.34.1 | Multi-dimensional arrays, Serialization | Boost Software License |
| GSL | 1.12 | Polynomial root finding | GPL |
| KLT | 1.3.4 | KLT Feature Tracker implementation | Public Domain |

*Table 2.3: Libraries used in the Proteus project.*

**Graphical User Interface**

The central choice of library for the project was probably the decision to use the **Qt4** toolkit for the user interface. Qt, originally created as a GUI toolkit, is a cross-platform framework that provides a broad range of functionalities such as concurrency control, file access and multimedia integration, implemented in a consistent, object-oriented way. It seemed ideally suited for this project due to its cross-platform nature, the availability of tools to facilitate rapid GUI design and modification [28] as well as IDE integration [29] and richness of documentation [30].

**Linear Algebra**

The third main choice that had to be made in terms of core libraries was the choice of a linear algebra library for matrix operations as well as for solving systems of linear equations. At first, the TooN library [9] was considered to be the best choice for this – mainly due to its appealing C++ interface that allows both statically and dynamically sized vectors and matrices and provides compile-time bounds-checking for statically sized ones.

However, it turned out that TooN's functionality for the Singular Value Decomposition [12, p. 585–ff] was insufficient for the needs of the project as TooN does not expose the three result matrices of the SVD and provides no way to access them.

In the end, **QuantLib** was chosen as linear algebra library. QuantLib is a very large set of libraries mainly intended for quantitative finance applications; only a small subset of its functionality was actually required and used, but this could be achieved by linking against parts of the library only. As its dependencies, QuantLib brought in the LAPACK and BLAS libraries; these are highly efficient linear algebra libraries written in Fortran-77 that form the basis of most modern linear algebra libraries.

#### Other

Small subsets of the functionality provided by the popular **Boost** C++ class library were used: In particular, `boost::multi_array` (dynamically-sized multi-dimensional arrays) and the Boost serialization interface were used.

For the implementation of the optimal triangulation algorithm (see Section 3.3.3), a library capable of finding the roots of polynomials of arbitrary degree was required. The **GNU Scientific Library** (GSL) was used for this purpose.

In order to implement the automated feature tracker extension, a reference implementation [4] of the Kanade-Lucas-Tomasi feature tracker [22] was compiled as a library and dynamically linked into the project.

### 2.5.3 Other Tools

For viewing, editing and debugging PLY mesh files and to verify the models displayed in the internal model viewer, the Qt4-based **Mesh Viewer**[2] ("mview") v0.3.2 was used, as well as **MeshLab**[3] v1.1.1.

### 2.5.4 Development Environment

The majority of development work was carried out on my personal machines, running Ubuntu Linux 8.04 and 8.10, although additional resources provided on the PWF and by the SRCF[4] were used for some purposes, primarily for backups (see Section 2.5.4).

#### Integrated Development Environments

The main part of the project code was written using the Eclipse IDE, v3.4.1 ("Ganymede"), with additional plugins to extend its capabilities: The C/C++ Developer Tools (`cdt`) for development of C/C++ code, Qt integration (`qtcpp`) and Subclipse for Subversion integration. Eclipse is a powerful IDE that facilitates good software development by aiding in organising projects into logical structures and making tedious code refactoring tasks easier. It also integrates with the `gdb` debugger to allow for single-step debugging, stack and memory analysis and profiling.

MATLAB code was written in the `gedit` text editor. This is a plain and simple text editor that provides syntax highlighting for *M* code. The code was executed and interactively tested using the *M* interpreter in the MATLAB -compatible open source package GNU Octave[5], although the commercial MATLAB package on the PWF had to be used in some cases when non-free toolkits (such as the MATLAB Image Processing Toolkit) were required.

[2] `http://mview.sourceforge.net/`, LGPL
[3] `http://meshlab.sourceforge.net/`, GPLv2
[4] `http://www.srcf.ucam.org`
[5] `http://www.gnu.org/software/octave/`

### Version Control

Some form of version control was crucially important to organising the development process for this project. I chose to use the Subversion 1.4 source control management system for this. This allowed me to, at any point in the development cycle, go back to a previous revision of a file and facilitated a *modify-debug-commit* cycle, and furthermore also allowed development work to proceed on multiple machines with the results being merged in the repository. Subversion was also used to hold the LaTeX source code of this dissertation whilst it was being written.

### Backup Strategy

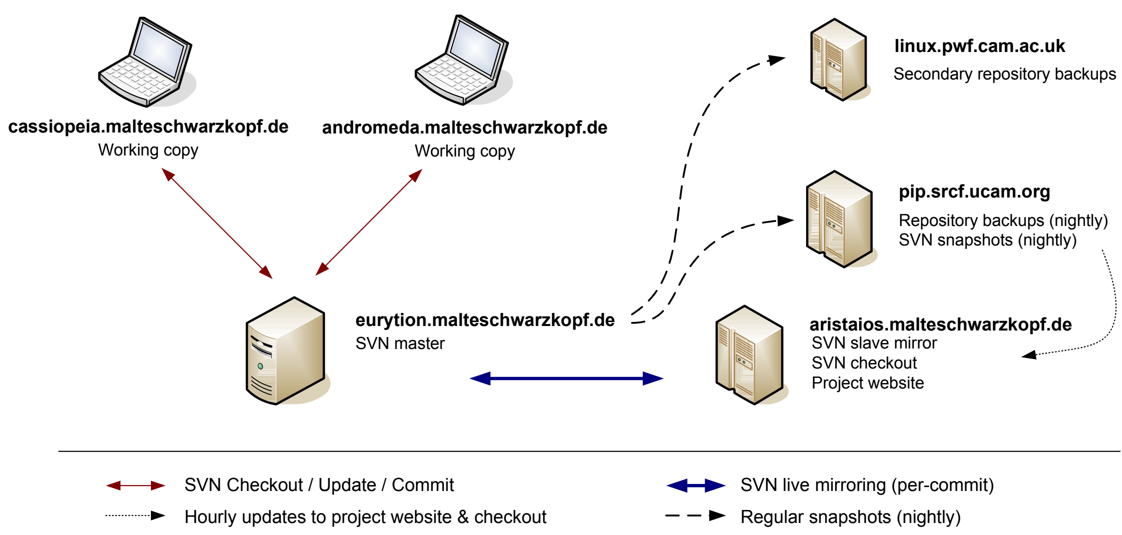



*Figure 2.8: Backup strategy employed for the Proteus project.*

The backup arrangements made were intended to cover all plausible failure modes and ensure that no reliance on any single component working was required. Figure 2.8 illustrates the backup strategy adopted.

All project data was organised in the centralised Subversion repository stored on my file server machine. The `svnsync` tool was used to keep a synchronous backup of the repository on an off-site machine. The commit hooks on the main repository were modified to trigger an immediate synchronisation after each commit in order to achieve maximum backup coverage.

Additionally, the entire repository was automatically copied to two off-site locations (the SRCF server and the PWF file space) by a shell script on a daily basis, and an automated clean copy of the workspace was kept to for recovery from possible repository corruption.

## 2.6 Software Engineering Techniques

The development methodology chosen for this project had to fit in with a number of requirements that arose as a result of the nature of the project. Based on the considerations detailed in Section 2.3, I decided to adopt a *Waterfall Development Model* [24, p. 66–ff.] in union with formalised unit-testing methodologies for the implementation of the core structure-from-motion algorithms, but to use the more rapid and prototyping-oriented *Iterative Model* for other components that are of a more exploratory nature.

In order to facilitate the iterative development methodology and allow modifications to individual modules without affecting other components of the project, a high degree of modularisation was aimed for from the very start of the project. This was additionally supported by the choice of C++ as the main implementation language: for example, the data structures holding the annotations associated with individual video frames (see Section 3.1.2) are completely generic and independent of both the GUI front-end implementation and the structure-from-motion algorithms; in fact, it was even possible to integrate the KLT feature tracker with them without requiring any changes to the data structures.

This modularisation had the secondary advantage of providing inherent protection (further improved by appropriate use of the C++ language facilities for protection such as classes, namespaces and access modifiers) – an important property in a project of this size with numerous concurrently executing components.

Finally, adherence to standard good-practice software engineering principles such as a high degree of cohesion of components, structured and defensive programming and appropriate graceful handling of exception conditions was actively pursued.

## 2.7 Summary

In this chapter, I have summarised the work undertaken prior to implementing the project. A brief introduction to the underlying concepts of 3D structure-from-motion as well as the particular approach chosen for this project was provided. Due to the extent of reliance on existing tools and libraries for this project, some detail was given about the choices made here, and the development methodology was described.

The following chapter will provide detailed insight into the implementation pursued and elaborate on the different components of the project and their integration.

# Chapter 3

# Implementation

*So might I, standing on this pleasant lea,*
*Have glimpses that would make me less forlorn;*
*Have sight of Proteus rising from the sea.*

— William Wordsworth, sonnet, 1807

This chapter describes the implementation of the interactive structure-from-motion framework that was described in the previous section. The overall system architecture and the major individual components are considered.

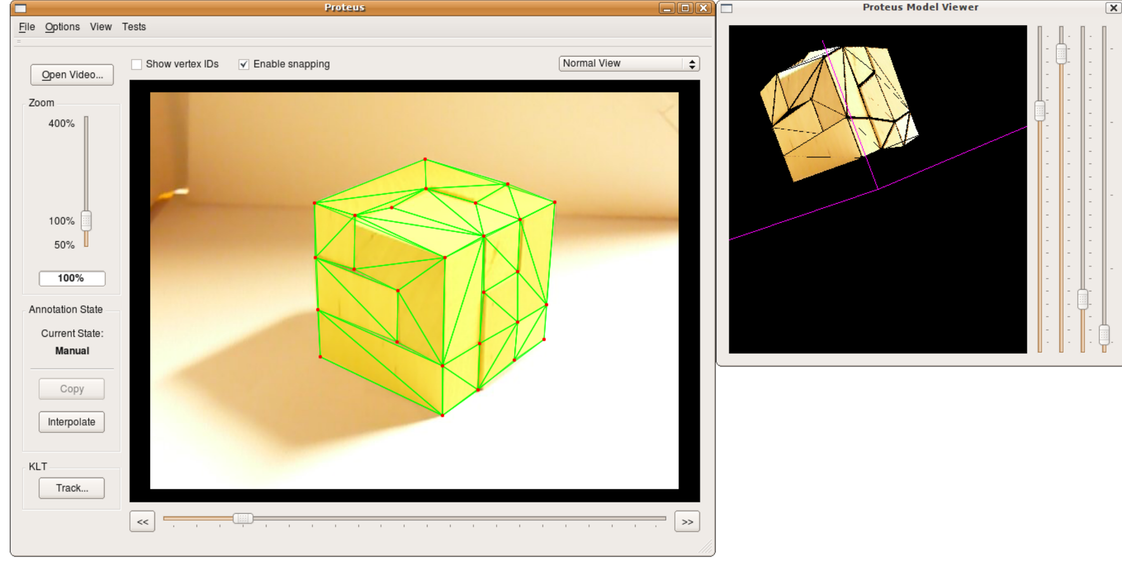



*Figure 3.1: A typical session in the* Proteus *user interface; the video annotator is shown on the left, and the generated model on the right.*

All components were implemented successfully and together form a fully working interactive 3D structure-from-motion system. A sample view of the system while in use is shown in Figure 3.1.

As outlined in the Preparation chapter, implementing this project was a large task, both in terms of complexity and in terms of scale: *Proteus* interfaces with eight different libraries and the code size of the C++ implementation alone exceeds 5,000 lines.

Hence in this chapter I will concentrate on giving a general architectural overview, as well as highlighting difficulties I encountered and decisions I made, but not elaborate on low-level implementation details. A more detailed description of the implementation will be given for certain key concepts and algorithms.

The key components of the project which are described in this chapter are:

1. **Video Decoding**
   Outlines the facility to load videos into the *Proteus* frontend.

2. **Data Structures**
   Describes the data structures used to hold annotation data displayed on the front-end and used within the SfM pipeline and those representing 3D model components.

3. **Front-end Interface**
   Introduces the front-end that is used to interactively annotate video frames and describes the guidance techniques implemented.

4. **Structure-from-Motion Pipeline**
   Gives an overview of the core algorithms used to reconstruct a 3D model from an annotated video sequence.

5. **Texture Extraction**
   Extends the SfM pipeline by showing the approach used to produce fully-textured output models.

6. **OpenGL Model Viewer**
   Briefly introduces the internal OpenGL viewer developed for the front-end.

7. **Kanade-Lucas-Tomasi Feature Tracker**
   Presents the KLT feature tracker and summarises how it was integrated with the project to provide a facility for fully automated feature tracking.

8. **Mesh Export**
   Explains the PLY mesh format and how the optional extension of allowing models to be saved this format was implemented.

9. **Serialization of Annotation Data**
   Elaborates on the additional extension of serialization support that I implemented to aid development.

## 3.1 System Architecture

### 3.1.1 Video Decoding

The video decoding component was the first part of the project to be implemented. Because the decoded video content would be too large to be buffered in memory in its entirety, decoding proceeds "on-the-fly" as the user navigates through the video. Only a single frame worth of raw pixel data is ever held in memory at a time. Since there is a some delay in decoding a video frame on-demand even on fast machines, I chose a multi-threaded implementation in order to keep the user interface responsive while decoding.

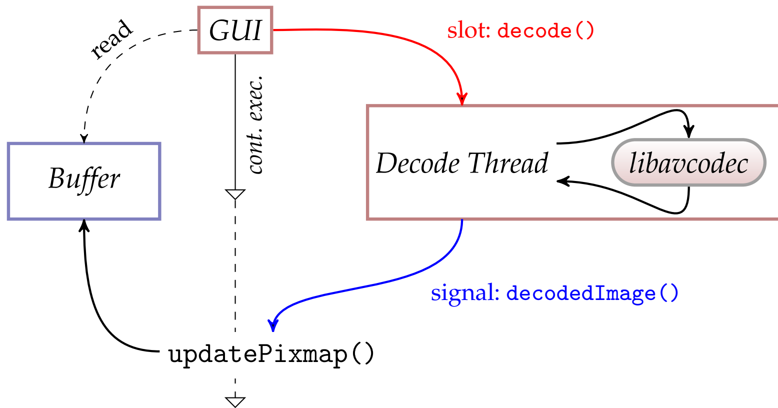



*Figure 3.2: Schematic overview process of decoding a video frame.*

The core of the video decoding components is the *decoder thread*. The GUI requests a new video frame to be decoded by asynchronously calling a function in the decoder thread. This will cause the decoder thread to request the appropriate frame from *libavcodec*. Once the frame has been decoded, the image is emitted with a Qt signal (`decodedImage`), triggering an update of the front-end display buffer.

The test videos used and their exact parameters are shown in Appendix A.8.

### 3.1.2 Data Structures

#### Annotation Data Structures

The identification and matching of features in the input material is the first step of any structure-from-motion implementation. These elementary data, which are a fundamental requirement for the rest of the structure-from-motion pipeline, must be stored in a data structure that supports a number of crucial properties:

- **Ubiquity**: All parts of the structure-from-motion pipeline that need to access feature correspondence data and all means of populating it should be able to use the same data structure.

- **Rapid Access**: Quick access should be provided not only to features local to the current image, but also to matched features in other frames.
- **Simplicity**: The implementation should be simple enough that it is intuitive to understand and can be undertaken in a reasonable amount of time.
- **Flexibility**: It would be advantageous to support different types of features (vertices, edges, triangles) that can be defined in terms of composition of each other (e.g. a triangle can be defined in terms of 3 vertices or 3 edges).
- **Extensibility**: If possible, it would be useful if other geometric primitives could be added without requiring modification to the fundamental representation.

It is clear that some of these desired properties (such as simplicity and rapid access) are at least partly orthogonal and some trade-offs will have to be made. Since none of the canonical data structures provided by standard libraries seemed especially suited for this, I designed a custom data structure, which is explained in the following and shown in Figure 3.3.

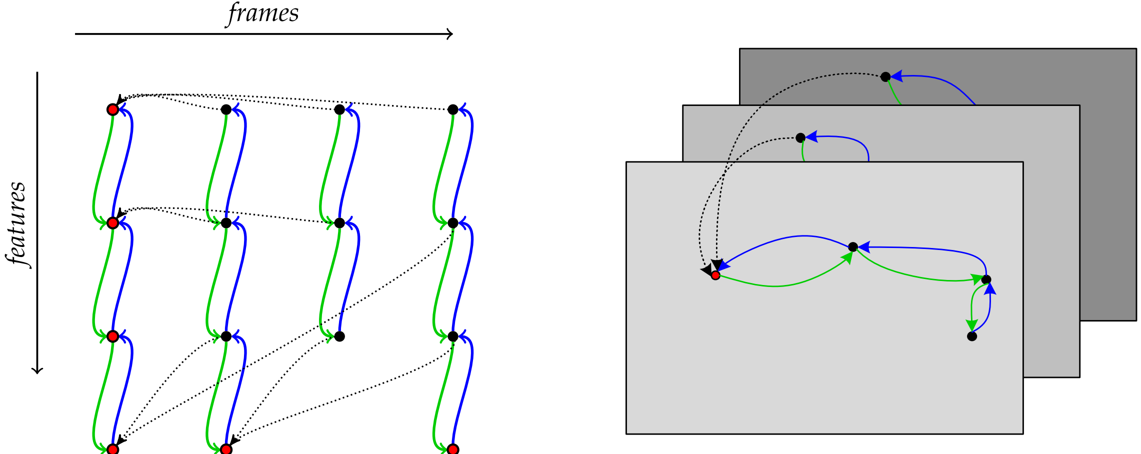



*Figure 3.3: The annotation data structure used: left – representation in memory, right – schematic illustration on a set of frames. Red nodes are base nodes,* i.e. *the first occurence of an annotation.*

This can, roughly, be seen as a three-dimensional net of pointers: On the horizontal axis, different frames are shown, and downwards on the vertical axis within a column, a doubly-linked list of annotation features is built. Cross-references to the first occurence of a feature are created by the `base` pointers (represented by dotted lines in Figure 3.3). These allow for rapid inter-frame access to annotation data.

The three different kinds of features supported – vertices, edges and polygons – all use the same universal paradigm (`previous`/`next`/`base` pointers), achieving the goal of ubiquity. The implementation is sufficiently simple that all necessary operations were easy to implement, and extension to new kinds of feature primitives is straightforward.

A sacrifice made to achieve these goals is access speed – linked lists are $O(n)$ in most operations; however, the performance impact of this complexity turned out to be acceptable even for large numbers of features (see detailed measurements in Section 4.3.1).

**3D Data Structures**

The data held in the annotation data structures is purely two-dimensional. The structure-from-motion pipeline (see Section 3.3) returns a 3-vector for each 3D model point inferred. However, in order to provide useable input to the OpenGL renderer (see Section 3.5) or the PLY exporter (see Section 3.7.2), more elaborate data structures representing 3D primitives and their properties, such as colour, texture and composition from elementary primitives, were required.

For this reason, I designed a number of simple data structures to represent objects in 3D space:

- `ThreeDPoint` – This represents a *vertex* in 3D space; consequently it has three fields corresponding to the $(x, y, z)$ coordinates and a `colour` attribute that is a RGB triple. It is drawn using the OpenGL `GL_POINTS` primitive.
- `ThreeDEdge` – Represents an *edge*, that is a line connecting two vertices, in 3D space. Its fields are two pointers to the start and end vertices represented as `ThreeDPoints`. This is drawn using the `GL_LINES` primitive.
- `ThreeDTriangle` – This is the most elaborate 3D primitive supported, and a special case of a general polygon in 3-space. It is defined in terms of three `ThreeDPoint` vertices, and drawn as a `GL_TRIANGLES` primitive.

`ThreeDTriangle` is the only primitive that can be textured – however, the texture and the texture coordinates are specified separately, as the texture extraction step (see Section 3.4) is independent of the generation of 3D primitives and OpenGL also sets up textures in a step separate to drawing primitives.

## 3.2 Annotation Interface

This section describes the graphical user interface of the *Proteus* system. It will go into some detail on the different modes of operation and interactive guidance features offered by the user interface when annotating video frames. A screenshot with annotations highlighting the important features of the user interface is shown in Figure 3.4 overleaf.

### 3.2.1 Modes of Operation

The annotation interface has two principal modes of operation which can be used individually or in combination: *Manual annotation* and *automated feature tracking*.

When using *manual annotation*, features such as corners, edge joints, or distinctive points in the object texture, are identified and matched by the user by clicking and dragging. This process can be supported by interactive guidance through edge snapping (see Section 3.2.3). Since complete annotation with a suitable number of features is quite a tedious task to undertake for multiple subsequent frames, several options are offered to make this step easier for the user:

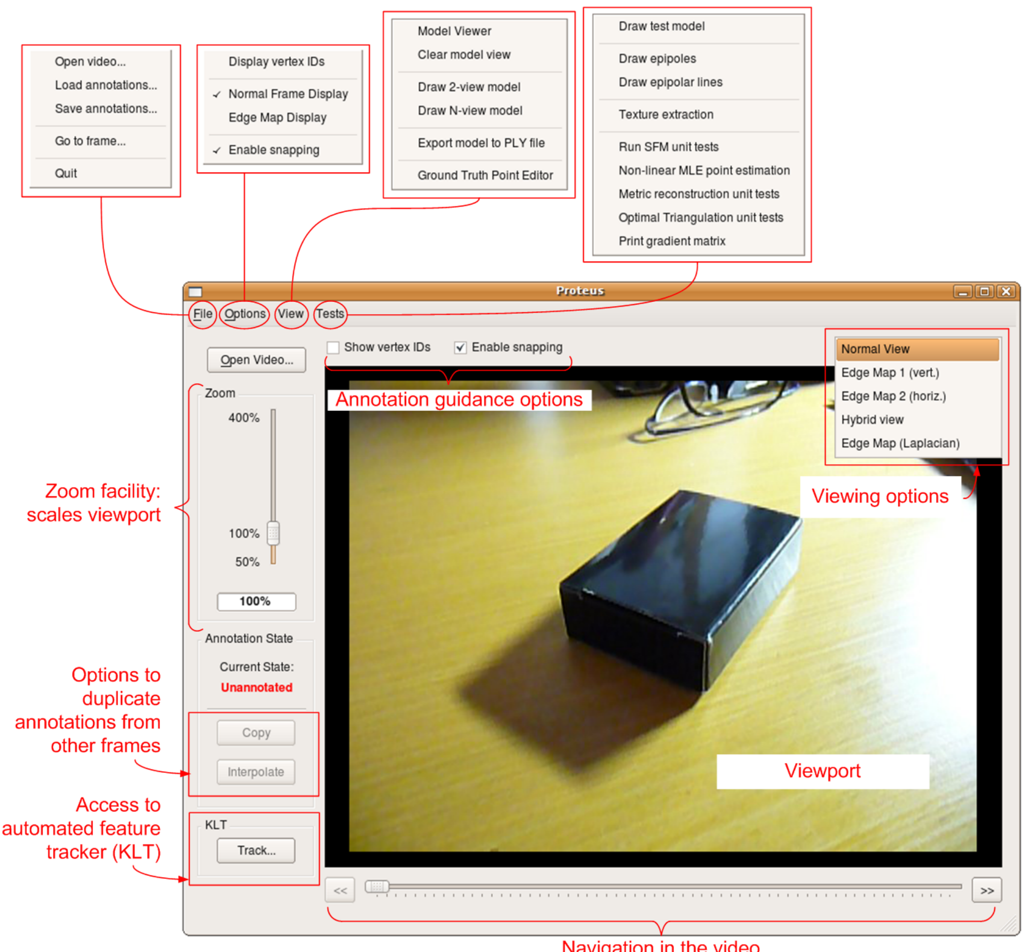



*Figure 3.4: Overview of the front-end user interface.*

- The complete annotation lists from a different frame can be **copied**. This will almost always result in the annotations ending up displaced from the feature points, so a step of manually moving the annotation points to the correct locations should follow.

- Annotations can be **interpolated** over a sequence of unannotated frames between two annotated key frames, assuming a simple linear translational camera motion in between.

The second approach, *automated feature tracking*, invokes the Kanade-Lucas-Tomasi feature tracker, which identifies a specified number of features in the first frame and tracks them for a specified number of frames. This only produces a set of matched feature points, and should normally be followed by a step of manual insertion of edges and triangles.

### 3.2.2 Adding Annotations

The process of adding annotations is carried out in an intuitive *point and click* manner: The user initiates the process of adding an annotation by clicking on a point in the image that is supposed to be an initial feature point. A red dot will appear, and a yellow dot and line will show the next feature point and the line connecting the two (see Figure 3.5). By clicking again, the next feature point can be placed.

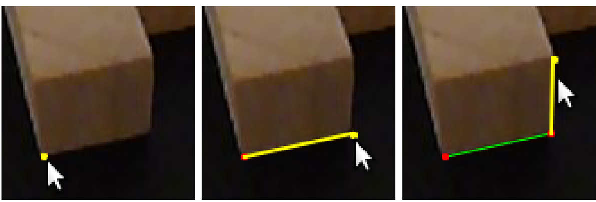
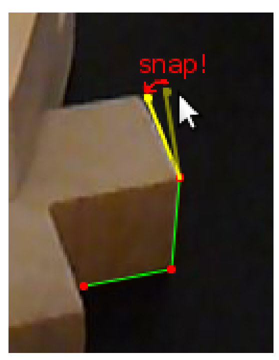


*Figure 3.5: Creation of two annotations connected by an edge: click, point, click.*

### 3.2.3 Interactive Guidance using "Snapping"




One of the most difficult and tedious parts of the process of manually annotating a video frame for a human user is to place annotations with pixel accuracy. Hence some way of guiding the user in a non-intrusive, interactive fashion would be useful.

A very similar problem has been described by Agarwala [1], concerning the annotation of video material for generating cartoons. There, an approximate tracing provided by the user is fitted to contours found in the image. I decided to adapt this for the *Proteus* project: The projected line during the annotation creation phase should "snap" to distinctive edges in the image, making it easy to trace the outlines of objects.

#### Outline of Snapping

The first step to be performed in order to implement snapping is the identification of edges in the image, which is done by applying an *edge detection operator*. An algorithm, described later in this section (see Algorithm 1), is run to work out the *snapping score* for the vicinity of the mouse cursor and allows annotations to snap to edges accordingly.

In addition to "snapping" to edges in the image, there is also *endpoint snapping*, which means that new annotations will snap to existing annotation points. The underlying assumption is that if the user is placing a new feature point very close to an existing one, it is likely that they meant to merge the two. Endpoint snapping takes precedence over edge snapping, but only considers a smaller window around the cursor.

Of course, there is a chance that the user does not want either form of snapping to happen – for this situation, there are both UI controls and a key combination (holding the CTRL key down whilst placing annotations) that can temporarily disable the snapping mechanisms.

### Choice of Edge Detection Operator

There is a number of edge detection operators that are commonly used in computer vision applications. These are usually classified in terms of being *first-order difference* or *second-order difference* operators, and the canonical way of applying them is to perform a 2D discrete convolution of the image with the operator.

The discrete form of convolution replaces integration by summation, and hence is expressed by

$$h(x, y) = \sum_{\alpha} \sum_{\beta} f(\alpha, \beta) \cdot g(x - \alpha, y - \beta),$$

where $f$ is the $\alpha \times \beta$ operator, $g$ is the 2D image and $h$ is the resulting image.

The operators considered for this project were the (second-order) *Laplacian* edge detection operator and the (first-order) *Sobel* operators [23], where the former is an isotropic second-order finite difference operator and the latter are directional first-order derivative operators.

The operation of discrete convolution can sometimes be implemented more efficiently in the Fourier domain than in the image domain, depending on the relative sizes of the image and the operator considered. In this case, the kernel size is $3 \times 3$, so this does not apply.

### Laplacian

A discrete approximation of the Laplacian operator $\nabla^2$ can be used to perform edge detection by detecting the zero-crossings of the gradient in convolution with the image.

However, there are numerous complications with the Laplacian operator, most prominently its complexity, the suboptimal performance of the difference-taking approach and its inherent noise sensitivity.

I initially implemented edge detection using the Laplacian operator, but decided against using it as the default edge detection method when it became clear that its noise sensitivity led to suboptimal detection. Instead, the Sobel operators were considered as an alternative. The implementation using the Laplacian is described in Appendix A.3.

### Sobel operators

The Sobel operators are a family of directional (*non-isotropic*) convolution kernels that separately detect horizontal and vertical edges:

| | | |
|---|---|---|
| −1 | 0 | 1 |
| −2 | 0 | 2 |
| −1 | 0 | 1 |

(a) $\frac{\partial}{\partial x}$

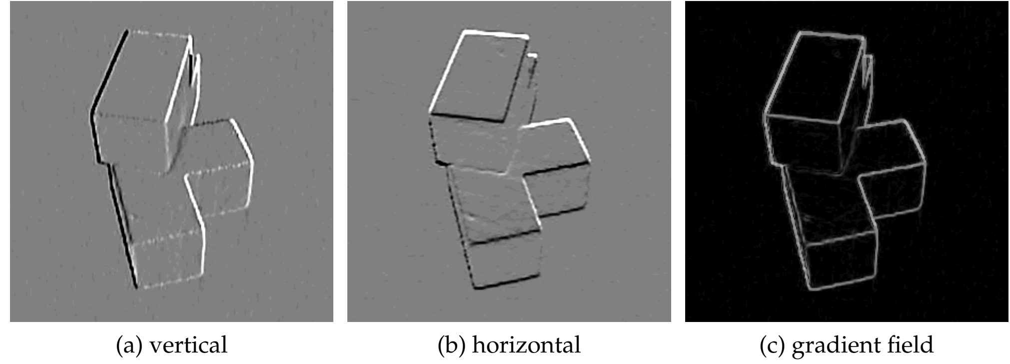
| | | |
|---|---|---|
| −1 | −2 | −1 |
| 0 | 0 | 0 |
| 1 | 2 | 1 |

(b) $\frac{\partial}{\partial y}$

These kernels each perform discrete differentiation (difference) in one direction and integration (summation) in the other. Furthermore, they are a $3 \times 3$ approximation to the first-

derivative-of-Gaussian kernels. An advantage of this is that it provides an inherent Gaussian blurring, making the operators less susceptible to noise.

The first kernel will filter for vertical edges (derivative with respect to $x$), whilst the second filters for horizontal edges. The resulting edges will have a *polarity*, indicating the (one-dimensional) direction of the edge normal. However, some edges that are at an angle will be detected by both operators, sometimes with opposite polarities (e.g. the top-left edge in the example in Figure 3.6).



(a) vertical (b) horizontal (c) gradient field

*Figure 3.6: Application of the Sobel edge detection operators: (a) vertical Sobel operator applied, (b) horizontal, and (c) the values of* $\mathbf{G}$, i.e. *the gradient field.*

It should be noted that with this operator, both very large and very small (negative) numbers indicate a high gradient. This is a result of the polarity property, and when combining the results of the two operator applications it needs to be ensured that opposite signs do not cancel each other out.

For this reason, the gradient approximations resulting from applying the two operators are combined into a gradient magnitude using the square root of their squared sum,

$$\mathbf{G} = \sqrt{\mathbf{G}_x{}^2 + \mathbf{G}_y{}^2}.$$

**Edge Detection Metric**

The *gradient field* – an array of per-pixel gradient values returned from the application of an edge detection operator – is pre-computed and stored in a dynamically-sized `multi_array` provided by the `boost` library (this choice was made so that the image dimensions would not have to be fixed, as would have been required with "normal" multi-dimensional C++ arrays).

In order to use this data to snap to edges when adding annotations, the basic assumption made was that the user will be pointing somewhere in the vicinity of the desired position. In other words, only a fixed window around the mouse cursor position on the image needs to be considered, and it can be assumed that this window contains one end of the edge. I chose to use a $20 \times 20$ pixel window, as this provides a reasonable trade-off between the captured range and the computational complexity.

Let $h(x, y)$ denote the 2D function representing the gradient field in the following. A plot of an example of this function is shown in Figure 3.7.

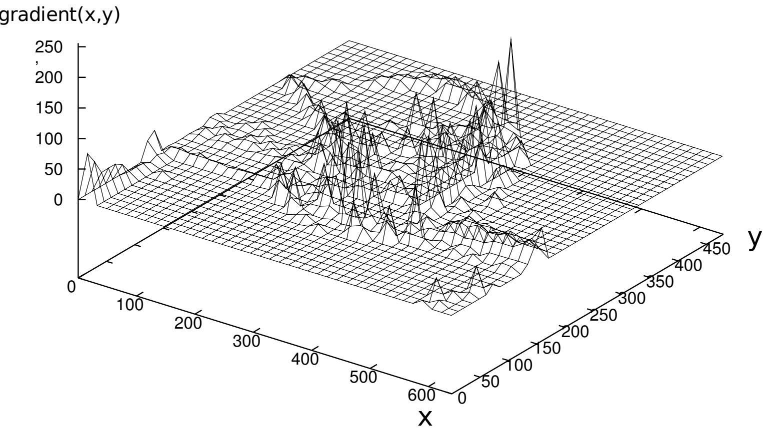



*Figure 3.7: Plot of the gradient field for a frame of the "Cube" example.*

For each of the pixels within the snapping window, a hypothetical line is constructed using Bresenham's line drawing algorithm [6]. The method used to find a metric respresenting how likely a given hypothetical line is to be on, or close to, an edge is as follows (Algorithm 1).

Objective: For a given cursor position $C = (x, y)$, find the most optimal pixel $S$ in a $k \times k$ window (*snapping window*) to snap a new line endpoint to.

Algorithm:

1. **Initialisation**: Set $S = C$ and $max = 0$.
2. **Iteration**: For each pixel $P_{ij} = (x_i, y_j)$ in the $k \times k$ window,
   (a) Construct a hypothetical line with endpoint $(x_i, y_j)$ using *Bresenham's line drawing algorithm*.
   (b) Calculate the *snapping score*, the discrete line integral $I$ over the gradient values along this line, and normalise it by the line length, giving $I_{\text{norm}}$.
   (c) If the value of $I_{\text{norm}}$ exceeds $max$ and it also exceeds a pre-defined significance threshold, then set $S = P_{ij}$ and $max = I_{\text{norm}}$.
3. **Termination**: Return the pixel coordinates of $S$.

*Algorithm 1: Outline of the algorithm used for edge snapping.*

For a hypothetical line $\mathbf{r}$ of length $n$ pixels, a summation of the form

$$I = \sum_{i=0}^{n} h(\mathbf{r}(x_i, y_i))$$

can be used as an approximation to the line integral. This can be derived from the line integral using a *Riemann sum* (see Appendix A.4). The value of this sum can be obtained while

the line is being constructed by summing the values of $h(x,y)$ for all pixels $x,y$ along the line.

The value of the line integral is *normalised* by the length of the projected line, giving an average per-pixel gradient value. This value can be used as a score for the hypothetical line, aiding in finding the optimal line trajectory.

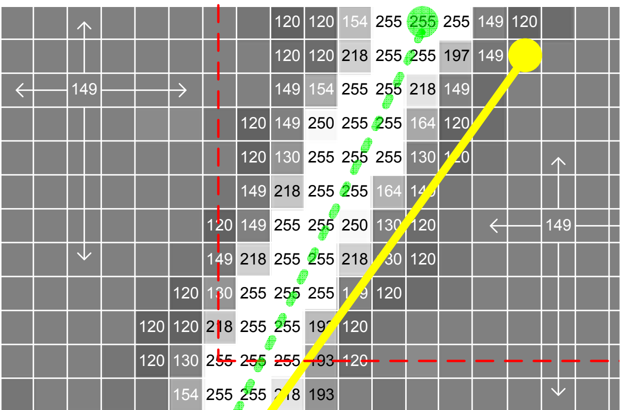



*Figure 3.8: Schematic view of the snapping calculations. The dashed red line is the boundary of the snapping window, the green line is the line corresponding to the optimal snapping score, the yellow line is where the user is pointing.*

### 3.2.4 Adding Surfaces

The process of inserting a surface into an annotated frame is equally straightforward to adding the initial annotations, and uses a similar interactive technique: when in surface insertion mode, the user can click a set of points (which may be existing or new feature points) in order to insert a triangle which has its boundaries defined by those points. Figure 3.9 shows an example of how this process works from the user's perspective, once again using the *click, point, click* paradigm that was already used for insertion of feature points.

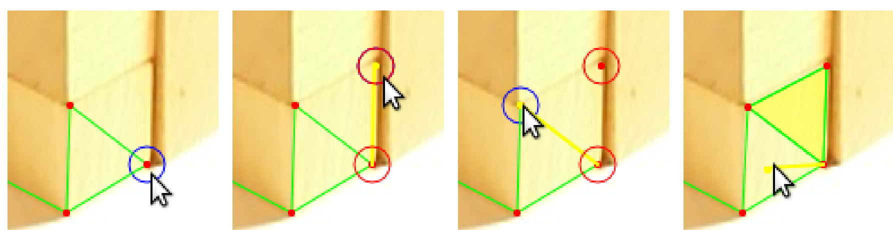


*Figure 3.9: Creating a new triangular surface in three steps; note how existing and new feature points are combined to form the vertices of the triangle.*

Guidance for the user is provided as the front-end highlights the feature points that have already been included in the surface, indicated by a red circle. Additionally, candidate points are highlighted with a blue circle as the mouse cursor moves into their vicinity.

## 3.3 The Structure-from-Motion Pipeline

The structure-from-motion implementation is the heart of the *Proteus* project and the part that presented the greatest technical challenge, for reasons described before (see Sections 1.2, 2.3 and 2.4).

Due to the linear nature of data being processed by subsequent algorithms, aiming to finally obtain a set of 3D coordinates suitable for construction of a 3D model, this was implemented in a pipelined fashion: each of the algorithms builds on the previous one and its output constitutes part of the inputs to the next one (see Figure 3.10 for a high-level view).

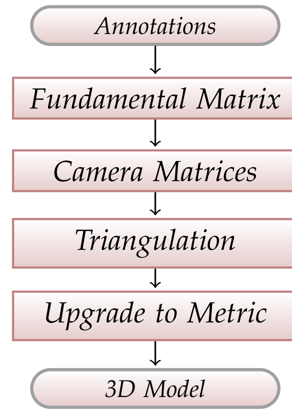



*Figure 3.10: The structure-from-motion pipeline.*

### 3.3.1 Computing the Fundamental Matrix

**Introduction – The Normalised 8-point Algorithm**

The first stage of the structure-from-motion pipeline is the computation of the *fundamental matrix*, $\mathtt{F}$. This matrix is an algebraic representation of epipolar geometry as a unique $3 \times 3$, rank 2 homogeneous matrix with 7 degrees of freedom[1]. Its property is that it maps a point $\mathbf{x}_i$ to the corresponding epipolar line $\mathbf{l}'_i$ in the other image, *i.e.* we have $\mathbf{l}'_i = \mathtt{F}\mathbf{x}_i$. This can be derived from the relations of epipolar geometry either algebraically or geometrically by point transfer via a 2D homography [12, p. 243–ff.].

The central property of the fundamental matrix is the **correspondence condition**: for any pair of corresponding points $\mathbf{x}_i \leftrightarrow \mathbf{x}'_i$ in two images, $\mathbf{x}'_i$ lies on the epipolar line $\mathbf{l}'_i$ in the second image, and we must have $\mathbf{x}'^{\mathsf{T}}_i \mathbf{l}'_i = 0$ and hence (since $\mathbf{l}'_i = \mathtt{F}\mathbf{x}_i$),

$$\mathbf{x}'^{\mathsf{T}}_i \mathtt{F}\mathbf{x}_i = 0.$$

The simplest algorithm for finding the fundamental matrix is the *normalised 8-point algorithm* [12, p. 281–282], summarised in Algorithm 2. I chose to implement this algorithm due to its relative simplicity compared to the alternatives – however, there are a number of degenerate cases in which it will not yield a unique or a high-quality solution.

**Singularity Constraint**

It is very important to ensure that the fundamental matrix is a singular matrix of rank 2. If this is not the case, the epipolar geometry relations represented by it will not be accurate and the epipolar lines $\mathbf{l}' = \mathtt{F}\mathbf{x}$ for different $\mathbf{x}$ will not coincide in one point (see Figure 3.11).

In order to enforce this constraint, the matrix $\mathtt{F}$ found by solving the simultaneous equations

[1] The central role of the fundamental matrix in epipolar geometry has even resulted in the dedication of a song – see *"The Fundamental Matrix Song"* by Daniel Wedge, `http://danielwedge.com/fmatrix/`

Objective: For a given set of $n$ image point correspondences in two images $\mathbf{x}_i \leftrightarrow \mathbf{x}'_i$, where $n \geq 8$, determine the fundamental matrix $\mathtt{F}$, subject to the correspondence condition $\mathbf{x}'^{\mathsf{T}}_i \mathtt{F} \mathbf{x}_i = 0$.

Algorithm:

1. **Normalisation:** The coordinates of the feature points are transformed according to normalising transformations $\mathtt{T}$ and $\mathtt{T}'$. Both of these scale and transform the points such that
   (a) their centroid is at the origin, and
   (b) their root mean square distance from the origin is $\sqrt{2}$.

   Hence the points considered in the next step are $\hat{\mathbf{x}}_i = \mathtt{T}\mathbf{x}_i$ and $\hat{\mathbf{x}}'_i = \mathtt{T}'\mathbf{x}'_i$.
2. **Finding the fundamental matrix** $\hat{\mathtt{F}}'$: This is a two-step process:
   (a) **Linear solution**: Build an $n \times 9$ matrix $\mathtt{A}$ that is composed from the matches $\hat{\mathbf{x}}_i \leftrightarrow \hat{\mathbf{x}}'_i$ and apply the SVD to find the column corresponding to the smallest singular value, giving a 9-vector that represents $\hat{\mathtt{F}}'$ in row-major order.
   (b) **Singularity constraint enforcement**: Constrain $\hat{\mathtt{F}}$ to be of rank 2 by minimising the Frobenius norm $\|\hat{\mathtt{F}} - \hat{\mathtt{F}}'\|$ for $\det \hat{\mathtt{F}}' = 0$ by taking the SVD of $\hat{\mathtt{F}} = \mathtt{UDV}^{\mathsf{T}}$ and building $\hat{\mathtt{F}}'$ as $\mathtt{U} \cdot \mathrm{diag}(r, s, 0) \cdot \mathtt{V}^{\mathsf{T}}$ where $r$ and $s$ are the two largest singular values of the SVD of $\hat{\mathtt{F}}$.
3. **Denormalisation**: The normalisation of step 1 is reversed by computing $\mathtt{F} = \mathtt{T}'^{\mathsf{T}} \hat{\mathtt{F}}' \mathtt{T}$.

*Algorithm 2: The normalised 8-point algorithm for finding the fundamental matrix, adapted from [12, p. 281–ff.].*

in the matrix $\mathtt{A}$ is corrected so that its corrected version $\mathtt{F}'$ minimises the *Frobenius norm*

$$\|\mathtt{F} - \mathtt{F}'\| = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |f_{ij} - f'_{ij}|^2},$$

subject to the condition that $\det \mathtt{F}' = 0$, *i.e.* that $\mathtt{F}'$ is rank-deficient. The recommended method for this is to take the SVD of the initial fundamental matrix, $\mathtt{F} = \mathtt{UDV}^{\mathsf{T}}$. Since $\mathtt{D}$ is by definition a $3 \times 3$ diagonal matrix, we can extract its two largest components: say $\mathtt{D} = \mathrm{diag}(r, s, t)$ where $r \geq s \geq t$. Using these, the fundamental matrix is re-assembled using $\mathtt{F}' = \mathtt{U} \cdot \mathrm{diag}(r, s, 0) \cdot \mathtt{V}^{\mathsf{T}}$.

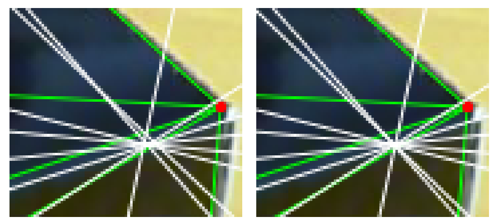


*Figure 3.11: Illustration of the effect of enforcing the singularity constraint (magnified); non-singular fundamental matrix* $\mathtt{F}$ *on the left, singular* $\mathtt{F}$ *on the right.*

**Implementation**

A crucial challenge in the implementation of this algorithm was to find a technique for solving the system of $n$ simultaneous equations introduced by the matrix A. In the general case, a linear solution is not possible, so a least-squares solution was pursued instead. Hartley and Zisserman suggest using the method of *Singular Value Decomposition* [21, p. 301–ff.], and specifically of finding the singular vector corresponding to the smallest singular value.

The *QuantLib* library provides a class, `SVD`, to perform the SVD of a matrix, which, importantly, provides methods to access the matrices `U`, `D` and `V` that make up the SVD.

The implementation of the singularity constraint enforcement was relatively straightforward since it only amounted to taking the SVD again, extracting the two largest singular values from `D` and rebuilding the matrix using the newly created diagonal matrix instead of `D`.

### 3.3.2 Finding the Camera Matrices

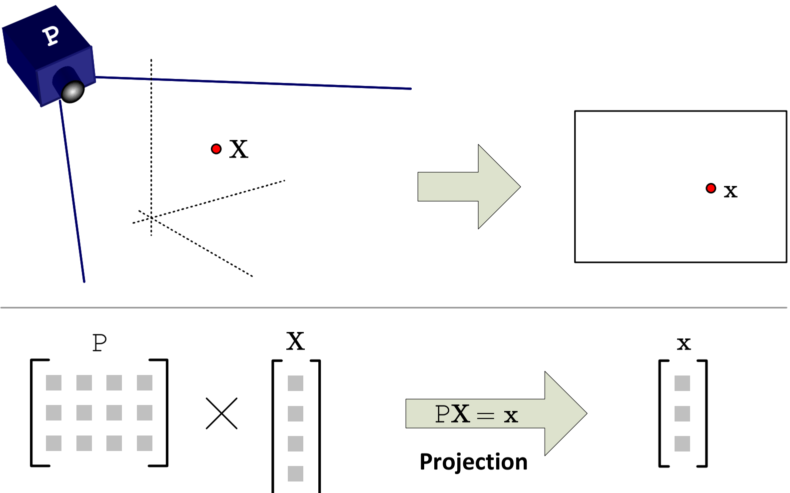



*Figure 3.12: Illustration of the role of the camera matrix in the projection process.*

After the fundamental matrix has been found, the next step is to extract the camera matrices. The significance of a $3 \times 4$ camera matrix is that it provides an algebraic way to perform a projection: For a 3D point $\mathbf{X}$, specified as a $4 \times 1$ vector, and a camera matrix P, it is always the case that $\mathbf{x} = \mathtt{P}\mathbf{X}$, where $\mathbf{x}$ is the 2D projection of $\mathbf{X}$ into the image plane of the camera C giving rise to P. This relationship can be seen more clearly in Figure 3.12.

The camera matrix also provides information about the position and orientation of the camera in 3-space, which is relevant for the following stages of the structure-from-motion process. It is, however, important to note that a pair of camera matrices uniquely determines a fundamental matrix, but that the converse is not true: due to projective ambiguities, any fundamental matrix can correspond to multiple sets of camera matrices.

The precise composition of the camera matrix depends on the camera model used and has been the subject of some extensive literature [12, p. 152–ff.] and discussing this in detail would be outside the scope of this dissertation. The relevant models for the purposes of this

project were the *projective* and *metric* cameras and, correspondingly, camera matrices. The determination of metric camera matrices is described further in the context of upgrading the reconstruction to metric in Section 3.3.4.

For a fundamental matrix $\mathtt{F}$, the corresponding camera matrices are given by

$$\mathtt{P} = [\mathtt{I} \mid \mathbf{0}] \qquad \text{and} \qquad \mathtt{P}' = [[\mathbf{e}']_{\times}\mathtt{F} \mid \mathbf{e}'],$$

where $\mathtt{I}$ denotes the $3 \times 3$ identity matrix, $\mid$ denotes matrix concatenation, $\mathbf{0}$ is a zero 3-vector and the notation $[\mathbf{e}']_{\times}$ is the skew-symmetric matrix

$$[\mathbf{e}']_{\times} = \begin{bmatrix} 0 & -e'_3 & e'_2 \\ e'_3 & 0 & -e'_1 \\ -e'_2 & e'_1 & 0 \end{bmatrix}.$$

The implementation of this was straightforward using the standard classes and linear algebra routines provided by *QuantLib*. I chose to use contraction with the *epsilon tensor* $\varepsilon_{ijk}$ [12, p. 563] as a means of implementing the generalised cross product between the 3-vector corresponding to the epipole $\mathbf{e}'$ and the fundamental matrix $\mathtt{F}$.

### 3.3.3 Triangulation of Points

In general, the process of triangulation in two or three dimensions describes a method to find the position of the third vertex of a triangle, knowing only the other two vertex positions and the respective angles from the baseline connecting them.

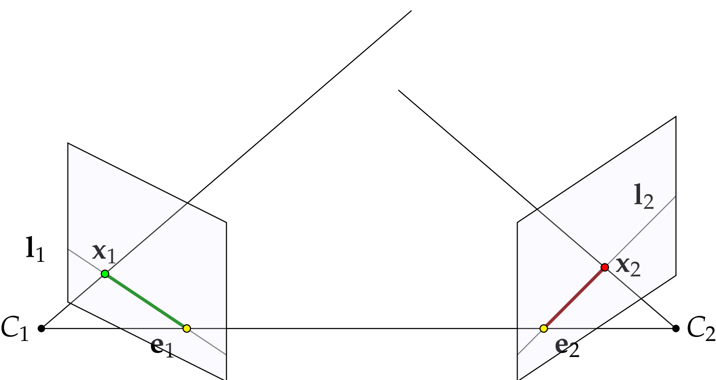



*Figure 3.13: Illustration of the problem of projection rays missing each other in 3D space.*

Naïve geometric triangulation assumes that for an existing triangle there is always an intersection point of the two lines that connect in the third vertex – this is true in two dimensions, but obviously not in three where rays can miss each other in space. As outlined before, with noisy input data to the structure-from-motion pipeline, this is the default case rather than an exception (see Figure 3.13 for an example). Hence the algorithms described in the following aim to minimise the *distance* between the projection rays in order to find the best approximation to the intersection point.

I implemented two algorithms for triangulation: The simple *homogeneous Direct Linear Transformation (DLT) method* and the *optimal triangulation algorithm* based on minimisation of a

Objective: Given a set of $n = j - i$ point correspondences across $n$ images of the form

$$\mathbf{x}_i \leftrightarrow \mathbf{x}_{i+1} \leftrightarrow ... \leftrightarrow \mathbf{x}_{j-1} \leftrightarrow \mathbf{x}_j$$

and a set of $n$ camera matrices $\mathtt{P}_i, ..., \mathtt{P}_j$, find the 3D point $\mathbf{X}$ corresponding to the $\mathbf{x}_i, ..., \mathbf{x}_j$ such that the algebraic distance between the projection rays is minimised.

Algorithm:

1. **Preconditioning**: A homography $\mathtt{H}$ that scales the 2D points $\mathbf{x}_i, ..., \mathbf{x}_j$ by $\frac{1}{2}$ of the image size and translates them by $-1$ in each direction is constructed and applied, giving $\hat{\mathbf{x}}_i, ..., \hat{\mathbf{x}}_j$.
2. **System of simultaneous equations**: For each image point $\mathbf{x}_k$, it is the case that $\mathbf{x}_k = \mathtt{P}_k\mathbf{X}$, which gives rise to two linearly independent equations per image. These equations form the $2 \times 12$ matrix $\mathtt{A}_k$. The different $\mathtt{A}_k$ are combined into a $2n \times 12$ matrix $\mathtt{A}$ such that $\mathtt{A}\mathbf{X} = \mathbf{0}$.
3. **Solution**: Solve by computing the SVD of $\mathtt{A}$ and choosing the smallest singular value as $\mathbf{X}$.

*Algorithm 3: The homogeneous DLT algorithm for triangulation across n images, based on [12, p. 312].*

geometric error cost function. While the linear homogeneous algorithm was fairly straightforward to implement and gave quick results, allowing for verification of other parts of the pipeline, the optimal algorithm has a much greater implementation complexity, but provides provably optimal results [12, p. 313].

### The Homogeneous DLT Algorithm

The choice to implement the homogeneous DLT as the basic triangulation algorithm was made mainly because it is easily generalised from two to $n$ views (cf. [12, p. 312], which provides this for two views). The full adapted derivation that I developed for $n$ views can be found in Appendix A.7.

My implementation is described in Algorithm 3. The step of establishing the equations for the matrix $\mathtt{A}$ was implemented by contraction of $\mathbf{x}_k\mathtt{P}$ with an $\varepsilon$-tensor that was already used for the cross product in the computation of the camera matrices (see Section 3.3.2). This gives rise to two equations in two unknowns from each image in the correspondence, which can be expressed as a $2 \times 12$ matrix that is concatenated onto $\mathtt{A}$. As before, the SVD is used to solve the resulting system of linear equations.

In order to aid modularisation of the implementation, I implemented a number of linear algebra helper functions, the most prominent of which was the set of overloaded `submatrix` methods that allow an arbitrary subset of a *QuantLib* matrix to be extracted into a new one.

### The Optimal Triangulation Algorithm

The second triangulation algorithm implemented is the *optimal triangulation algorithm* [12, p. 313–ff.], an overview of which is given in Algorithm 4 (overleaf). The virtue of this algorithm is that it is invariant to the projective frame of the cameras (the homogeneous DLT

algorithm is only invariant to affine transformations, but not projective ones) and is less susceptible to noise in the input images than the linear algorithms.

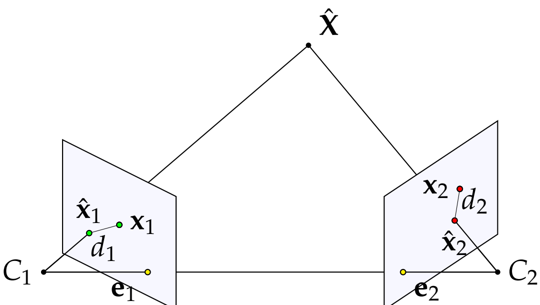



*Figure 3.14: The 3-space point $\hat{\mathbf{X}}$ projects to points $\hat{\mathbf{x}}_1$ and $\hat{\mathbf{x}}_2$. These satisfy the epipolar constraint, while the measured points $\mathbf{x}_1$ and $\mathbf{x}_2$ do not. The point $\hat{\mathbf{X}}$ should be chosen so that the reprojection error $d_1^2 + d_2^2$ is minimised.*

The underlying theoretical foundation stems from the fact that the feature correspondences identified in the images will not in general correspond to exact projections of the correct 3-space point, as illustrated in Figure 3.14. Noise and lack of precision in the sub-pixel region mean that the measured points $\mathbf{x}_1 \leftrightarrow \mathbf{x}_2$ for two views do not exactly satisfy the epipolar constraint (see Section 2.1). Hence this algorithm seeks to identify points $\hat{\mathbf{x}}_1 \leftrightarrow \hat{\mathbf{x}}_2$ that satisfy the epipolar constraint $\hat{\mathbf{x}}_2{}^\mathsf{T} \mathtt{F} \hat{\mathbf{x}}_1 = 0$ and still lie as close as possible to the measured points, *i.e.* minimise the geometric error function $\mathcal{C}(\mathbf{x}_i, \mathbf{x}_j) = d(\mathbf{x}_i, \hat{\mathbf{x}}_j)^2 + d(\mathbf{x}_i, \hat{\mathbf{x}}_j)^2$, subject to the epipolar constraint.

This minimisation can be performed in a variety of ways; Hartley and Zisserman present a Sampson approximation [12, pp. 314–315] and a non-iterative solution method that is provably optimal under the assumption of Gaussian noise.

The latter method was implemented for this project and is described in detail in Algorithm 4 and further, including the full derivation, by Hartley and Zisserman [12, p. 315–ff.].

A considerable challenge for the C++ implementation was posed by the solution of the polynomial in step 7 – since this is a sixth-degree polynomial, no general closed-form solution exists. A simple Newton-Raphson approximation was not deemed sufficient in this case since finding an appropriate initial guess is very difficult and cannot guarantee convergence for all (up to six, possibly complex) roots. I ended up using the complex polynomial root finding functions provided by the GNU Scientific Library[2], which successfully identify all roots.

A disadvantage of the optimal triangulation algorithm is that it does not straightforwardly generalise to more than two views. This is because some steps, particularly those to do with the cost function, i.e. steps 6–9, are inherently based on there being a pairwise correspondence. For this reason, it was only implemented for two-view reconstructions, and the homogeneous DLT algorithm was used for the *n*-view reconstruction code.

[2] GSL-1.12, `http://www.gnu.org/software/gsl/`

Objective: Given a point correspondence of the form $\mathbf{x}_i \leftrightarrow \mathbf{x}_j$ and a fundamental matrix $\mathtt{F}$,

a. first compute the corrected correspondence $\hat{\mathbf{x}}_i \leftrightarrow \hat{\mathbf{x}}_j$ such that the geometric error

$$\mathcal{C}(\mathbf{x}_i, \mathbf{x}_j) = d(\mathbf{x}_i, \hat{\mathbf{x}}_j)^2 + d(\mathbf{x}_i, \hat{\mathbf{x}}_j)^2$$

is minimised subject to the epipolar constraint $\hat{\mathbf{x}}_j{}^\top \mathtt{F} \hat{\mathbf{x}}_i = 0$,

b. and then find the 3D point $\mathbf{X}$ corresponding to $\hat{\mathbf{x}}_i \leftrightarrow \hat{\mathbf{x}}_j$.

Algorithm:

1. Define transformation matrices $\mathtt{T}_i$ and $\mathtt{T}_j$:

$$\mathtt{T}_i = \begin{bmatrix} 1 & & -x_i \\ & 1 & -y_i \\ & & 1 \end{bmatrix} \quad \text{and} \quad \mathtt{T}_j = \begin{bmatrix} 1 & & -x_j \\ & 1 & -y_j \\ & & 1 \end{bmatrix}$$

which takes $\mathbf{x}_i = (x_i, y_i, 1)^\top$ and $\mathbf{x}_j = (x_j, y_j, 1)^\top$ to the origin.

2. Translate $\mathtt{F}$ to the new coordinates by setting $\mathtt{F} = \mathtt{T}_j^{-\top} \mathtt{F} \mathtt{T}_i^{-1}$.

3. Find the epipoles for the cameras corresponding to the fundamental matrix such that $\mathtt{F}\mathbf{e}_i = \mathbf{0}$ and $\mathbf{e}_j{}^\top \mathtt{F} = \mathbf{0}$.

4. Form the rotation matrices $\mathtt{R}_i$ and $\mathtt{R}_j$ such that $\mathtt{R}_i \mathbf{e}_i = (1, 0, e_{i_3})^\top$ and $\mathtt{R}_j \mathbf{e}_j = (1, 0, e_{j_3})^\top$:

$$\mathtt{R}_i = \begin{bmatrix} e_{i_1} & e_{i_2} & \\ -e_{i_2} & e_{i_1} & \\ & & 1 \end{bmatrix} \quad \text{and} \quad \mathtt{R}_j = \begin{bmatrix} e_{j_1} & e_{j_2} & \\ -e_{j_2} & e_{j_1} & \\ & & 1 \end{bmatrix}$$

5. Redefine $\mathtt{F}$ again to be $\mathtt{F} = \mathtt{R}_i \mathtt{F} \mathtt{R}_j{}^\top$.

6. Let $f_i = e_{i_3}$, $f_j = e_{j_3}$, $a = F_{22}$, $b = F_{23}$, $c = F_{32}$, $d = F_{33}$.

7. For the polynomial $g(t)$, defined as

$$g(t) = t((at+b)^2 + f_j^2(ct+d)^2)^2 - (ad-bc)(1+f_i^2t^2)^2(at+b)(ct+d) = 0$$

find solutions for $t$, obtaining up to 6 real or complex roots.

8. Evaluate the cost function

$$s(t) = \frac{t^2}{(1+f_i^2t^2)^2} + \frac{(ct+d)^2}{(at+b)^2 + f_j^2(ct+d)^2}$$

at the real part of the each of roots. Select the minimum value $t_{\text{min}}$.

9. Evaluate the two epipolar lines $\mathbf{l}_i = (tf, 1, -t)$ and $\mathbf{l}_j = (-f_j(ct+d), at+b, ct+d)^\top$ at $t_{\text{min}}$ and find the closest points to the origin on these lines:
These points are $\hat{\mathbf{x}}_i$ and $\hat{\mathbf{x}}_j$.

10. Transfer these back to the original coordinates by setting $\hat{\mathbf{x}}_i = \mathtt{T}_i^{-1} \mathtt{R}_i{}^\top \hat{\mathbf{x}}_i$ and $\hat{\mathbf{x}}_j = \mathtt{T}_j^{-1} \mathtt{R}_j{}^\top \hat{\mathbf{x}}_j$.

11. Use these points as input to the homogeneous DLT algorithm and obtain the 3D point $\hat{\mathbf{X}}$.

*Algorithm 4: The optimal triangulation algorithm, adapted from [12, p. 318].*

### 3.3.4 Transformation to Metric Reconstruction

After the initial triangulation, a *projective reconstruction* will have been obtained. This is a reconstruction that is identical to the original object only up to an arbitrary projective transformation – more precisely, merely intersection and tangency properties of lines and surfaces in 3-dimensional space are invariant. The perceived quality of the reconstruction is thus highly dependent on the viewing position and the projection resulting from it – from a position that is close to the original camera positions, the reprojection error may be very small, whilst for a different viewing position it might be very large. This problem is illustrated in Figure 3.15.

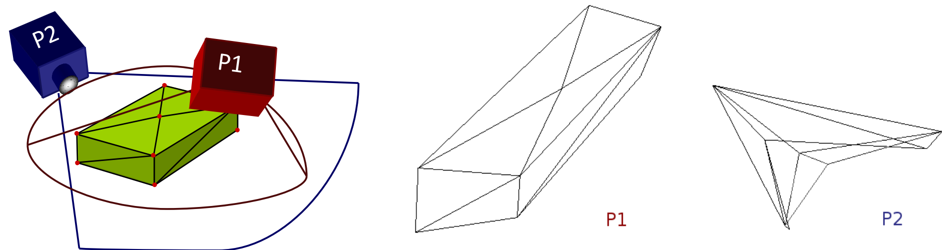



*Figure 3.15: Weakness of the initial projective reconstruction – the same object seen from different perspectives; the approximate camera perspectives are shown on the left. P1 was close to the original camera position, whereas P2 was opposite it.*

In order to obtain a more useful and generic 3D representation, this reconstruction can be "upgraded" to an *affine* or a *metric reconstruction*.

There is a large variety of techniques to implement this step, including consideration of three views using trifocal tensors [12, p. 365–ff.] in order to refine the reconstruction based on pairwise matching, or the computation of a *stratified reconstruction* using an infinite homography $\mathtt{H}_\infty$ by considering the plane at infinity ($\pi_\infty$; the plane in which the vanishing points lie) and subsequent upgrade to a metric reconstruction based on the absolute conic $\omega$.

All of these approaches are very complex both in implementation and computation. For this reason, I chose to implement the *direct method* for upgrading to a metric reconstruction [12, p. 275–ff.]. This approach uses ground truth coordinates specified by the user in order to infer a homography that allows the metric camera matrices $\mathtt{P}_M$ to be found.

In order to be able to use this with the rest of the framework, a facility for the user to input the ground truth coordinates was required. A *Ground Truth Point Editor* dialog was added to the front-end, allowing specification of five (or more) ground truth points in a table (Figure 3.16).

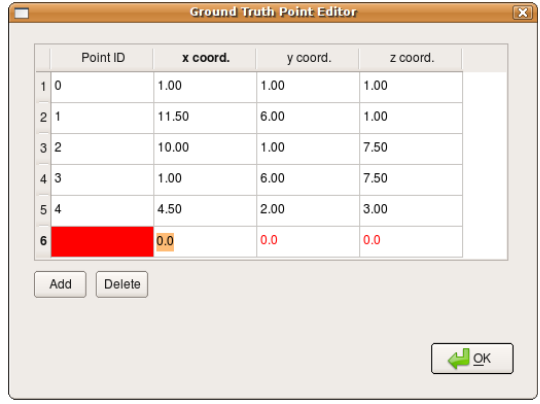



*Figure 3.16: The Ground Truth Point Editor in use.*

Objective: Given a set of projected points $\mathbf{x}_i$ and their initial projective reconstructions $\mathbf{X}_i$ as well as a set of (at least five) ground truth coordinates $\mathbf{X}_{Ei}$, find the metric reconstructions for all $\mathbf{X}_{Ei}$.

Algorithm:

1. **Get metric camera**: Using an adapted version of the DLT algorithm for camera resection [12, p. 178], compute two linearly independent equations per given ground truth point:

$$\begin{bmatrix} \mathbf{0}^\top & -w_i\mathbf{X}_{Ei}{}^\top & y_i\mathbf{X}_{Ei}{}^\top \\ w_i\mathbf{X}_{Ei}{}^\top & \mathbf{0}^\top & -x_i\mathbf{X}_{Ei}{}^\top \end{bmatrix} \begin{pmatrix} \mathbf{P}^1 \\ \mathbf{P}^2 \\ \mathbf{P}^3 \end{pmatrix} = \mathbf{0}$$

   and solve this for $\mathbf{P}$ (12-vector that holds the entries of camera matrix $\mathtt{P}$), giving the metric camera matrix $\mathtt{P}_M = \mathtt{PH}^{-1}$, where $\mathtt{H}$ is a homography relating the projective camera to the metric one.

2. Do the same for subsequent images, obtaining further metric cameras.
3. **Triangulate again**: Using the metric camera matrices and the initial point correspondences as input, run the normal triangulation algorithm to obtain the metric positions of 3D point $\mathbf{X}_i$.

*Algorithm 5: The transformation to metric using ground truth points.*

The suggested method of approaching the direct reconstruction itself infers the homography $\mathtt{H}$ by direct correlation of the ground truth points to the 3D points in the projective reconstruction using the homography, *i.e.* $\mathbf{X}_{Ei} = \mathtt{H}\mathbf{X}_i$, for measured ground truth point $\mathbf{X}_{Ei}$. Consequently, the projection of $\mathbf{X}_i$ is related by $\mathbf{x}_i = \mathtt{PH}^{-1}\mathbf{X}_{Ei}$. However, solving this for $\mathtt{H}$ is non-trivial, since it requires the homography $\mathtt{H}$ to be obtained by decomposition of the matrix product $\mathtt{PH}^{-1}$. For this reason, I devised a new algorithm to derive the metric camera matrices that does so without explicitly solving for $\mathtt{H}$ – the crucial insight being that $\mathtt{H}$ is actually only required to transform the projective camera matrices to their metric counterparts. The algorithm is described in Algorithm 5. Since it only uses input data from a single image, it is inherently generalisable to $n$-views.

Implementing this algorithm was reasonably straightforward – as usual, a MATLAB implementation preceded the C++ implementation in order to ensure the soundness of results.

A discussion of the effects of choosing different possible ground truth points can be found in the Evaluation chapter – see Section 4.3.4.

## 3.4 Texture Extraction

As the final step of the structure-from-motion pipeline, I implemented a facility to extract texture data from the input video and impose it onto the previously generated 3D model.

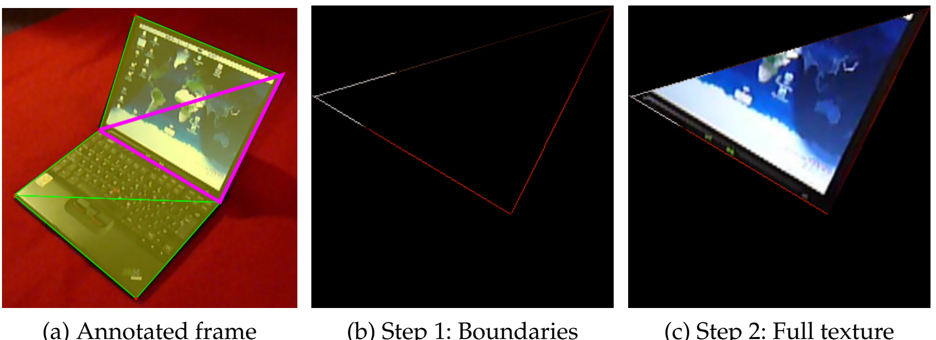
The implementation I chose for extracting textures is a two-step process:

1. **Extraction of pixel data from image**:

   (a) The annotations, and in particular the surfaces, are used to determine the boundaries of the texture to be extracted.

(b) The pixel data between these boundaries is copied to a buffer in memory or written to a file and padded appropriately.

2. **Determination of texture coordinates**: OpenGL requires a set of texture coordinates that identify where in the texture data the polygon vertices should be localised. These coordinates are computed and stored in a data structure.



(a) Annotated frame (b) Step 1: Boundaries (c) Step 2: Full texture

*Figure 3.17: Texture extraction example; the highlight in (a) was added manually for illustration.*

### 3.4.1 Texture Data Extraction

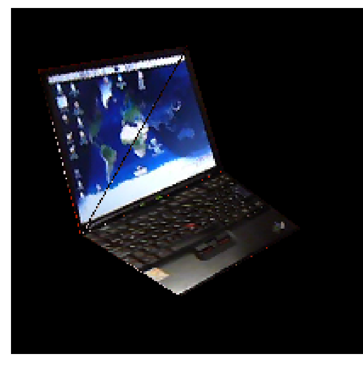


*Figure 3.18: Textured model produced from the example in Figure 3.17.*

In the data extraction step, first the correct size of the texture is calculated by considering the bounding box of the surface. Since OpenGL textures are required to have side lengths that are equal to powers of two, the width and height of the bounding box are rounded up to the next power of two. The buffer is then initialised with all black pixels.

In the next step, Bresenham's line drawing algorithm [6] is used to demarcate the edges of the texture in the buffer, filling a one pixel wide edge with the appropriate colour. An example of the intermediate shape generated by this is shown in Figure 3.17b.

Subsequently, a scanline algorithm (see Algorithm 6) is run on the buffer, filling the parts of the image that are within the texture with the appropriate pixel data. In order to determine whether a boundary crossing is entering or leaving the texture, a crossing count is maintained.

### 3.4.2 Texture Coordinates

In the final part of the texture extraction code, the texture coordinates are calculated. These control the way in which OpenGL will project the texture onto the surface, and hold the position of the vertices relative to the texture rectangle.

Objective: Given an image containing triangle outlines, fill in the texture data within the triangle.

Algorithm:

1. **Initialisation**: Set $y = 0$, $x = 0$.
2. **Scanline Iteration**: Set $crossings = 0$. Iterate over all $x$. For each pixel $(x, y)$,
   - if it is *not* black, it is on a line. Increment *crossings* when the next black pixel is detected.
   - if *crossings* is odd, fill in the pixel (it must be inside the triangle).

   Increment $y$ and repeat.
3. **Termination**: Once the top of the image is reached, terminate.

*Algorithm 6: The texture extraction algorithm (summarised, disregarding special cases).*

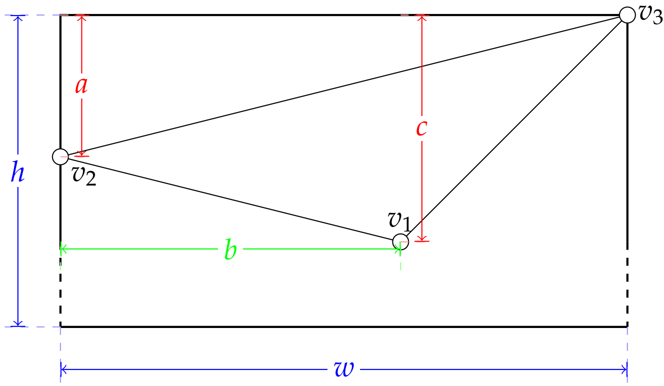



*Figure 3.19: Texture coordinates example.*

An illustrative example of this, which the following explanation will refer to, is given in Figure 3.19. The coordinates of the vertices relative to the bounding box, *i.e.* $v_1 = (b, c)$, $v_2 = (0, a)$, $v_1 = (w, 0)$, are taken and divided by the side lengths of the texture in the respective components. This gives, for each component, a ratio in the range $[0, 1]$ of the vertex position within the texture. More precisely, in the above example, we get

$$T_{v_1} = \left(\frac{b}{w}, \frac{c}{h}\right), \qquad T_{v_2} = \left(0, \frac{a}{h}\right), and \qquad T_{v_3} = (1, 0) .$$

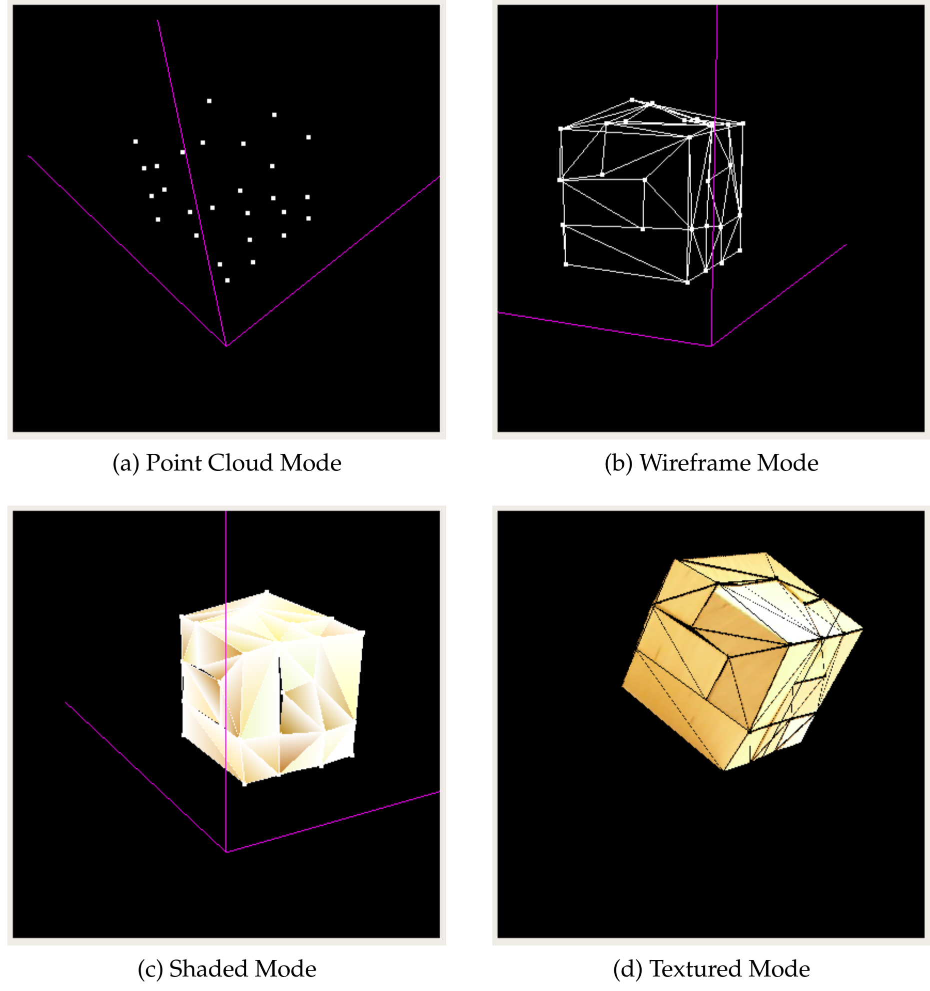
This ratio is then stored to a `TextureCoords` data structure, which is just a $n \times 2$ array (where $n$ is the number of vertices on the polygon, so $n = 3$ for triangles) and this data structure is passed to the rendering code, along with a pointer to the texture data in memory.

## 3.5 Model Viewer

OpenGL provides a cross-language, cross-platform API for 3D graphics that is fundamentally based on a *stateful*, procedural approach to rendering.

The model viewer is an OpenGL-based viewing widget that can display fully textured reconstructed 3D models right away from the output of the structure-from-motion pipeline.

### 3.5.1 Display Modes



(a) Point Cloud Mode

(b) Wireframe Mode

(c) Shaded Mode

(d) Textured Mode

*Figure 3.20: Examples of the different rendering modes in the model viewer.*

The model viewer supports four different display modes, as illustrated in Figure 3.20.

1. **Point cloud mode**: In this mode, merely the vertices of the model are assembled into a drawing list and are then rendered in `GL_POINTS` mode. This produces uniformly coloured 2D points that always face the camera, located at the vertex coordinates.

2. **Wireframe mode**: This mode adds lines connecting the vertices, and displays both interconnecting lines (rendered as a `GL_LINES` list) and vertices. The lines are uniformly coloured.

3. **Shaded mode**: In shaded mode, the colour values at vertices are taken into account and are linearly interpolated across the triangles in the model.

4. **Textured mode**: A list of textured `GL_TRIANGLE` elements is rendered. The textures are extracted using the techniques described in Section 3.4 and a list of pointers to their representations in memory is provided to the model viewer along with a list of texture coordinates. This then passes these pointers and texture coordinates to OpenGL, which draws and interpolates the textures on the model. This mode is the most complex display mode and consequently it takes marginally longer to render the model than in shaded mode.

Note that not every combination of annotations may provide output in every mode – for example, if only points and edges exist in the annotations, then obviously shaded and textured mode will be no different in appearance from wireframe mode.

### 3.5.2 OpenGL with Qt

In order to provide this output, a customly written Qt widget, `GLWidget`, that provides an OpenGL viewport via Qt's OpenGL module is used. Even though the Qt module abstracts away some of the complexity of integrating an accelerated OpenGL viewport into a Qt dialog window, it still requires the programmer to write some fairly basic OpenGL setup and rendering code.

A full description of the OpenGL code written for the implementation of the model viewer is beyond the scope of this dissertation, but a short summary will be given in the following (and some example code for drawing a textured triangle can be seen in Listing 3.1).

```
GLuint list = glGenLists(1);
glNewList(list, GL_COMPILE);

glEnable(GL_TEXTURE_2D);
glShadeModel(GL_FLAT);

glBindTexture(GL_TEXTURE_2D, textures[tex_index]);
glBegin(GL_TRIANGLES);
    for (uint8_t j = 0; j < 3; j++) {
        glTexCoord2f(tc[j][0], tc[j][1]);
        glVertex3f(t.v(j).x(), t.v(j).y(), t.v(j).z());
    }
glEnd();

glEndList();
```

*Listing 3.1: Code used for drawing a textured triangle.*

In order to render a model in projective OpenGL space, it is first necessary to set up a viewing volume and configure various global parameters such as back face culling and shading.

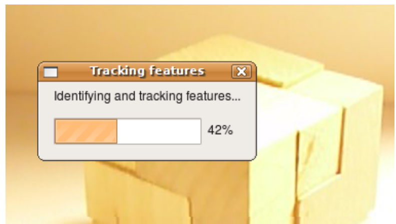
Objects to be drawn are arranged in *drawing lists*, which contain a set of coordinates and interpret them in different ways depending on the type of drawing list. Implementing this was made fairly straightforward by the abundance of OpenGL related documentation, although the official OpenGL programming guide, the "Red Book" [16] proved most useful. A small complication I encountered was the fact that Qt and OpenGL use different byte-orderings for specifying RGB triplets[3], but this was easily overcome by passing the `GL_BGRA` option to the `glTexImage2D()` function, which is responsible for loading textures into OpenGL.

## 3.6 Kanade-Lucas-Tomasi Feature Tracker

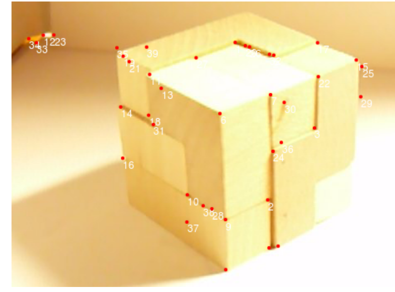
The Kanade-Lucas-Tomasi feature tracker (KLT) is a standard feature tracking algorithm commonly used in computer vision applications. Based on work initially published by Lucas and Kanade in 1981 [15], it was later extended by Shi and Tomasi [22] and has been implemented for many languages and platforms.

It first locates features by examination of the minimum Eigenvalue of a local gradient matrix, computed by convolution with the derivative of a Gaussian for the surroundings of each pixel. It then subsequently uses a Newton-Raphson minimisation of the difference in intensity between two windows in two different images to track the window movement.

I thought that it would be a useful extension to the project to integrate an automated means of feature tracking – this could facilitate very rapid construction of reconstructions by simply connecting up automatically inserted feature points.




(a) Tracking features...



(b) ... and the result

*Figure 3.21: KLT in action.*

The implementation of the KLT algorithm that I used is provided as a C library by Stan Birchfield [4]. It provides a functional interface that, given image data pointers, image dimensions and a desired number of features, will detect features in the first image and then track them across a sequence of images, returning a `KLT_FeatureList` data structure.

My integration of this into *Proteus* amounted to adding the following components:

1. A **tracker thread** in which the feature tracking computation runs – this is so that the GUI does not become unresponsive during feature tracking. It operates in a way that is very similar to the video decoding thread described in Section 3.1.1.

[3] Qt internally uses the endianness of the machine to determine the byte order, which on a little endian x86 machine amounts to BGRA, *i.e.* placing the blue value in the most significant byte of a word. OpenGL by default uses RGBA, and hence expects the red value in the most significant byte.

2. Writing code to convert between the native representation of annotations in *Proteus* and between `KLT_FeatureList` data structures.

Of those, the former turned out to be significantly more challenging than the latter – before features can be tracked for a video frame, it first needs to be decoded. In order to ensure this, a regime of locks using the facilities provided by `QFuture` (an abstract locking implementation provided by Qt on top of `pthreads`, using condvars and mutexes) had to be built, allowing the GUI thread continue running concurrently, but enforcing strictly serial execution of decoding and tracking for any particular single video frame.

The conversion of data structures on the other hand was extremely straightforward to implement: Only the `x` and `y` fields of the `KLT_Feature` struct and a unique identifier for every feature are required in order to be able to populate the list of annotation points used by the front end.

## 3.7 Mesh Export

### 3.7.1 Choice of Model Format

A facility to export 3D models generated in *Proteus* into a well-supported model format was one of the proposed extensions included in the original project proposal.

I chose to use the PLY mesh format [5], also known as the *Stanford Triangle Format*, which is a standard format for portable storage of high-resolution mesh data in scientific applications. The choice fell onto the PLY format for two reasons: it is an open and well-documented format, and encoding into it is straightforward to implement since it comes with an ASCII variant.

The following section will briefly touch on how exporting to PLY files was implemented in this project.

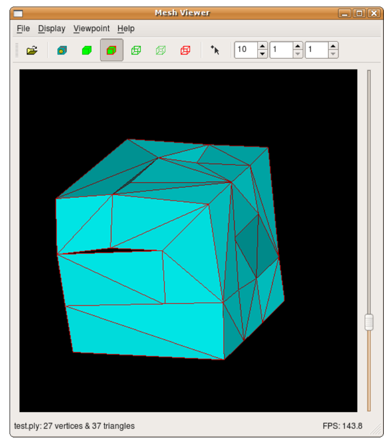



*Figure 3.22: Exported mesh in the* mview *mesh viewer.*

### 3.7.2 PLY Exporter

Due to the simplicity of the PLY file format, the implementation of the PLY export code was fairly straightforward. All that was required was a facility to convert the collections of `ThreeDPoint`, `ThreeDEdge` and `ThreeDTriangle` primitives into their respective representations in the PLY format and write those out to a file, one line per primitive.

The actual implementation uses the C++ standard library file stream class `ofstream` to create a file stream object with an overloaded "«" operator to store strings into the file. Looping over the primitive collections (passed as `std::vector`), strings are generated by simple concatenation and passed into the output stream using the overloaded operator.

## 3.8 Serialization of Annotation Data

During development, it became clear that a way of permanently storing a sequence of annotations and being able to load them again at a later point in time would be advantageous to facilitate debugging and comparative testing.

In order to implement this, a mechanism to serialize the annotation data structure – in fact, the entire list of data structures corresponding to the individual frames of the video – had to be devised.

The difficulty of this task was increased by the fact that by design the annotation data structures rely heavily on pointers, which are normally pointing into memory dynamically allocated at run time. Simply writing out the pointer targets would not suffice, since the memory addresses would be different on the next execution of the program. Hence a *deep copy* mechanism for pointers had to be used.

Investigation showed that the `boost` C++ library includes a serialization framework that is capable of performing deep copies of pointers and, in addition to this, supports ASCII and XML output formats. These are both human-readable and hence ideal for manual inspection of the contents of the data structures as well as modification of details. In the process of serializing the data structure, `boost` resolves pointers and replaces them by numerical references to other parts of the output file.

I implemented the necessary methods to support the `boost` serialization mechanism – an excerpt from an example export is shown in Listing 3.2.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="4">
<fv class_id="0" tracking_level="0" version="0">
   <count>210</count>
   <item_version>0</item_version>
   <item class_id="1" tracking_level="0" version="0">
      <id>0</id>
[...]
      <list class_id="2" tracking_level="0" version="1">
         <ehead class_id="3" tracking_level="1" version="0" object_id="_0">
            <n1 class_id="4" tracking_level="1" version="0" object_id="_1">
               <x>361.5</x>
               <y>393</y>
               <base class_id="-1"></base>
               <next class_id_reference="4" object_id="_2">
                  <x>473</x>
                  <y>302</y>
[...]
```

*Listing 3.2: Sample XML Serialization Export.*

Furthermore, `boost` serialization supports versioning – thus making it possible to load serialized data from previous versions even if the data structure has been changed in the meantime: each object carries a version number with it that can be tested for in the deserialization code.

## 3.9 Summary

This chapter described the implementation work I have done for the *Proteus* project. This encompassed a survey of the interactive annotation frontend, its operation and guidance features as well as the underlying data structures. Then I moved on to describing the structure-from-motion pipeline, which consists of implementations of algorithms for finding the fundamental matrix, finding the camera matrices, triangulation and upgrading the reconstruction to metric. The theory behind these core algorithms was explained and related to the design decisions made when implementing them.

Finally, I outlined the texture extraction process and the implementation of the internal OpenGL model viewer, before moving on to the additional extensions implemented, namely integration of the KLT feature tracker, export to PLY meshes and serialization of annotation data to the XML format. All of these components were implemented successfully, and together they form a complete interactive structure-from-motion framework.

# Chapter 4

# Evaluation

*O Heaven, were man*
*but constant, he were perfect: that one error*
*fills him with faults; makes him run through all sins*
*Inconstancy falls off, ere it begins.*

– Proteus in *"The Two Gentlemen of Verona"* by William Shakespeare.

The objective of this chapter is to present a concise overview of the results of the design and implementation work undertaken for *Proteus*, and to compare the results to the initial goals. It describes the main quantitative results obtained and, using a number of examples, demonstrates how 3D models are successfully reconstructed from video sequences.

Furthermore, the chapter elaborates on the conduct and analysis of a user study that was carried out to evaluate the front-end user interface, looking in particular at its interactive guidance features.

## 4.1 Overall Results

The success criteria of the project as described in the initial proposal (see Appendix A.10) were threefold (summarised for brevity here):

**Criterion 1**: *A video annotation framework that supports [...] "snapping" of feature points [...] has been implemented.*

This goal was tackled at the start of the implementation, and the results are described in Sections 3.1.1, 3.1.2 and 3.2. The result is a fully-functional graphical front-end for annotation of videos, supporting advanced guidance features using edge detection (see Section 3.2.3).

**Criterion 2**: *An interactive structure-from-motion implementation [...] has been implemented. It is shown to be working by applying it to a 360 degree shot of a cube or some other distinctive geometric shape.*

This goal made up the core of the project and, as expected, was the most difficult of the objectives. A structure-from-motion pipeline was built (see Sections 2.1 and 3.3), using various

different algorithms. Its evaluation is undertaken in Section 4.3 and includes the aforementioned "Cube" example.

**Criterion 3**: *Textures for the faces in the extracted 3D model are obtained from the input image data using per-polygon texture extraction. [This] is shown to be working by applying the method to a textured cube.*

The final goal has been described in Section 3.4. Textures can be extracted and applied to the OpenGL model generated by the structure-from-motion pipeline.

Furthermore, a number of optional extensions – both ones that were included with the original project proposal and ones that were developed additionally – have been implemented successfully. Namely, the most important of these are:

- An interactive OpenGL-based viewing widget (Section 3.5).
- Integration of the KLT feature tracker (Section 3.6).
- Export to the PLY mesh format (see Section 3.7.2).
- A facility to save/load sets of annotations as/from an XML file (Section 3.8).

A pleasing overall result is that the minimum quality of input content supported actually turned out to be much lower than originally expected – relatively noisy VGA-resolution M-JPEG videos from a standard digital photo camera were found to be sufficiently good to produce acceptable 3D reconstructions.

## 4.2 Testing

Since this project encompassed several components that had to be developed in a highly linear fashion, testing and verification were crucial to its success. Hence each component was tested both individually and when it was integrated with the rest of the system.

### 4.2.1 MATLAB Unit Tests

Since the MATLAB implementations of some algorithms were used as a basis for the C++ implementations in the actual production code, the verification of their correctness was especially crucial. For this reason, I wrote a modular unit testing framework that verifies all MATLAB components individually using pre-defined synthetic input data with known results that had been verified either manually or using existing toolkits [13; 25; 34].

Each unit test consists of a *test case* with well-defined inputs and expected outputs. It calls the appropriate MATLAB function with the inputs, and then compares the outputs with the expected values. For every test, an individual maximum error threshold is set (between $10^{-3}$ and $10^{-9}$, depending on the task) and the results are checked against this. This approach can be seen as an instance of the methodology of *Black Box Testing*, where knowledge about the internals of an application is ignored and merely the correctness of the output for well-defined input is judged.

```
Terminal
File Edit View Terminal Tabs Help
malte@cassiopeia:~/University/Project/misc/matlab/unittests$ ./run.sh
+--------------------------------------------------------------------------------+
| Proteus Maths Unit Tests                                                       |
+--------------------------------------------------------------------------------+

  Run started

  [  OK  ] : 2D Homography
  [  OK  ] : Residual function
  [  OK  ] : Epsilon Tensor

  [  OK  ] : Fundamental Matrix
  [  OK  ] : Camera Matrices
warning: Inconsistent orientation of point match
  [  OK  ] : Triangulation (linear algorithm)
warning: Inconsistent orientation of point match
  [  OK  ] : Triangulation (optimal algorithm)
  [  OK  ] : Upgrade to Metric

  Done.
  8 passed,  0 failed.

  Run finished

malte@cassiopeia:~/University/Project/misc/matlab/unittests$
```

*Figure 4.1: The output from the unit test framework for the* MATLAB *routines.*

A short shell script was written in order to be able to quickly run all of these tests in sequence. The resulting output from the script is shown in Figure 4.1; the script itself is shown in the appendix in Section A.2.2 and an example test case is shown in Section A.2.1.

### 4.2.2 Verification of the C++ Implementation

After the MATLAB code had been established to work correctly, the process of porting it to C++ was mainly characterised by *stepwise porting* the algorithms, verifying agreement between the two implementations after every major stage. This was done using the same synthetic input data that was already used for the MATLAB tests, so that comparisons were straightforward.

In order for the implementation to pass a test to satisfactory standard, the values returned by the C++ and MATLAB implementations were allowed to differ at most by orders of magnitude that could be attributed to differences in floating point representations (MATLAB uses single-precision, while my C++ code uses double-precision floating point).

## 4.3 Empirical Evaluation

In this section, I will briefly outline how some parts of the project were evaluated in terms of quantitative measures – in particular, the performance of the system and the accuracy of the reconstruction mechanisms.

### 4.3.1 Performance Evaluation

While an analysis of the user interface responsiveness is difficult to conduct empirically, the performance of the reconstruction code can be measured. Table 4.1 shows measurements taking using different reconstruction modes and different numbers of input features.

| | 8 features | 26 features | 100 features |
|---|---|---|---|
| Projective, 2-view, wireframe | 3.1 | 7.2 | 22.1 |
| Projective, 2-view, shaded | 17.3 | 40.8 | – |
| Projective, 2-view, textured | 134.0 | 540.1 | – |
| Projective, $n$-view, wireframe ($n = 10$) | 13.0 | 27.0 | 85.2 |
| Metric, wireframe | 5.0 | 10.7 | 22.9 |
| Metric, shaded | 23.4 | 45.4 | – |
| Metric, textured | 142.0 | 541.2 | – |

*Table 4.1: Time measurements for different reconstruction techniques and input sizes. All values are medians of 50 runs and all times are in milliseconds. "–" denotes no measurement being taken. The 100 feature case was only measured for wireframe mode since no detailed model with surfaces for 100 features was available.*

A number of observations can be made from this data. The reconstructions in general are computed very quickly. The largest single overhead in the computation is added by texture extraction, which results in the reconstruction time going up by about an order of magnitude. This was expected and is explained by the texture extraction algorithm being unoptimised and large textures having to be copied into the OpenGL memory buffers.

There is no significant difference between the projective and the metric reconstruction approaches – the metric reconstruction tends to take marginally longer, which again is not surprising since it encompasses the projective reconstruction. Similarly, the $n$-view projective reconstruction for $n = 10$ takes longer than the simple 2-view one by about an order of magnitude, which corresponds to a linear growth in the number of frames considered.

### 4.3.2 Structure-from-Motion Pipeline

The main component of the project that can be evaluated quantitatively is the structure-from-motion pipeline.

A useful measure for this purpose is the idea of the *reprojection error* that is introduced by a reconstruction [12, pp. 95–98]. In other words, if a 3D point $\mathbf{X}$ is reconstructed from (amongst others) a projection $\mathbf{x}$, then the reprojection error is the mismatch between $\mathbf{x}$ and the point

$\hat{\mathbf{x}} = \mathtt{P}\mathbf{X}$, *i.e.* projecting $\mathbf{X}$ again using the appropriate camera matrix P. This is commonly expressed as

$$d(\mathbf{x}_i, \hat{\mathbf{x}}_i) = \sqrt{((x_i \;/\; w_i) - (\hat{x}_i \;/\; \hat{w}_i))^2 + ((y_i \;/\; w_i) - (\hat{y}_i \;/\; \hat{w}_i))^2},$$

*i.e.* the geometric distance between reprojected point $\hat{\mathbf{x}}_i$ and the original projection $\mathbf{x}_i$.

A useful way of relating the reprojection error to an entire frame with multiple annotation points is to calculate the *mean reprojection error* for the image, which for $n$ annotation points is given by

$$\frac{1}{n} \sum_i d(\mathbf{x}_i, \hat{\mathbf{x}}_i).$$

The reprojection error will be used as the main evaluation metric in the following, and a number of other criteria will also be mentioned as appropriate.

### 4.3.3 Projective Reconstruction

The projective reconstruction code is evaluated using the "Box" example. This is a very simple geometric shape and described with a small set of eight annotations, which is the exact minimum required for being able to find the fundamental matrix (see Section 3.3.1).

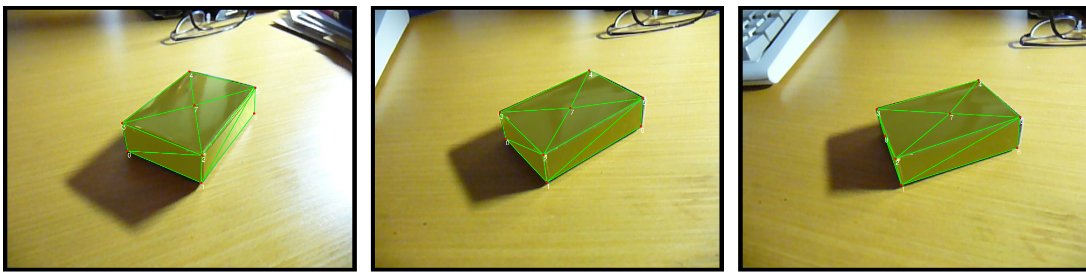
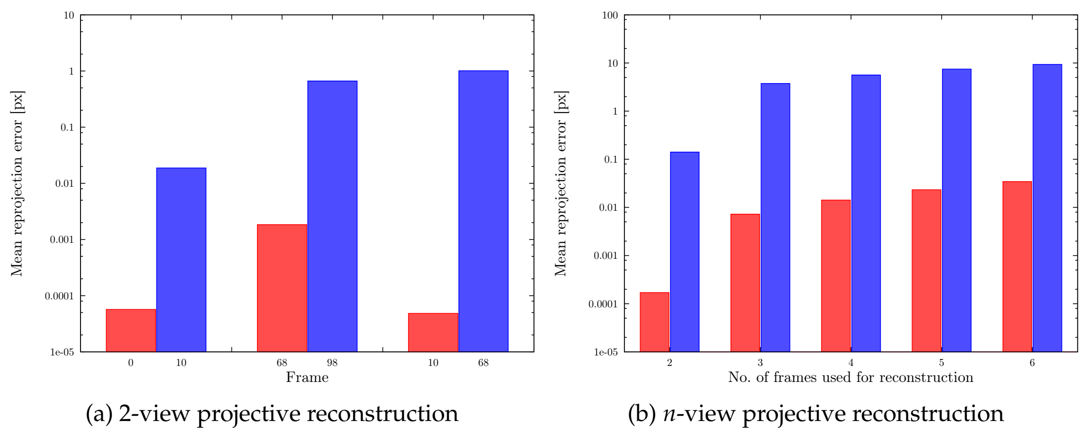


*Figure 4.2: "Box" projective reconstruction example input frames.*

#### 2-view Projective Reconstruction

The reprojection error for the 2-view reconstruction was generally found to be very low. The geometric distances of the reprojected points from the original image points barely ever exceed 2 pixels (3.125 ‰ in $x$ and 4.167 ‰ in $y$-direction) and are normally within the subpixel range.

A wider baseline between the views considered was, as expected, found to yield a greater reprojection error. Figure 4.3a (overleaf) shows the mean reprojection error for 2-view reconstructions based on three annotated frames shown in Figure 4.2, plotted on a $\log_{10}$ scale. It can clearly be seen that the reprojection error increases significantly with the 30 and 58-frame baseline in the second and third image pair.




(a) 2-view projective reconstruction (b) $n$-view projective reconstruction

*Figure 4.3: (a) Mean reprojection error for three pairs of projective views on the "Box" sequence (Frames 0/10, 10/68 and 68/98), (b) Mean reprojection error for pairs of images generated from 2 up to 6 input frames using n-view reconstruction (Frames used: 68–75). Red – first frame of the pair, blue – second frame.*

**N-view Projective Reconstruction**

The algorithms used to compute the $n$-view reconstruction are the same as those used for the 2-view reconstruction, and the computation merely differs in the fact that $(n-1)$ subsequent camera matrices are computed and the triangulation algorithm is supplied with $n$ inputs per point.

The baseline reprojection error is identical to the one introduced by the 2-view reconstruction since the camera matrices are computed iteratively across pairs of views. Hence it should be expected for the reprojection error to increase as it is accumulated over a series of views. The diagram in Figure 4.3b shows the evolution of the reprojection error for reconstructions based on two to six input frames. It is notable that the reprojection error increases significantly between the 2-view and the 3-view reconstruction, but only slowly increases thereafter.

### 4.3.4 Metric Reconstruction

The test content used for the evaluation of the metric reconstruction is the "Cube" example. Two annotated frames from the cube sequence and the resulting reconstruction are shown in Figure 4.5. There are an overall of 26 annotations in this example.

When considering the geometric distance of the reprojected points from the original projections in this example, as plotted in Figure 4.4, an interesting observation is that the disparities in $y$-direction are significantly greater than those in $x$-direction. This can be explained by the fact that the camera movement in the "Cube" example sequence is primarily a translation in $x$-direction, and hence this is reflected more accurately in the camera matrices.

Note also that the first six reprojections are exactly on the original points – this is because

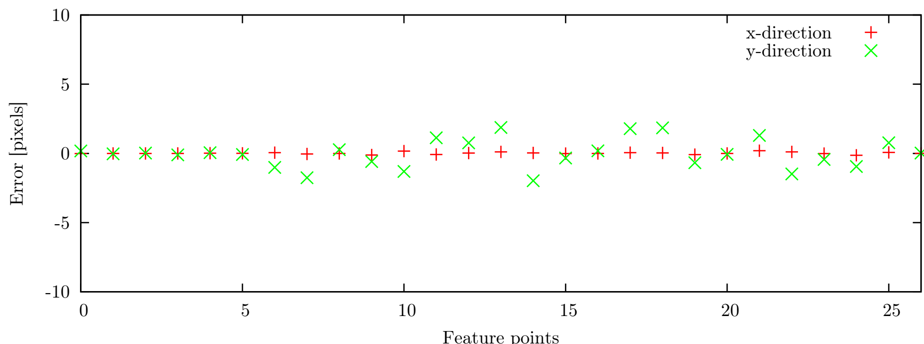
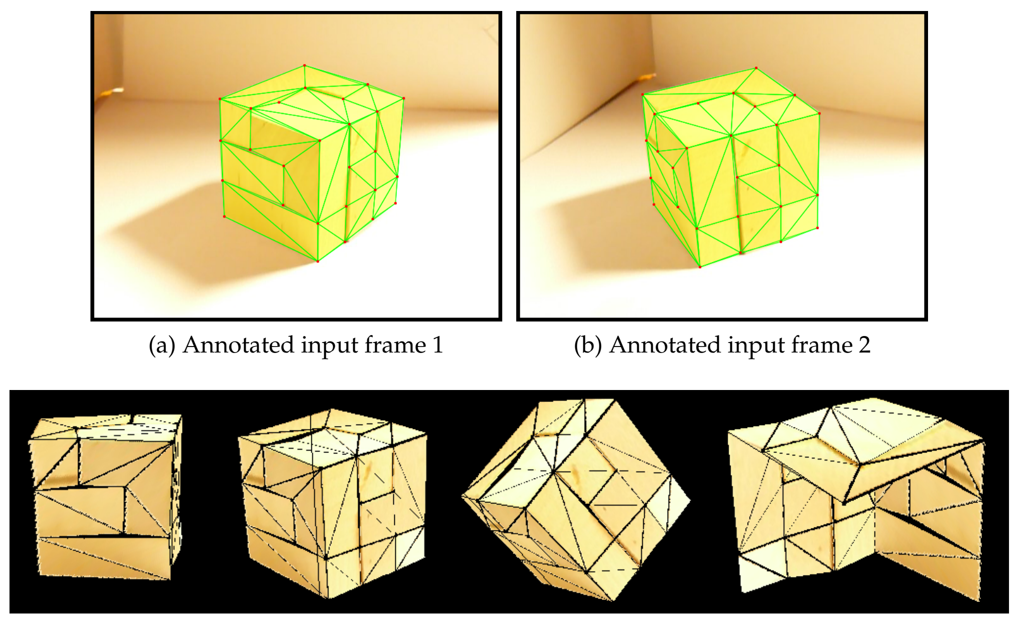



*Figure 4.4: Geometric distance of reprojected points in "Cube" sequence from original image points.* $\mu = (0.0119, -0.0242)$, $\sigma = (0.0748, 1.024)$.

these points were specified as ground truth points and hence are reprojected extremely accurately by the metric camera matrices as these are computed from them.

From the distance data in Figure 4.4, the reprojection error for the metric reconstruction can be calculated. As the fairly low disparities would suggest, this turns out to be rather small for the "Cube" sequence. The mean reprojection error for two pairs of frames, with a baseline of 30 and 40 frames respectively, is found to consistently be close to 1 pixel. Notably, after the upgrade to metric, there seems to be less variation in the reprojection error.

Experiments also showed that sequences with more annotation information generally had a lower and less variable reprojection error after the upgrade to metric than those with only scarce annotation information.



(a) Annotated input frame 1 (b) Annotated input frame 2

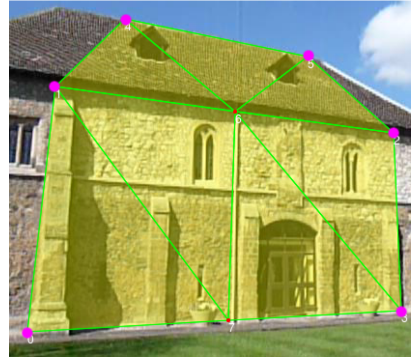
(c) Metric reconstruction of the cube, various views (backside view on the far right)

*Figure 4.5: "Cube" metric reconstruction example.*

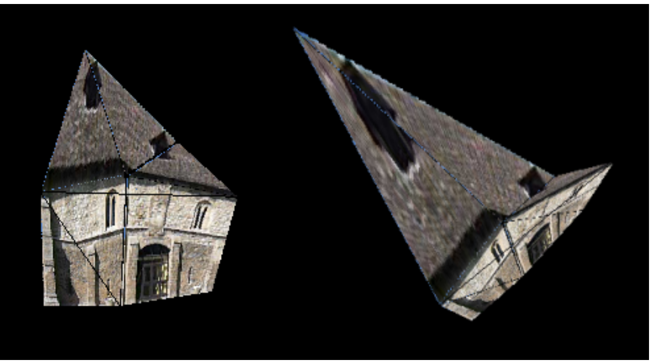
### Choice of Ground Truth Points

When experimenting with different choices of ground truth points for the upgrade from projective to metric reconstruction (as outlined in Section 3.3.4), it quickly became evident that not all possible choices yield reconstructions of the same quality.

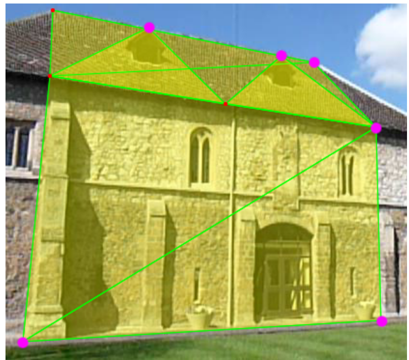
As pointed out by Hartley and Zisserman [12, p. 275–ff.], using more than three co-planar points will yield a poor quality of reconstruction, since these only add information in two dimensions. An example of this can be seen in Figure 4.6 – note how the vertices on the roof are displaced enormously from their real coordinates.



(a) Initial ground truth points



(b) Results: the roof vertices are displaced due to coplanarity of the other vertices



(c) Amended ground truth points

(d) Results: the reconstruction is reasonably accurate

*Figure 4.6: (a), (b): Attempt of a reconstruction of the facade of the School of Pythagoras in St John's College – the upgrade to metric fails due to coplanarity of more than three ground truth points; (c), (d): Using a different set of ground truth points, the reconstruction now succeeds. Ground truth points highlighted in pink for illustration.*

This indicates that not all possible ground truth points (and combinations thereof) are equally well suited to give rise to a metric reconstruction. Furthermore, I found that using points with a wide baseline and close to the boundaries of the bounding box of the object yielded significantly better results than if a set of points inside the object was used. The reason for this is likely to be that the implicit homography created in the step of upgrading to the metric reconstruction corresponds better to the overall model if points on the outside are chosen.

## 4.4 User Study

Originally, I did not intend this project to include a user study. However, it became clear that some components could only be evaluated properly through subjective judgement. In particular, the front-end and guidance techniques implemented do not lend themselves towards an objective evaluation and are judged by users according to highly subjective criteria.

### 4.4.1 Conduct

The study was conducted over a range of ten days in April 2009 and included six participants, all of which were current or former students of the University of Cambridge. All subjects agreed to participate in the study and for their data to be used anonymously for the purpose of this dissertation by signing an informed consent form (see Appendix A.9.2). The same computer was used in all runs of the study in order to avoid the results being skewed by environmental factors. Furthermore, all participants were given the same information about the project before they participated (see script in Appendix A.9.3).

### 4.4.2 Identification of Variables and Choice of Tasks

In order to decide which tasks to include with the user study, it was crucial to first establish the variables that were to be tested for. The exact details of this process are provided in Appendix A.9.1.

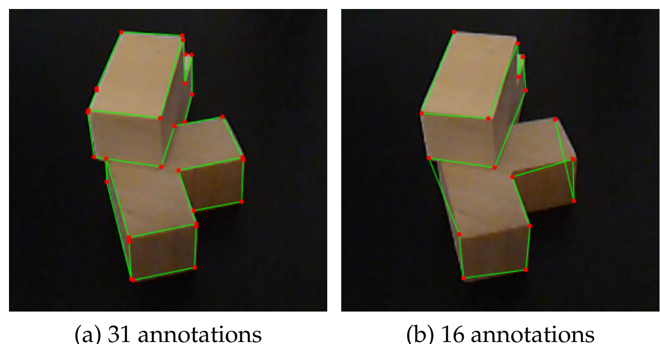
As the main aim of this evaluation, I chose to analyse the perception of the interactive guidance through "edge snapping" in terms of usefulness and accuracy. In order to facilitate this, the subjects were asked to annotate a frame of a given video using the front-end, successively using three different variants of the program:

1. With edge and point snapping, controllable by the user (*i.e.* the CTRL key can be used to disable snapping temporarily).
2. With edge and point snapping, but not controllable.
3. Without any snapping or guidance at all.

The time taken to complete the tasks was measured, and participants were asked to complete a short questionnaire about the subjective perception of the annotation process after completing the tasks. The questionnaire is included in Appendix A.9.4.

### 4.4.3 Annotation Performance

The quality of the annotations made varied greatly between different participants – two examples are shown in Figure 4.7. Some participants also included "imaginary" lines and features in their annotation sequences (*i.e.* annotations for features that are obscured in the image, but which they knew were located at a certain position). This did, however, not happen frequently enough to allow for a proper analysis of the effects on the reconstruction.



(a) 31 annotations (b) 16 annotations

*Figure 4.7: Sets of annotations of the "Blocks" sequence from different participants of the user study.*

Since the participants were not constrained in the number of annotation points they could place, the absolute time they took varied considerably (between about 36 seconds and 3 minutes). In order to obtain a useful quantitative metric from this, the total time was normalised by the number of annotation points placed to give the *per-point annotation time*. The times are summarised in Figure 4.8; the numbers above the bars indicate which order the participant was given the techniques in.

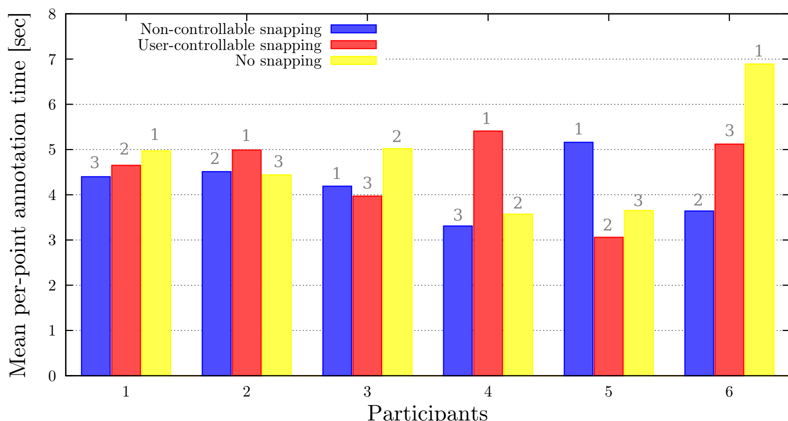



*Figure 4.8: Normalised per-annotation time taken to annotate the video frame by each participant. Numbers indicate the order in which the test runs were performed.*

The dominant trend in this data is quickly found in the fact that there seems to be a significant learning effect between the first and subsequent runs, despite the users being given a chance to familiarise themselves with the interface before doing the actual tasks. With one exception, all participants took longest on the first run. I believe that this must partly result from a continued learning process and partly from familiarisation with the video.

Furthermore, and this was an unexpected result, the method with controllable snapping in many cases yields a longer per-point annotation time than non-controllable snapping. In retrospect, this is not all that surprising: the participants usually made an effort to create a more exact and accurate annotation when they were using this technique. Additionally, the use of both keyboard and mouse in the process might have had a slowing effect.

### 4.4.4 Preference for Edge Snapping Algorithms

The central question that the evaluation study sought to answer was whether there is a preference for interactive guidance through edge snapping. The data gathered from the post-study questionnaire would suggest that this is the case: most participants found the guidance useful and accurate.

#### Snapping – Usefulness

In the first question, participants were asked to rate the perceived usefulness of the interactive guidance provided. The results are shown in Figure 4.9 – the most positive feedback was given on the technique that supported user-controllable edge snapping.

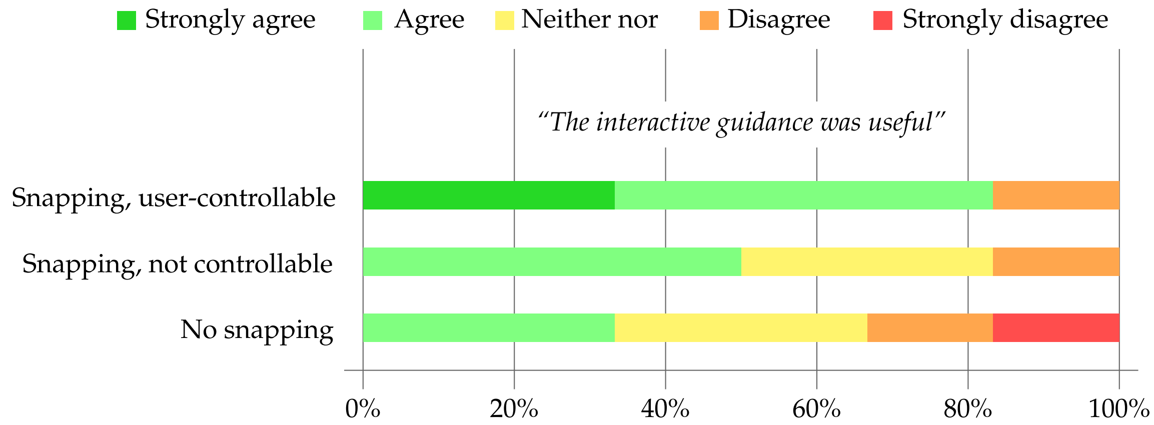



*Figure 4.9: Results: Section 1, Question 1 – Usefulness of interactive guidance.*

In order to verify the statistical significance of these results, the *Kruskal-Wallis test* was used. This test aims to confirm or reject the null hypothesis that there is *no statistically significant difference* between a set of groups, *i.e.* that the differences observed are purely by chance. It does so by ranking results and relating a test statistic to a Chi-squared distribution. The results of applying this test are shown in Table 4.2, with statistically significant results highlighted.

| | *p-value* |
|---|---|
| *Question 1 – Usefulness of snapping* | |
| All methods | 0.152 |
| Pairwise: no snapping and controllable snapping | **0.081** |
| Pairwise: no snapping and non-controllable snapping | 0.440 |
| *Question 2 – Edge detection accuracy* | |
| All methods | **0.013** |
| Pairwise: no snapping and controllable snapping | **0.011** |
| Pairwise: no snapping and non-controllable snapping | **0.017** |
| *Question 3 – Tediousness of annotation* | |
| All methods | 0.397 |

*Table 4.2: Results of the Kruskal-Wallis test. Statistically significant results are highlighted in bold.*

The Kruskal-Wallis test for the first question indicates that there is likely to be a statistically significant difference between the controllable method and the one without snapping, but not with the other pair.

### Snapping – Accuracy

The second question specifically aimed to capture the perceived accuracy of the edge detection in the different methods – including a *placebo-control* in the case of the third method, which did not have any edge detection. The results in Figure 4.10 show that the snapping was judged to be mostly accurate, with only minor differences between the two snapping-based methods.

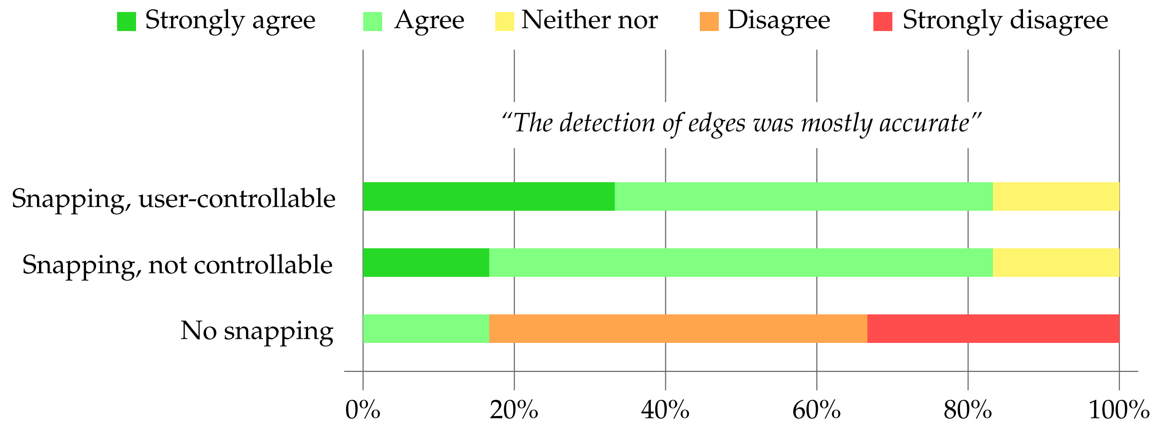



*Figure 4.10: Results: Section 1, Question 2 – Accuracy of edge snapping.*

In the case of the control, only one in six participants was under the impression that there was accurate edge detection, all others judged it to be inaccurate, hence confirming the judgement to be largely objective.

There is no statistically significant difference between the two snapping-based methods according to the Kruskal-Wallis test, but there is between each of those and the version without snapping.

### Snapping – Tediousness

In the third question, participants were asked to judge how tedious they found the annotation process. I included this question in order to evaluate whether there was a correlation between the perception of tediousness and the provision of interactive guidance. The results in Figure 4.11 indicate that there might be a correlation, but that it is not especially strong.

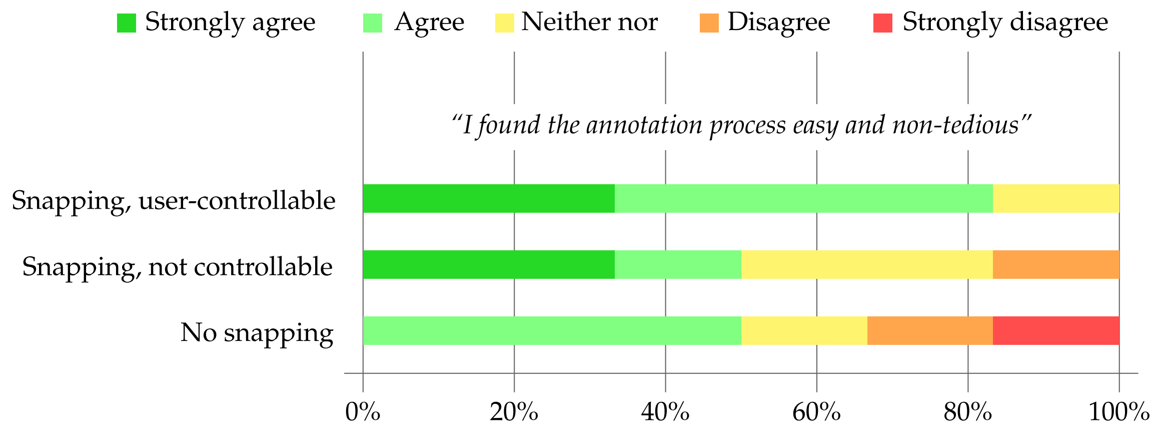



*Figure 4.11: Results: Section 1, Question 3 – Ease/tediousness of the annotation process.*

### 4.4.5 Overall Impression

In the final part of the questionnaire, participants were asked to rate their subjective impression of how easy to use the toolkit they just tested was, plus how it compares to other means of constructing 3D models, and to provide any further comments they might have.

The universal opinion seemed to be that the toolkit was indeed easy to use and that this method of constructing models was significantly more accessible than current professional 3D modelling packages. The results are summarised in Figure 4.12.

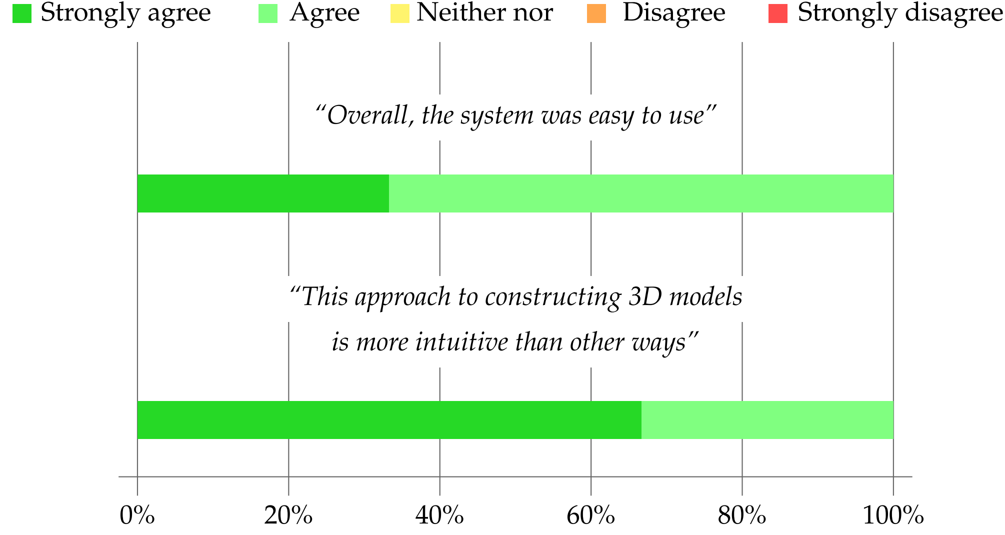



*Figure 4.12: Results: Section 2 – Overall impression. The subjective perception is that the system is easy and intuitive to use.*

Including all of the comments that were made by participants of the evaluation study would exceed the scope of this dissertation, but a brief survey of comments is given below.

*"The snapping was really good [...]. It really is a tedious task and the [...] snapping is definitely helpful at making it less so."*

*"It was good!"*

*"[...] snapping is mostly accurate, [but] is frequently a few pixels off; I suspect the approach without snapping would produce a more accurate 3D model although it would take much longer [...]"*

*"Cool!"*

*"Can we use this on Google StreetView images to build a model of Cambridge?"*

### 4.4.6 Discussion

The analysis of the results of the user study has shown that there is a statistically significant preference for variants of the user interface that utilise edge detection to provide interactive "snapping" when annotating the video. Both the statistical results and subjective impressions conveyed by participants suggest that the variant which allows the user to temporarily turn snapping off was received best.

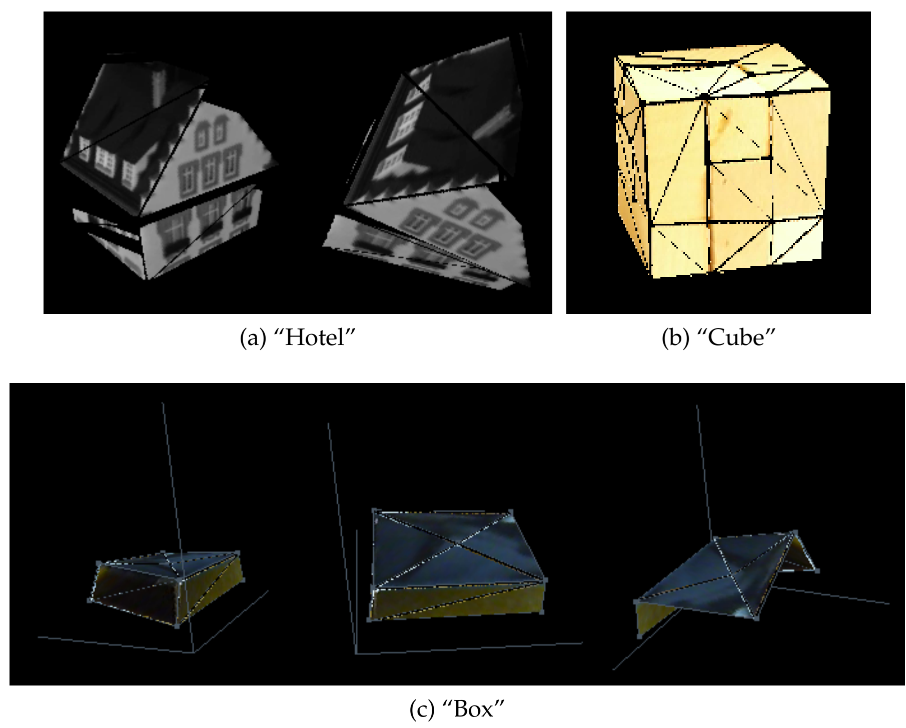
## 4.5 Summary

In this chapter, I have presented an evaluation of the *Proteus* project from a variety of different perspectives. I first compared the results achieved to the original aims of the project – all of which have been fulfilled completely, and were in fact extended by a number of extensions that make the toolkit more useful.

An empirical evaluation of the reconstruction performance and quality was undertaken, and the relative errors were found to be acceptably low for all reconstruction methods implemented. Errors could be reduced further using a range of techniques (see Appendix A.6 for a brief overview), but the level reached is more than satisfactory for the purpose of obtaining basic 3D models.

Finally, I described the user study that was undertaken to evaluate the experimental and subjective aspects of the project. The results show a very positive and promising resonance towards the concept of interactively guided structure-from-motion, which is in line with the overall success of the project.

A number of reconstructions that were produced using the *Proteus* framework are shown in Figures 4.13, and Figure 4.14 shows a practical use case of a model generated using *Proteus* being placed in Google Earth, implementing the example use case mentioned in the introduction chapter.



(a) "Hotel" (b) "Cube"

(c) "Box"

*Figure 4.13: Various reconstructions generated using* Proteus.

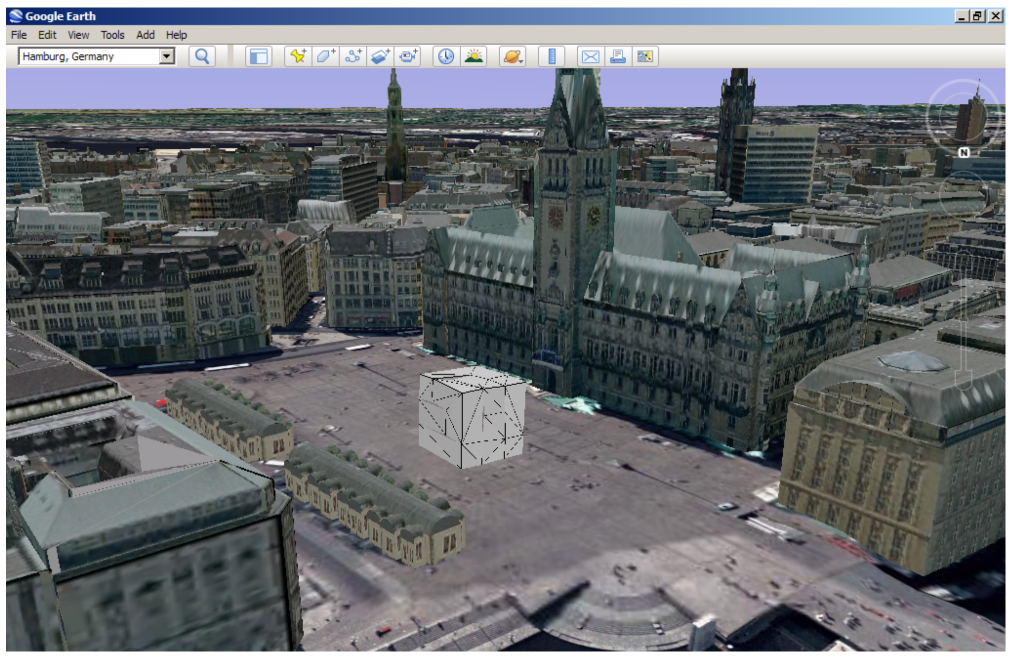



*Figure 4.14: A mysterious cube has appeared on the Rathausmarkt in Hamburg... (model generated using Proteus, converted using MeshLab and placed using Google Sketchup). Google Earth; 3D Buildings © 2008 CyberCity AG, Images © 2009 AeroWest*

# Chapter 5

# Conclusions

This dissertation has described the design, implementation and evaluation of the *Proteus* interactive structure-from-motion framework.

Looking back to the start of the project, I have to admit that I was – to an extent – ignorant towards the enormous complexity of this undertaking. Had I known the complexity and amount of the work I took upon myself, I might have thought twice. However, my ignorance may well have been a blessing – for most parts, I thoroughly enjoyed the journey I embarked on, and whilst working on *Proteus*, I certainly learnt a lot about an exciting field of computer vision that I had previously been unfamiliar with.

## 5.1 Achievements

All initial goals of the project have been achieved, and in fact surpassed – not only was the core of the project implemented successfully, a number of optional extensions, have also been implemented. Some of these were part of the original proposal and some were only conceived during the development phase.

In addition to the end product that exists with the *Proteus* software, I have also acquired a good degree of insight into the concepts and fundamentals of multiple view geometry, structure-from-motion and state-of-the-art algorithms as well as cutting-edge research being done in this area.

I consider the project successful both in the objective sense and in the personal sense of having furthered my knowledge and abilities.

*Proteus*, to my knowledge, breaks new grounds in that it is the first publically available structure-from-motion toolkit that works on input material obtained from consumer-grade digital cameras and also the first one that makes use of interactive annotation as a method of feature identification and matching.

## 5.2 Lessons Learnt

The main lesson that I learnt from this project was that it is very easy to underestimate the complexity of implementing a large-scale application based on research-level techniques of which no reference implementations exist yet. In the core parts of the structure-from-motion components, it often happened that what was a reasonably short description of a method in literature turned out to be a large and complex piece of code to implement. However, a systematic approach imposed control and allowed this to eventually succeed, even leaving time for some more optional extensions.

Since the user study for the evaluation of the front-end was not initially planned to be a part of the project, it had to be designed and conducted within a very short timeframe. This resulted in a tight constraint on the number of subjects and hence a low sample size. If I were to do this project again, I would design the user study much earlier and aim to recruit a larger group of participants in order to improve results.

## 5.3 Future Work

The *Proteus* implementation is complete in the sense that it works and fulfils the success criteria for the project. Nonetheless, there is scope for some further work on it.

My code will be made available on the internet[1], licensed under the GNU General Public License (GPL), for anyone to download and use.

The major improvements that could be made to the project are the implementation of more elaborate algorithms for the structure-from-motion pipeline and improvements to the graphical front end. A brief summary of possible further development is given below.

- **User Interface**: Selectable "tools" for drawing line sequences and surfaces would make the front end more intuitive to use than the current scheme using hotkeys for mode selection.
- **Structure-from-Motion Pipeline**: A full *stratified reconstruction* approach [12, p. 267–ff.] that does not require ground truth information could be implemented. In addition to this, a statistical algorithm for determining the fundamental matrix using maximum-likelihood estimation could be added.
- **Automatic Triangulation**: For example, Delaunay triangulation [8] could be used to automatically triangulate surfaces on an object with highlighted edges.

[1] On the project website, `http://proteus.malteschwarzkopf.de`

# Bibliography


[1] AGARWALA, A. SnakeToonz: A semi-automatic approach to creating cel animation from video. In *NPAR '02: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering* (2002), pp. 139–ff.

[2] AKBARZADEH, A., FRAHM, J.-M., MORDOHAI, P., ENGELS, C., GALLUP, D., MERRELL, P., PHELPS, M., SINHA, S., TALTON, B., WANG, L., YANG, Q., STEWENIUS, H., YANG, R., WELCH, G., TOWLES, H., NISTÃĽR, D., AND POLLEFEYS, M. Towards urban 3D reconstruction from video. In *Proceedings of 3D Data Processing, Visualization and Transmission* (2006), pp. 1–8.

[3] BEARDSLEY, P. A., TORR, P. H. S., AND ZISSERMAN, A. P. 3D model aquisition from extended image sequence. Technical Report 2089/96, University of Oxford, 1996.

[4] BIRCHFIELD, S. KLT: An Implementation of the Kanade-Lucas-Tomasi Feature Tracker. `http://www.ces.clemson.edu/~stb/klt/`. Last accessed April 28, 2009.

[5] BOURKE, P. PLY - Polygon File Format. `http://local.wasp.uwa.edu.au/~pbourke/dataformats/ply/`. Last accessed May 8, 2009.

[6] BRESENHAM, J. E. Algorithm for computer control of a digital plotter. *IBM Systems Journal 4*, 1 (January 1965), 25–30.

[7] CORNELIS, K., POLLEFEYS, M., VERGAUWEN, M., AND GOOL, L. V. Augmented Reality Using Uncalibrated Video Sequences. *Lecture Notes in Computer Science 2018* (2001), 144–160.

[8] DELAUNAY, B. Sur la sphère vide. *Otdelenie Matematicheskikh i Estestvennykh Nauk* (1934), 793–800. Izvestia Akademii Nauk SSSR.

[9] DRUMMOND, T. TooN – Tom's object-oriented numerics library. `http://mi.eng.cam.ac.uk/~twd20/TooNhtml/index.html`. Last accessed Apri 28, 2009.

[10] HADAMARD, J. Sur les problèmes aux dérivées partielles et leur signification physique. *Princeton University Bulletin* (1902), 49–52.

[11] HÄMING, K., AND PETERS, G. An Object Acquisition Library for Uncalibrated Cameras in C++. In *AIP Conference Proceedings of the Fifth International Workshop on Information Optics (WIO 2006)* (October 2006), G. Cristóbal, B. Javidi, and S. Vallmitjana, Eds., vol. 860, pp. 520–526.

[12] HARTLEY, R. I., AND ZISSERMAN, A. *Multiple View Geometry in Computer Vision*, second ed. Cambridge University Press, ISBN: 0521540518, 2004.

[13] KOVESI, P. MATLAB and Octave Functions for Computer Vision and Image Processing. `http://www.csse.uwa.edu.au/~pk/Research/MatlabFns/`. Last accessed May 8, 2009.

[14] LORENSEN, W., AND CLINE, H. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (1987), pp. 163–169.

[15] LUCAS, B., AND KANADE, T. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence* (1981), pp. 674–679.

[16] NEIDER, J., DAVIS, T., AND WOO, M. OpenGL Programming Guide: The Red Book. *USA, Silicon Graphics* (1994).

[17] POLLEFEYS, M. *Self-calibration and metric 3D reconstruction from uncalibrated image sequences*. PhD thesis, K.U.Leuven, 1999.

[18] POLLEFEYS, M. Obtaining 3D Models with a Hand-Held Camera/3D Modeling from Images. presented at Siggraph 2002/2001/2000, 3DIM 2001/2003, ECCV 2000, 2000.

[19] POLLEFEYS, M., KOCH, R., VERGAUWEN, M., AND GOOL, L. V. Hand-held acquisition of 3D models with a video camera. In *Proceedings of the 1999 International Workshop on 3-D Digital Imaging and Modeling (3DIM)* (1999), IEEE Computer Society Press, pp. 14–23.

[20] POLLEFEYS, M., VAN GOOL, L., VERGAUWEN, M., CORNELIS, K., VERBIEST, F., AND TOPS, J. Image-based 3D acquisition of archaeological heritage and applications, 2001.

[21] RILEY, K., HOBSON, M., AND BENCE, S. *Mathematical methods for physics and engineering*, third ed. Cambridge University Press, 2006, ch. 8.18.3, p. 301.

[22] SHI, J., AND TOMASI, C. Good Features to Track. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (1994), pp. 593–600.

[23] SOBEL, I., AND FELDMAN, G. A $3 \times 3$ Isotropic Gradient Operator for Image Processing. presented at a talk at the Stanford Artificial Project in 1968.

[24] SOMMERVILLE, I. *Software engineering*, 8th ed. Addison-Wesley, 2006.

[25] TORR, P. H. S. Structure and Motion Toolkit in Matlab. `http://cms.brookes.ac.uk/staff/PhilipTorr/Code/code.htm`. Last accessed May 8, 2009.

[26] TORR, P. H. S., AND MURRAY, D. W. The Development and Comparison of Robust Methods for Estimating the Fundamental Matrix. *International Journal of Computer Vision 24*, 3 (1997), 271–300.

[27] TORR, P. H. S., AND ZISSERMAN, A. Performance Characterizaton of Fundamental Matrix Estimation Under Image Degradation. *Machine Vision and Applications 9* (1997), 321–333.

[28] TROLLTECH/NOKIA. Qt Designer – Interactive GUI design tool for Qt4. `http://doc.trolltech.com/4.5/designer-manual.html`. Last accessed April 27, 2009.

[29] TROLLTECH/NOKIA. Qt Eclipse Integration for C++. `http://www.qtsoftware.com/developer/eclipse-integration`. Last accessed April 27, 2009.

[30] TROLLTECH/NOKIA. Qt Reference Documentation. `http://doc.trolltech.com/4.4/`. Last accessed April 27, 2009.

[31] VAN DEN HENGEL, A., DICK, A., THORMÄHLEN, T., WARD, B., AND TORR, P. H. S. VideoTrace: rapid interactive scene modelling from video. *ACM Transactions on Graphics 26*, 3 (2007), 86.

[32] VERGAUWEN, M., POLLEFEYS, M., MOREAS, R., XU, F., VISENTIN, G., VAN GOOL, L., AND VAN BRUSSEL, H. Calibration, Terrain Reconstruction and Path Planning for a Planetary Exploration System. In *Proceedings of the International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS)* (2001).

[33] VERGAUWEN, M., POLLEFEYS, M., AND VAN GOOL, L. Calibration and 3D measurement from Martian terrain images. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)* (2001), vol. 2.

[34] ZISSERMAN, A., CAPEL, D., FITZGIBBON, A., KOVESI, P., WERNER, T., AND WEXLER, Y. MATLAB Functions for Multiple View Geometry. `http://www.robots.ox.ac.uk/~vgg/hzbook/code/`. Last accessed May 8, 2009.

# Appendix

## A.1 Detailed List of Requirements

| *System Feature* | | *Priority* | *Difficulty* | *Risk* | ✓ / × |
|---|---|---|---|---|---|
| **Graphical User Interface Front End** | | | | | |
| 1.1 | Basic design and skeleton code | High | Low | Low | ✓ |
| 1.2 | Multithreaded video decoding using *libavcodec* | High | Medium | Low | ✓ |
| 1.3 | Navigation within a video | High | Medium | Low | ✓ |
| 1.4 | Annotation facilities for nodes, edges and triangles | High | Medium | Low | ✓ |
| 1.5 | Internal OpenGL-based model viewer | High | Medium | Medium | ✓ |
| 1.6 | Ground-truth point editor | Medium | Low | Low | ✓ |
| **Data Structures** | | | | | |
| 2.1 | Design and implement annotation data structure | High | Medium | Medium | ✓ |
| 2.2 | Data structures for 3D primitives | High | Medium | Medium | ✓ |
| **Interactive Guidance** | | | | | |
| 3.1 | Implement edge detection using the Laplacian operator | High | Medium | Low | ✓ |
| 3.2 | Implement edge detection using the Sobel operators | High | Medium | Low | ✓ |
| 3.3 | Integrate edge detection metrics with the GUI | High | Medium | Low | ✓ |
| 3.4 | Snapping to existing annotation points | High | Medium | Low | ✓ |
| 3.5 | Interactive guidance for triangle insertion | Medium | Medium | Medium | ✓ |
| 3.6 | Automatic triangulation of annotation sets | Low | High | Medium | × |
| **Structure-from-Motion** | | | | | |
| 4.1 | Implement linear algebra & epipolar geometry helper functions | High | Medium | Low | ✓ |
| 4.2 | Fundamental matrix computation using normalised 8-point algorithm | High | High | High | ✓ |
| 4.3 | Camera matrix computation | High | Medium | Medium | ✓ |
| 4.4 | Triangulation using homogeneous DLT algorithm | High | Medium | High | ✓ |
| 4.5 | Triangulation using optimal algorithm | Medium | High | Medium | ✓ |
| 4.6 | Direct upgrade to metric reconstruction using ground truth points | High | High | High | ✓ |
| **Texture Extraction** | | | | | |

*Continued on Next Page…*

Table A.1 – Continued

| | | | | | |
|---|---|---|---|---|---|
| 5.1 | Extract texture data to image files | High | Medium | Medium | ✓ |
| 5.2 | Compute texture coordinates & pass them to OpenGL | High | Medium | Medium | ✓ |
| **Export / Permanent Storage** | | | | | |
| 6.1 | Support exporting models into PLY mesh format | Medium | Medium | Low | ✓ |
| 6.2 | Support saving/loading of annotation data (serialization) | Medium | High | Medium | ✓ |
| **Automated Feature Tracking** | | | | | |
| 7.1 | Implement multithreaded decoding/tracking across a frame sequence | Medium | High | Medium | ✓ |
| 7.2 | Conversion of KLT data structures to internal ones | Medium | Medium | Low | ✓ |
| **Testing and Evaluation** | | | | | |
| 8.1 | MATLAB unit testing framework | High | Medium | Low | ✓ |
| 8.2 | Facility for calculating reprojection error | High | Medium | Low | ✓ |

*Table A.1: Detailed breakdown of requirements identified in the* Proteus *project.*

## A.2 Unit Test Code

### A.2.1 Example Test Case – `test_triang_optimal.m`

The test case shown in Listing A.1 performs a test run of the *optimal triangulation algorithm* (see Section 3.3.3) in MATLAB . The input test data is set up by the call to `test_setup`, and the results in `Xc` are compared to the known-good result in `X`. If the two agree within an error margin of $10^{-6}$, then the test succeeds.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Unit test for optimal triangulation algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function r = test_triang_optimal()

test_setup

[F,dummy,dummy] = fundmatrix(x1,x2);

P1 = eye(3,4);
P2 = P_from_F(F);

ps = {P1,P2};

is = [ 640 640; 480 480; ];

% triangulate using optimal algorithm (minimise geometric distance)
[xh1,xh2] = opttri(F,p1(:,1),p1(:,2));

ut = [hnormalise(xh1) hnormalise(xh2)];

u = [ ut(1,:); ut(2,:) ];

X = X_from_xP_lin(u,ps,is);

Xc = [ 0.7308134
      -0.6825388
      -0.0029042
       0.0066419 ];

% work out the boolean test result - is our result within 10^-6 of the expected
% answer?
r = !(isequal(round2(X,0.000001),round2(Xc,0.000001)));

return
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% main body

if (BATCHMODE == 1)
 r = test_triang_optimal();
 exit(r);
end
```

*Listing A.1: Test case code for testing the optimal triangulation algorithm.*

### A.2.2 Batch Testing Script – `run.sh`

In order to be able to quickly and conveniently run all unit tests and ascertain whether a change introduced any errors in a directly or indirectly dependent algorithm, I wrote a small shell script that runs all unit tests and outputs the results (see Figure 4.1 in Section 4.2.1 for an example of what the output might look like). The script is shown below in Listing A.2.

```
#!/bin/bash

SUCC=0
FAIL=0

log_success() {
   echo -e "  [ \e[1;32m OK \e[m ] : $@"
   SUCC=$(($SUCC + 1))
}

log_fail() {
   echo -e "  [ \e[1;31m FAILED \e[m ] : $@"
   FAIL=$(($FAIL + 1))
}

run_test() {
   if [ $3 -eq 1 ]; then
      octave -q --eval 'BATCHMODE = 1;' $1 --persist
   else
      octave -q --eval 'BATCHMODE = 1;' $1 --persist > /dev/null
   fi

   R=$?

   if [ $R -eq 0 ]; then
       log_success $2;
   else
       log_fail $2;
   fi
}


##########################################
# main body
##########################################

if [[ -nz $1 && $1 = '-v' ]]; then
   VERBOSE=1;
else
   VERBOSE=0;
fi

echo "+----------------------------------------------------------------------+"
echo "| Proteus Maths Unit Tests                                             |"
echo "+----------------------------------------------------------------------+"
echo
echo "  Run started at `date +%k:%M:%S` on `date +%a\,\ %e\ %b\ %Y`"
echo
run_test "test_homography2d.m" "2D Homography" $VERBOSE
run_test "test_resid.m" "Residual function" $VERBOSE
run_test "test_epstensor.m" "Epsilon Tensor" $VERBOSE
echo
run_test "test_fundmatrix.m" "Fundamental Matrix" $VERBOSE
run_test "test_cameras.m" "Camera Matrices" $VERBOSE
run_test "test_triang_linear.m" "Triangulation (linear algorithm)" $VERBOSE
run_test "test_triang_optimal.m" "Triangulation (optimal algorithm)" $VERBOSE
run_test "test_upgrade.m" "Upgrade to Metric" $VERBOSE
echo
```

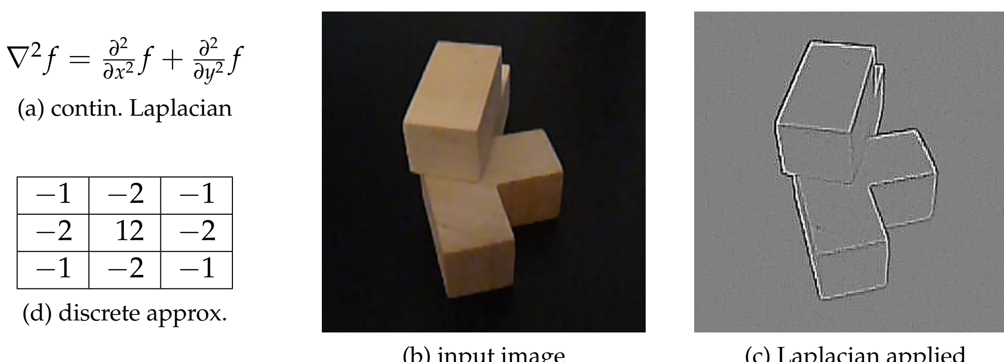
```
echo "  Done."
echo -e "  \e[1;32m$SUCC\e[m passed, \e[31m $FAIL\e[m failed."
echo
echo "  Run finished at `date +%k:%M:%S` on `date +%a\,\ %e\ %b\ %Y`"
echo
```

*Listing A.2: Shell script to run all unit tests in a batch and output the results.*

## A.3 Laplacian Operator for Edge Detection

The Laplacian operator $\nabla^2$ for continuous functions in Cartesian coordinates is defined as the sum of the second partial derivatives in $x$ and $y$. One of several possible discrete approximations to the Laplacian operator is given by the isotropic discrete convolution kernel shown in Figure A.1d.

The result of convolution with this kernel can be seen in Figure A.1c. The edges are localised at zero-crossings of the 2D function – this can clearly be seen in the adjacent bright and dark lines.



$$\nabla^2 f = \frac{\partial^2}{\partial x^2} f + \frac{\partial^2}{\partial y^2} f$$

(a) contin. Laplacian

| | | |
|---|---|---|
| −1 | −2 | −1 |
| −2 | 12 | −2 |
| −1 | −2 | −1 |

(d) discrete approx.

(b) input image

(c) Laplacian applied

*Figure A.1: Application of the Laplacian edge detection operator.*

The results can then be used to generate the gradient field – however, the gradient value of zero at the edges needs to be converted into a large value in order to be able to use the line integral method described before. In order to be able to do this, the gradient values are clamped to the interval $\{-128, ..., 127\}$ and the absolute value is stored in the gradient field.

## A.4 Derivation of the Discrete Line Integral

In Section 3.2.3, an approximation to the discrete line integral is given by

$$I = \sum_{i=0}^{n} h(\mathbf{r}(x_i, y_i)).$$

This can be derived from the continuous line integral using a *Riemann sum* as shown in the following.

Consider the continuous line integral along the projected line:

$$\int_a^b f(\mathbf{r}(t))\, |\mathbf{r}'(t)|\, dt$$

for some scalar field $f : U \subseteq \mathbb{R}^n \to \mathbb{R}$ and $\mathbf{r} : [a, b] \to L$ being a parametrisation of line $L$ such that $\mathbf{r}(a)$ and $\mathbf{r}(b)$ give the endpoints of L.

In order to derive a discrete equivalent of this, a *Riemann sum* can be used. Consider a sum of the form

$$I = \lim_{\Delta t \to 0} \sum_{i=1}^{n} f(\mathbf{r}(t_i))\Delta s_i$$

where $\Delta s_i = |\mathbf{r}(t_i + \Delta t) - r(t_i)| = |\mathbf{r}'(t_i)|\Delta t$ is the distance between subsequent points on the line.
Since in the given case the line is defined by a *discrete* set of points at fixed distance (the pixels), the distance term $\Delta s_i$ simply becomes 1, and the parameter $t$ is replaced by a pair of the form $(x, y)$. The limit then becomes redundant, since the $\Delta t$ term just collapses to the distance between two pixels on a line. By substituting $h$ for $f$, the approximation shown above can be found.

## A.5 Derivation of the Fundamental Matrix

In order to find the fundamental matrix, sufficient data to find the homography $\mathtt{H}_{\boldsymbol{\pi}}$ is required. If this is given, then the above equation can be used to compute $\mathtt{F}$: it can be expanded for $\mathbf{x} = (x, y, 1)^\mathsf{T}$ and $\mathbf{x}' = (x', y', 1)^\mathsf{T}$, giving

$$x'xf_{11} + x'yf_{12} + x'f_{13} + y'xf_{21} + y'yf_{22} + y'f_{23} + xf_{31} + yf_{32} + f_{33} = 0$$

where $f_{ij}$ are the (unknown) entries of the fundamental matrix.
Letting $\mathbf{f}$ denote the 9-vector corresponding to the entries of $\mathtt{F}$ in row-major order, a set of linear equations can be obtained for $n$ point matches (each match corresponds to a row):

$$\mathtt{A}\mathbf{f} = \begin{bmatrix} x'_1x_1 & x'_1y_1 & x'_1 & y'_1x_1 & y'_1y_1 & y'_1 & x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x'_nx_n & x'_ny_n & x'_n & y'_nx_n & y'_ny_n & y'_n & x_n & y_n & n \end{bmatrix} \mathbf{f} = \mathbf{0}$$

Since $\mathtt{F}$ has seven degrees of freedom, at least seven point correspondences are required in order to be able to solve for the fundamental matrix.

## A.6 Techniques to Optimise Reconstruction Quality

The initial solution obtained from the structure-from-motion algorithms is likely to only be a very rough estimation of the actual scene geometry: It is only a *projective reconstruction*, and input data ambiguities will be reflected in the data.

In the *Proteus* implementation, I have used the method of *direct reconstruction using ground truth data* due to its relative simplicity and universal applicability. However, there are various other ways of improving on the initial reconstruction, a brief survey of which is given in the following.

1. **Camera calibration**: Attempt to extract the intrinsic camera parameters such as focal length and aperture from the given point correspondences; the space of solutions and approaches to this task is quite large, but a common approach seems to be to translate the camera parameters to constraints on an absolute conic, $\boldsymbol{\omega}$, solve for this and use the result as a calibration pattern.
   This approach is generally used as part of a *stratified reconstruction* approach.
2. **Trifocal tensors**: This extends the pairwise matching used in 2-view geometry by relating triples of views in addition to pairs, and uses this to reduce the ambiguity in the projection by further restricting the possible positions of the reconstructed 3D point through the constraints imposed by the third view.
3. **Dense depth representation**: In addition to the strong feature correspondences used for the initial reconstruction, less pronounced correlations between other image points can be used to extract further surface detail for a higher quality reconstruction.
4. **Bundle Adjustment** of point correspondences: For a reconstruction that has been obtained from a sequence of images, a final improvement can be made by performing a global minimisation of the reprojection error, aiming to find the camera matrices $\hat{\mathbf{P}}_k$ and the 3D points $\hat{\mathbf{X}}_i$ which have projections minimising the least squares distance to the known image points $\mathbf{x}_i$. Since this is a minimisation problem of an enormous scale, usually sparse techniques like Levenberg-Marquardt minimisation are employed. Notably, this approach basically implements the techniques used in the evaluation chapter of this dissertation as a "feedback loop" to the structure-from-motion algorithms, thereby aiming to obtain an optimal reconstruction.

Some of these techniques, especially trifocal tensors [12, p. 365–ff.] and bundle adjustment [12, p. 434–ff.] are well-described in literature already, whilst others, such as self-calibration of hand-held cameras, are still an open area of research.

## A.7 Derivation of $n$-view Homogeneous DLT Triangulation

The central component of linear triangulation algorithms is a system of simultaneous linear equations, represented as a matrix $\mathtt{A}$. This matrix is constructed by considering that for each point in images $i, ..., j$ there is a relation of the form $\mathbf{x}_i = \mathtt{P}_i\mathbf{X}$ (this is just the algebraic expression for the projection), although $\mathbf{x}_i$ and $\mathtt{P}_i\mathbf{X}$ can differ in magnitude by an arbitrary non-zero scale factor.

This relationship can also be expressed by saying that the cross product of the two vectors must be the zero vector, *i.e.* $\mathbf{x}_i \times \mathtt{P}_i\mathbf{X} = \mathbf{0}$. This cross product can be written as a set of equations linear in the components of $\mathbf{X}$ (let $\mathbf{p}_i^{k\mathsf{T}}$ denote the $k^{\text{th}}$ row of the $i^{\text{th}}$ camera matrix $\mathtt{P}_i$):

$$\begin{aligned}
x_i(\mathbf{p}^{3\mathsf{T}}\mathbf{X}) - (\mathbf{p}^{1\mathsf{T}}\mathbf{X}) &= 0 \\
y_i(\mathbf{p}^{3\mathsf{T}}\mathbf{X}) - (\mathbf{p}^{2\mathsf{T}}\mathbf{X}) &= 0 \\
x_i(\mathbf{p}^{2\mathsf{T}}\mathbf{X}) - y_i(\mathbf{p}^{1\mathsf{T}}\mathbf{X}) &= 0.
\end{aligned}$$

Of these three equations, only two are linearly independent since the third is a linear combination of $y_i$ times the first plus $x_i$ times the second. Including two equations from each image only, this cross product can be written out as a set of $2n$ simultaneous equations in $2n$ homogeneous unknowns for $n = l - k$ images:

$$\underbrace{\begin{bmatrix} x_k\,\mathbf{p}_k^{3\mathsf{T}} - \mathbf{p}_k^{1\mathsf{T}} \\ y_k\,\mathbf{p}_k^{3\mathsf{T}} - \mathbf{p}_k^{2\mathsf{T}} \\ \vdots \\ x_l\,\mathbf{p}_k^{3\mathsf{T}} - \mathbf{p}_l^{1\mathsf{T}} \\ y_l\,\mathbf{p}_k^{3\mathsf{T}} - \mathbf{p}_l^{1\mathsf{T}} \end{bmatrix}}_{\mathtt{A}} \mathbf{X} = \mathbf{0}.$$

In order to solve a set of equations of the form $\mathtt{A}\mathbf{X} = \mathbf{0}$, two methods can be utilised: The *homogeneous DLT method*, and an *inhomogeneous method* [12, p. 313]. The former was implemented for this project, as described in Section 3.3.3.

## A.8 Test Content

### A.8.1 Content Creation and Acquisition

The test content that I used whilst developing and evaluating the *Proteus* project was mostly obtained by myself using a Panasonic Lumix DMC-TZ5 digital camera, taking short video shots of various objects. The "Hotel" example was created from a set of images[2] obtained from the Carnegie Mellon VASC Image Data Base, which supplies a number of public domain image sequences for computer vision research. The images were converted into a video sequence that could be used with *Proteus*.

### A.8.2 List of Test Content Sequences

| No. | Sample Frame | Description |
|---|---|---|
| 1 | | **Box**<br>00:20 min., M-JPEG, 640×480, 30fps<br>*Used for:* Basic projective reconstruction development, texture extraction tests |
| 2 | | **Blocks**<br>00:10 min., M-JPEG, 640×480, 30fps<br>*Used for:* Annotation tests, user study |
| 3 | | **Cube**<br>00:07 min., M-JPEG, 640×480, 30fps<br>*Used for:* Metric reconstruction development, texture extraction tests, user study |
| 4 | | **Laptop**<br>00:17 min., M-JPEG, 640×480, 10fps<br>*Used for:* Texture extraction development, annotations with noisy input data |
| 5 | | **Hotel**<br>00:04 min., MPEG-4, 512×480, 25fps<br>*Used for:* Testing automated feature tracking and projective reconstructions |
| 6 | | **Rickroll**<br>03:35 min., MPEG-1, 352×240, 30fps<br>*Used for:* Basic video decoding tests |

[2] "hotel" sequence in the "Motion" category `http://vasc.ri.cmu.edu//idb/html/motion/hotel/index.html`

## A.9 User Study Material

### A.9.1 Identification of Variables

Since the main user interface had mainly been designed to be functional (e.g. not using toolbars or visual keys to represent different options), an evaluation of this was deemed to be unnecessary – the interface itself can be refined independently of the underlying system and could be redone completely for a commercial implementation.

The annotation mechanism itself however does present a number of variables that lend themselves to evaluation:

1. Perceived helpfulness of interactive guidance through "snapping".
2. Perceived accuracy of the snapping process.
3. User appeal of the idea of "drawing on video".
4. Overall ease of use and intuitivity.
5. Comparison to other modelling techniques.

All of these variables were included in the user study.

### A.9.2 Informed Consent Form

Proteus – GUI Evaluation Study

Informed Consent Form

I wish to participate in an user evaluation study which is being conducted by Malte Schwarzkopf, a C
Science student at the University of Cambridge Computer Laboratory. The purpose of the research is
the results of work carried out during a final year project. The study involves use of a graphical user
and subjective assessment of different features. My name will not be identified at any time, but co
make may be anonymously quoted. I understand that I am free to withdraw from participation at m
any time.

I agree to the above Informed Consent Form.

______________________________

Signed

### A.9.3 Outline of running the Study

There are three sample videos: **Cube**, **Blocks** and **Disk**.

The user is given a basic explanation of how the Proteus GUI front end works (see below) and is asked to annotate a video sequence. The number of annotations they can make is not constrained; instead an average "time per annotation" value will be taken later to compensate for different amounts of detail being used.

> *"Proteus is an interactive modeling tool. It works by the user, i.e. you, putting annotations onto a video and then relates them to generate a three-dimensional representation.*
> *This study aims to evaluate how well different techniques in how the user interface guides you when annotating a video work.*
> *You will now be given a video and I am asking you to "annotate" it by drawing outlines onto it, highlighting the edges in the video. You can start by left-clicking the point where you want to start and it should be intuitive from there. You can terminate a sequence of lines by double-clicking the right mouse button."*
> [ Before doing case 2:
> *"You can also disable the magnetic snapping by holding down the* CTRL *key."* ]

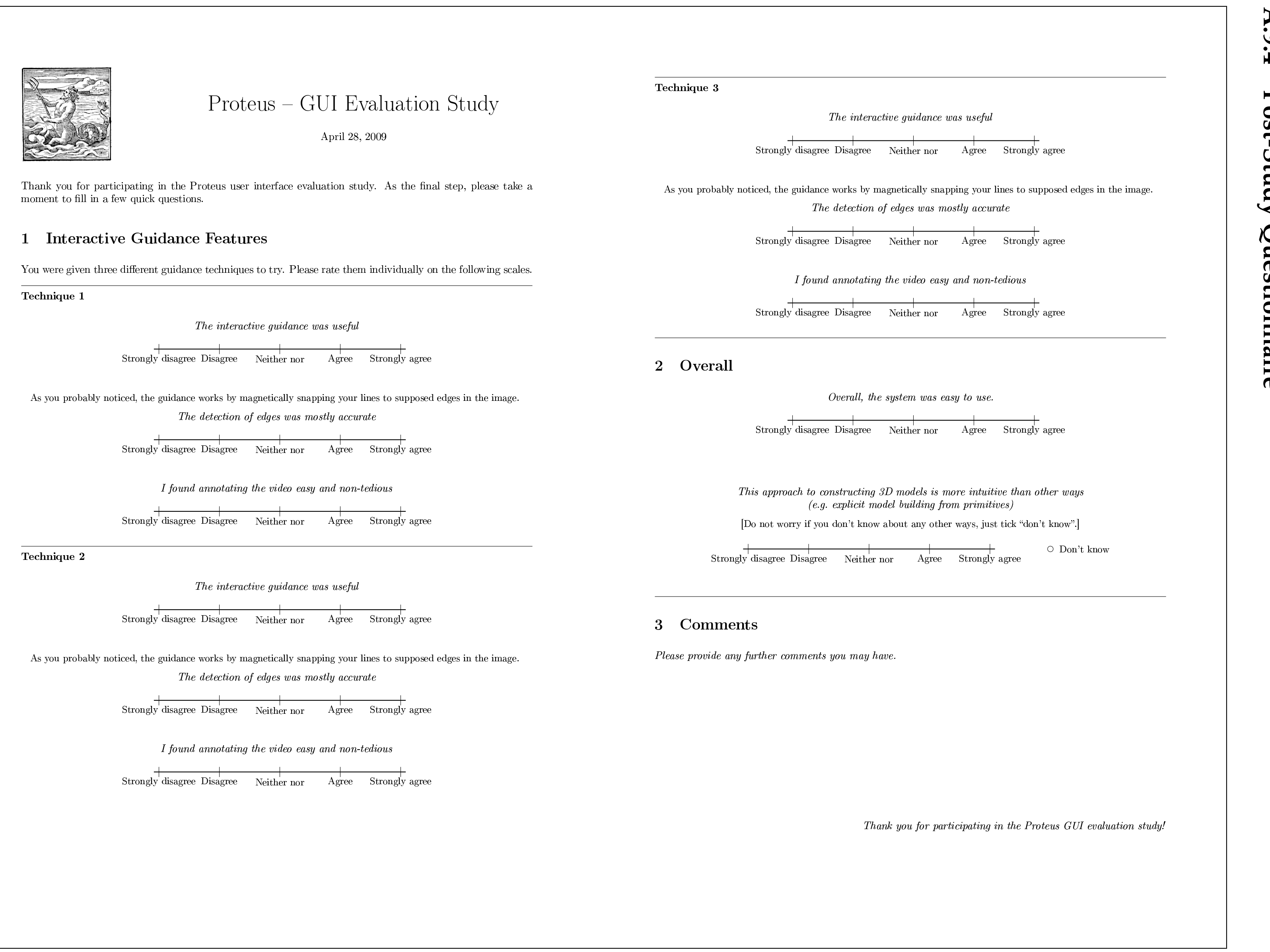
There are three annotation modes that are being evaluated:

1. Snapping only, using the Sobel operators
2. Snapping and no snapping, using the Sobel operators
3. No snapping

The user is asked to annotate a video using these 3 techniques. The time taken is measured and normalised by the number of annotations made. Different subjects are asked to annotate different videos and with the different in different orders, compensating for the learning effect.

At the end, the user is given an evaluation form and is asked to rate the different techniques in terms of ease of use. This data will be related to the measured times to compare subjective and objective perception.

### A.9.4 Post-Study Questionnaire



# Proteus – GUI Evaluation Study

April 28, 2009

Thank you for participating in the Proteus user interface evaluation study. As the final step, please take a moment to fill in a few quick questions.

## 1 Interactive Guidance Features

You were given three different guidance techniques to try. Please rate them individually on the following scales.

**Technique 1**

*The interactive guidance was useful*

Strongly disagree | Disagree | Neither nor | Agree | Strongly agree

As you probably noticed, the guidance works by magnetically snapping your lines to supposed edges in the image.

*The detection of edges was mostly accurate*

Strongly disagree | Disagree | Neither nor | Agree | Strongly agree

*I found annotating the video easy and non-tedious*

Strongly disagree | Disagree | Neither nor | Agree | Strongly agree

**Technique 2**

*The interactive guidance was useful*

Strongly disagree | Disagree | Neither nor | Agree | Strongly agree

As you probably noticed, the guidance works by magnetically snapping your lines to supposed edges in the image.

*The detection of edges was mostly accurate*

Strongly disagree | Disagree | Neither nor | Agree | Strongly agree

*I found annotating the video easy and non-tedious*

Strongly disagree | Disagree | Neither nor | Agree | Strongly agree

**Technique 3**

*The interactive guidance was useful*

Strongly disagree | Disagree | Neither nor | Agree | Strongly agree

As you probably noticed, the guidance works by magnetically snapping your lines to supposed edges in the image.

*The detection of edges was mostly accurate*

Strongly disagree | Disagree | Neither nor | Agree | Strongly agree

*I found annotating the video easy and non-tedious*

Strongly disagree | Disagree | Neither nor | Agree | Strongly agree

## 2 Overall

*Overall, the system was easy to use.*

Strongly disagree | Disagree | Neither nor | Agree | Strongly agree

*This approach to constructing 3D models is more intuitive than other ways*
*(e.g. explicit model building from primitives)*

[Do not worry if you don't know about any other ways, just tick "don't know"]

Strongly disagree | Disagree | Neither nor | Agree | Strongly agree ○ Don't know

## 3 Comments

*Please provide any further comments you may have.*

*Thank you for participating in the Proteus GUI evaluation study!*

## A.10 Original Project Proposal

Malte Schwarzkopf
St John's College
ms705

Computer Science Tripos Part II Individual Project Proposal

# Proteus – An interactive 3D model acquisitor

October 23, 2008

**Project Originator:** Malte Schwarzkopf and Christian Richardt

**Resources Required:** See attached Project Resource Form

**Project Supervisor:** Christian Richardt

**Signature:**

**Director of Studies:** Dr Robert Mullins

**Signature:**

**Overseers:** Dr Frank Stajano and Dr David Greaves

**Signatures:**

## Introduction and Description of the Work

In recent years, consumer-grade digital video capture equipment has become much more affordable, whilst at the same time the computing power available at low cost increased. On a similar ground, 3D computer graphics hardware has become a lot faster and more affordable. These factors have led to these particular uses of digital technology becoming more ubiquitous. Fast broadband connections and the emergent “web 2.0” phenomenon that gave rise to platforms such as YouTube have similarly contributed to giving home-made videos a much more prominent role in our everyday lives and culture.

These factors have led to an increasing research interest in combining the classical 2D image and video processing technology with three-dimensional computer graphics. Structure-from-motion and the idea of acquiring 3D models from two-dimensional image material is a specific branch of a broader research area, and one that has seen some significant steps forward recently, yet still seems to bear a lot of promise and potential.

A lot of research effort has been put into analysing video data in order to obtain information about objects for guidance systems or recognition problems, however this has usually been limited to either a poor visual quality or utilised very specific hardware setups and highly specialised equipment. Little consideration has been given to the idea of providing home users, with often limited computing experience, with the necessary tools to transform videos into 3D world models, using the consumer-grade equipment available to them.

The aim of this project is to implement a simple structure-from-motion system that can generate a 3D model of an object in a scene from a video, using inputs interactively provided by the user to assist the model extraction.

It is well known that the human visual system is well-suited to recognising shapes and objects in an image, and can do so more efficiently than current image processing technology can. In order to utilise this in the context of a 3D model acquisitor that uses video as its input data, an interface that allows the user to “annotate” a video frame with object information, hence pointing out distinctive features in the object such as corners, edges and shapes. In order to assist the user in this process and provide some additional guidance, a "snapping" concept, which snaps lines to edges in the video, will be implemented.

After the first key frames have been annotated, the software will attempt to use the data provided by the user to estimate how the perspective has changed relative to another video frame. Features are then interpolated between frames, and the user is give an opportunity for corrective measures in case the interpolation has produced an inaccurate result.

Ultimately, a completely annotated sequence of video frames should have been obtained. This is then used to extract the 3D structure of the object in the video. In the implementation planned for this project, some information will already be provided by the interactive annotations on the video, so that the step of feature identifcation and matching can be simplified significantly compared to fully automatic systems.

## Resources Required

In order to model a real-world use case as accurately as possible, the project will use videos taken using a hand-held digital camera as sample input material. Please see the attached project resource form for details on this.

Apart from the camera, there are no special requirements. I am planning to undertake development on my personal machines, doing regular automated and manual backups to off-site locations such as the PWF and the servers provided by the SRCF.

## Starting Point

In proposing this project, I am planning to build on the following resources:

- **Internships at Broadcom Europe in 2007 and 2008** – I worked as a summer intern in the 3D graphics group of the Mobile Multimedia department at Broadcom Europe Ltd. for two summers. My work was primarily related to 3D hardware, but I was exposed to the basic principles OpenGL and 3D computer graphics.
- **Previous programming experience** – I have a good working knowledge of C and have in the past implemented projects of significant size in C. I am familiar with C++, but haven't used it much beyond the Part IB lecture course.

## Substance and Structure of the Project

In order to ensure successful completion of the project, it has to be divided into manageable chunks of work, and there have to be various levels of fallback options to compensate for possible difficulties and issues that might delay the implementation.

For this project, there are three central stages that each consist of a number of sub-tasks that need to be completed:

1. *Interactive video annotation toolkit:* This step will provide the user with a very simple GUI framework that can be used to "annotate" key frames in a video with feature information. There is a obvious and direct relation between the number and distance of key frames, the accuracy of the annotations, and the quality of the reconstruction of the object in a 3D model. This stage of the project involves the following tasks:
    - Creating a video annotation framework: This will be a simple C/C++ implementation of an image viewer that displays individual frames of a video and that has got an annotation framework on top, allowing for a wireframe to be drawn onto the image. For video decoding, the *libavcodec* library, which is part of the open source *ffmpeg* cross-platform video library, is intended to be used; the user interface toolkit to be used will either be Gtk+ or Qt, both of which are free and cross-platform.

- Implementing interpolation: For the frames in between key frames, the annotations from previous keyframe are copied and an estimation of their possible relative displacement is made. The user will be allowed to edit them (move, add and remove vertices).
- Implement "snapping": Inspired by the concept outlined in [1], snapping of feature points to to detected edges or feature outlines in the image as well as other feature points will be implemented. This will mean that lines drawn on the model by the user will "snap" to the most likely location, thereby correcting for small inaccuracies between user input and image data, allowing for a rapid, yet accurate annotation process.

2. *3D model generation:* From the annotated video data, proceed through extracting the relative camera motion between frames, and hence the camera perspectives in the individual frames, using principles of epipolar geometry (projective reconstruction) on to regenerating a metric representation of the scene geometry and rectification of the different views. Finally, generate depth maps and from those, generate spatial surfaces that can then be used to build the 3D model. Display the final model using an OpenGL canvas. An approach to this is presented in [19], although the assumption here is that all information about the object has to be extracted automatically and autonomously from the video data by the system.
3. *Texture extraction and projection:* Implement a simple texture extraction scheme whereby the "best" frame is determined (i.e. the one where most of a plane/polygon is visible, ideally a direct front-facing shot) and extract the bitmap texture data from the video for this frame. Apply to the generated output polygon. Note that this requires working out what the OpenGL varyings for this texture mapping should be.

**Possible Extensions**

1. *Scene background extraction:* Extract the (static) scene background and project it onto a background plane in the generated 3D scene.
2. *Export of 3D model data into some well-supported format:* Implement support for exporting the generated structures to a free OpenGL model format so that it can be used with any application understanding this format.

## Success Criteria

The primary success criteria for this project are detailed in the following. These criteria are essential for the project to be considered a success.

1. A video annotation framework that supports annotation of video frames and implements "snapping" of feature points selected, thereby guiding the user in annotating the video, has been implemented.

2. An interactive structure-from-motion implementation that uses the feature points supplied by interactive annotation has been implemented. It is shown to be working by applying it to a 360° shot of a cube or some other distinctive geometric shape.

3. Textures for the faces in the extracted 3D model are obtained from the input image data using per-polygon texture extraction. It is shown to be working by applying the method to a textured cube.

In terms of possible extensions of the project, implementing one or more of the following would increase the success of the project, but this is not essential.

1. The final 3D model generated can be exported into a well-supported model format so that it can be used in real-world applications (such as Google Earth, Second Live or a 3D computer game).

2. Extraction of the invariant scene background and projection onto a bounding box to create the illusion of the 3D object being in the surroundings it was captured in.

## Timetable and Milestones

There are four and a half months of time available to do this project. I am aiming to finish the implementation work by the middle of March, i.e. the beginning of the Easter vacation.

I estimate that the work will be partitioned into 3 week slots as follows:

**Slot 0: October, 1$^{st}$ – October, 17$^{th}$**

- Research, familiarisation with literature on the subject and analysis of different existing concepts.
- Work out an approach to do the feature data to 3D conversion.
- Investigate libraries and write test programs to utilise them and for familiarisation.

**Milestone:** Write project proposal.

**Slot 1: October, 18$^{th}$ – November, 7$^{th}$**

- More research and reading. Familiarise with ideas of epipolar geometry and the mathematical foundations of the processes of projective reconstruction, self-calibration, rectification and model generation.
- Start on implementing the annotation framework. Have facility to load and decode video and to display individual frames and draw annotations onto them.

**Slot 2: November, $8^{th}$ – November, $28^{th}$**

First part of the core implementation phase.

- Get annotation framework to work and have per-frame annotations working together with techniques to interpolate the feature annotations between frames.
- Implement "magnetic snapping" of lines/features to regions in the image.

**Milestone:** Have a working annotation interface, including video decoding and automatic "snapping" of feature points at the end of this time slot.

**Slot 3: November, $29^{th}$ – December, $19^{th}$**

Second part of the core implementation phase.

- Implement first iteration of the core model acquisition code: Projective reconstruction from data provided by the annotations, self-calibration of camera parameters, metric reconstruction.
- I will also need to write some auxiliary code at this point in order to be able to test and debug the different steps of the model acquisition process whilst not yet having a final output; this might make development more tedious.

**Slot 4: December, $20^{th}$ – January, $9^{th}$**

Third part of the core implementation phase.

- Add final stages of model acquisition code: Depth maps, surface generation and model building. Tackle any issues with the earlier stages.
- Implement generation of the OpenGL data structures used for models and implementation of a display canvas.
- Implement simple texture extraction by using the best possible texture for a face found in a frame and applying it to the face in the final model.

**Milestone:** Working implementation of model acquisition and -generation for simple objects.

**Slot 5: January, $9^{th}$ – January, $30^{th}$**

Progress report, implementation of extensions and catchup time.

- Finish texture extraction implementation, trying to refine the quality of the extracted textures and their projection onto surfaces.
- Start implementing one or several of the optional extensions to the project.

- Write progress report and hand it in.

**Milestone:** Present progress, have a fully texture simple geometric object being recreated using interactive annotation by the end of the time slot.

**Slot 6: January, 31$^{st}$ – Feburary, 20$^{th}$**

Further extension implementation and catchup time.

- Continue implementing optional extensions to the project.
- Investigate further extension possibilities that might have arisen during development and possibly implement extra features.

**Milestone:** Working implementation of one or several extensions, depending on the choice made.

**Slot 7: Feburary, 21$^{st}$ – March, 13$^{th}$**

Catchup time. Start writing dissertation.

- Buffer time to catch up any delay incurred at previous stages.
- Fix remaining bugs and outstanding issues and optimise some parts of the code. Take measurements and benchmarks and analyse the results for evaluation.
- Start writing the dissertation by producing an outline and writing the introductory sections.

**Slot 8: March, 14$^{th}$ – April, 3$^{rd}$**

Write dissertation up to draft standard.

- Write the main parts of the dissertation. Generate figures, intergrate results and evaluation.
- Typeset the dissertation and produce a draft to be handed in.

**Milestone:** Dissertation draft finished by the end of the timeslot.

**Slot 9: April, 4$^{th}$ – April, 24$^{th}$**

Finish dissertation.

- Finish off the dissertation. Add more benchmarking and evaluation results.
- Address any formatting and typesetting-related issues.
- Incorporate comments on draft received from supervisor and proofreaders.

**Slot 10: April, 25$^{th}$ – May, 15$^{th}$**

Buffer time, dissertation deadline at the end of the timeslot.

- Address any remaining issues with the dissertation and hand it in when done.
- Use the remaining time for revision for the exams.